2001

Distributed query processing using composite semijoins.

Ma. Lei
University of Windsor

Follow this and additional works at: http://scholar.uwindsor.ca/etd

Recommended Citation
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.
Distributed Query Processing Using Composite Semijoins

BY

Ma Lei

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
University of Windsor

Windsor, Ontario, Canada
2000
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.
Abstract

The utilizing of semi-join is often a common starting point for join algorithms in distributed databases. It helps reduce the quantity of data transferred between sites. In our thesis, we propose an algorithm, based on the semi-join operator. By utilizing the maximum reduction capability of the semi-join operation, we use our algorithm to reduce the query relations as much as possible. In order to improve the reduction ability of our algorithm, we combine composite semi-joins into our algorithm. Usually, composite semi-join may produce more reduction than separate simple semi-joins in our algorithm with more time costs. Although a composite semi-join itself may not be beneficial because of its more total time costs, it always is gainful to the execution of subsequent join operations. Our proposed algorithm is evaluated objectively against the effects of a full reducer and the total cost of initial feasible solution (IFS). It has been shown that the algorithm gives substantial reductions on relations and total costs.
To my parents, my brothers and my wife Xiaoan
Acknowledgements

I would like to acknowledge the guidance provided by Dr. J. Morrissey, whose time, dedication and effort has contributed in guiding me through this thesis. Without her patience and guidance it would have been impossible to complete this thesis in this time frame. I am grateful to Dr. I. Schmitt for proving valuable suggestions and comments that aided in my thesis. I would also like to thank Dr. Q. Wen for being willing to serve as my external reader.

Special thanks to my wife Xiaoan and my brother Xiaobo for their encouragement and support.
CONTENT

Abstract ................................................................. iv
Acknowledgements ..................................................... vi
1. Introduction .......................................................... 1
2. Background .......................................................... 4
2.1 Three Phase In Distributed Query Optimization ................. 4
2.2 Cost Model ........................................................ 6
2.3 Definitions ........................................................ 6
2.4 Semijoin-based Approaches ..................................... 9
2.5 Filter-Based Approaches ....................................... 15
2.6 Specialized Semi-joins .......................................... 20
2.6.1 Two-way Semi-joins ......................................... 20
2.6.2 Domain Specific Semi-joins ................................. 22
2.6.3 Composite Semi-join ......................................... 23
3. The Algorithm ..................................................... 24
3.1 Motivation ........................................................ 24
3.1.1 Query Graph and Full Reduction .......................... 24
3.1.2 Bloom Filter and its Collision ............................. 26
3.1.3 Hypothesis .................................................. 27
3.2 The Algorithm .................................................. 29
3.2.1 Details of The Algorithm .................................. 30
3.2.2 An Example of Using The Algorithm ........................................... 31
4. Experiments ................................................................. 41
4.1 The Queries .............................................................. 41
4.2 Individual Runs ......................................................... 43
4.3 Full Reducer ............................................................. 44
4.4 Initial Feasible Solution (IFS) and Total Time Cost .................. 47
5. Results of Evaluation ..................................................... 49
5.1 Comparison against Full Reduction .................................. 49
5.2 Comparison against IFS ................................................. 52
5.3 Discussion .............................................................. 54
6. Conclusion ................................................................. 58
Bibliography ................................................................... 60
LIST OF FIGURES

Figure 2-1  The semijoin $R_i \bowtie R_j$ over attribute B ........................................ 10
Figure 3-1  A example of query graph ................................................................. 25
Figure 3-2  Comparison between semi-join and composite semi-join .................... 28
Figure 3-3  The five relations of the example ....................................................... 32
Figure 3-4  Query Graph of the query example .................................................... 33
Figure 3-5  Adjacency lists ...................................................................................... 33
Figure 3-6  After step 2 ......................................................................................... 34
Figure 3-7  After step 3 ......................................................................................... 35
Figure 3-8  After step 4 ......................................................................................... 36
Figure 3-9  After step 5 ......................................................................................... 37
Figure 3-10 After step 6 ......................................................................................... 38
Figure 3-11 After step 7 ......................................................................................... 38
Figure 3-12 After step 8 ......................................................................................... 39
Figure 3-13 After step 9 ......................................................................................... 39
Figure 3-14 The five reduced relations after the algorithm .................................... 40
Figure 4-1  The query generator program ............................................................... 43
Figure 4-2  Result of joining of five relations ........................................................... 45
Figure 4-3  Results of fully reduced relations ......................................................... 46
Figure 5-1  Comparison with fully reducer ............................................................. 50
Figure 5-2  Comparison with IFS  ... ... ... ... ... ... ... ... ... ... ... ...52
Figure 5-3  Propagation of benefits  ... ... ... ... ... ... ... ... ... ... ... ...55
1. INTRODUCTION

A distributed database management system (DDBMS) is characterized by the distribution of the system components of hardware, control, and data. Distributed database management systems offer many advantages including increased reliability, more efficient processing of local queries and increased data availability.

As with traditional centralized databases, distributed database systems must provide an efficient user interface that hides all of the underlying data distribution details of the DDBMS from the users. The use of a relational query allows a user to specify a description of the data that is required without having to know where the data is physically located. The retrieval of data from different sites in a DDBMS is referred to as distributed query processing [ES80]. Such query processing often requires shipping relations between different sites. Furthermore, given a query, there may exist many strategies that a distributed database management system can choose to execute to generate the answer. All these strategies are equivalent in the sense of their final answers but they may vary in their execution costs. Therefore the distributed DBMS must have a module to examine alternatives and choose the “best” strategy that costs the least. This module is called a distributed query optimizer [YE96]. Hence, query optimization in such systems is a very important task. The objective is to select a sequence of operations and data transfers so that data transmission cost is minimized. Query optimization research has been explored
extensively in the last 20 years. Many works concentrated on joins [CY94], semi-joins [AHY83, EW81, VV84], Bloom filter [WO98] or a combination of joins and semi-joins [CY90, CY92, CY93].

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 cost. To solve these problems the semi-join was proposed and in most cases semi-join based algorithms are very efficient. However, sometimes, semi-joins may still require high data transmission. 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 that 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 our thesis, we are still concerned with a semijoin based strategy, where semijoins are executed so as to reduce the size of intermediate relations and thus minimize the time required to process the query. More specifically, in order to improve the reduction ability of semi-joins, we combine composite semi-joins into our algorithm. Composite semijoins may produce more reduction than separate simple semijoins in our algorithm. The proposed algorithm is evaluated, but not against other algorithms, instead against the effects of a full reducer and initial feasible solution (IFS). The algorithm is evaluated to determine how close it comes
to achieving full reduction of query relations under various conditions and how much it costs in terms of total time costs. As a result, our primary goal is to investigate if a semijoin-based algorithm can perform significant reduction under feasible costs.

This thesis is organized into six sections. In section 2, some fundamental concepts and techniques for the optimization of distributed query processing will be reviewed as the basis for our thesis. Section 3 includes the motivation of my thesis work and our proposed algorithm is represented in detail. The experimental system and its rationale for the evaluation will be discussed in Section 4. In section 5, the results of the evaluation are presented followed by the discussion about our algorithm. Finally, conclusions are made in section 6.
2. BACKGROUND

Query optimization is a large area within the database field and has been investigated extensively [MJ84]. However, the problem of optimal query processing in distributed database systems was shown to be NP-hard [CL84, YHL89]. So, most algorithms for general query are based on heuristics, which give efficient, close to optimal solutions.

The initial research in this area was done by Wong [EW77]. He proposed an optimization method based on a greedy heuristic that produces efficient, but not necessarily optimal query processing strategies. An enhanced version of this method is implemented in the SDD-1 system [EW81]. Motivated by this method, many query optimization algorithms have been developed. The intention of this section is to outline a number of query optimization algorithms. Some definitions will be given, then some representative approaches in the area of distributed query optimization will be presented.

2.1 Three Phase Approach In Distributed Query Optimization

In a distributed relational database system, the processing of a query involves data transmission among different sites via a computer network. Since data are geographically distributed in such a system, the processing of a distributed query, as pointed out in [AHY83, EW81, KP87, YC84, CM96, YOK84], is composed of the following three phases:
1) **Local processing phase** which involves all local processings such as selections and projections. All irrelevant tuples and attributes are filtered out by appropriate selection, projection, and join operations at the local sites prior to any data transmission. This has the effect of reducing the amount of data transmitted over the network. The optimization on this phase is sometimes called a local optimization level.

2) **Reduction phase** where a sequence of reducers (i.e., semi-joins and joins) is used to reduce the sizes of relations. In this phase, all unneeded tuples from the materialized relations are eliminated by some distributed data reducing techniques. In a distributed database environment, it is common that queries access data from different sites. In such situations, it is reasonable to attempt to limit the amount of data transfer across the network by permuting the order of operations in a query. Comparing with local optimization level, it sometimes is called a global query optimization level. Two major tools, the semijoin operation and Bloom filters, which will be discussed later, are often employed in the global query optimization level.

3) **Final processing phase** in which all resulting relations are sent to the assembly site where the final query processing is performed. In this phase, the remaining tuples in each materialized relation are sent to the querying site, and, the arriving relations are processed at the querying site to complete the query.
2.2 Cost Models

The goal of processing a distributed query is to derive a sequence of relational operations, or an execution strategy, which incurs the minimum cost. Several cost models have been proposed. The two most popular are the total cost model and the response time cost model. The total cost model includes both the data transmission cost and the local processing cost. However, most heuristics assume that the local processing cost is negligible in comparison to the data transmission cost. Therefore, in most cases, the total cost model calculates the cost of data transmissions only. The response time cost model calculates the total execution time of the query from the beginning to the calculation of the final result. In the latter case, most heuristics make assumptions concerning network line contention and queueing delays which simplify the cost calculation.

2.3 Definitions

The relational data manipulation operations used in our thesis are projection, selection, join, semi-join, and two-way semi-join, and Bloom filter. They are described as follows:

**Projection.** The projection of relation R on a set of attribute D is denoted by R, D or R (D), where R is a relation with schema X and D is subset of X. It is obtained by discarding all columns of R that are not in D, and eliminating duplicated rows, if necessary [PM92].
**Selection.** The selection of those tuples whose A-attribute values equal to a specified constant in relation R is denoted by (R. A = the specified constant), where A is an element of X. It is obtained by choosing all rows of R whose A-attribute values are equal to the specified constant. One or more select clauses on the same relation may be used in selection. Operators other than "=" (e.g., ≥ and ≠) are allowed [PM92].

**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 (or joining domains) [MAX97].

**Join.** The join of relation Rᵢ with relation Rⱼ on attribute A is denoted by (Rᵢ. A ⊙ Rⱼ. A), where Rᵢ and Rⱼ are the joining relations. Let X and Y be the schema of Rᵢ and Rⱼ, respectively. The attribute A, which is an element of X and Y, is the joining attribute of Rᵢ and Rⱼ. The join is obtained by concatenating each row of Rᵢ with each row of Rⱼ whenever the A-attribute values of the two rows are equal. Since the equality operation results in two identical columns, one column may be eliminated. One commonly used join operation is the natural join, where two rows from the joining relations are concatenated whenever the corresponding values under all common attributes of the two relations are equal [PM92].

**Semi-join.** The semi-join from relation Rⱼ to relation Rᵢ on attribute A is denoted by Rⱼ-A→Rᵢ, where Rⱼ is the sending relation, Rᵢ is the reduced relation,
and A is the joining attribute. Sometimes, the semi-join from relation R_i to R_j can be written as R_i \bowtie R_j. A semi-join R_i \bowtie R_j over the join attribute A is executed as: projecting R_j over attribute A to get the projection R_j[A], shipping the projection R_j[A] to the site of relation R_i, and then executing R_j[A] \bowtie R_i [MWB96].

**Two-way semi-join.** As the name implies, the two-way semi-join of relation R_j by relation R_i over attribute A is obtained by performing two semi-joins. The first being the semi-join of relation R_j by relation R_i and the second being the semi-join of relation R_i by relation R_j [KP87]. It can be said that a 2-way semi-join is obtained by adding backward reduction to the semi-join.

**Bloom Filter.** In order to reduce the cost of data transmission, a search filter, which represents the semi-join projection with a small bit array, is used. This kind of search filter is also called a Bloom filter. Bloom filtering provides an alternative way to reduce the local processing load of regular relational operations and thus avoids the creation of large intermediate relation results for transmission [MAX97].
2.4 Semijoin-based Approaches

The utilizing of the semijoin is often a common starting point for query algorithms in distributed databases. It helps reduce the quantity of data transferred between sites.

Let the two relations for a join operation reside at different site, say \( R_i \) and \( R_j \). We can ship both relations directly to the query site and perform the join there. Alternatively, we can reduce one or both relations using semijons before shipment. The symbol \( R_i \bowtie R_j \) is generally accepted for denoting a semi-join on \( R_j \) with \( R_i \).

The semi-join operation of \( R_j \) with \( R_i \) over attribute \( B \) is executed as follow [MWB96]:

1) Project relation \( R_j \) over attribute \( k \) to get the projection \( d_{jB} \).

2) Ship the projection \( d_{jB} \) to the site of relation \( R_i \).

3) Execute \( d_{jB} \bowtie R_i \) on the site of relation \( R_i \).

The semijoin reduces the size of \( R_i \) by eliminating the tuples which cannot be part of \( R_i \bowtie R_j \). The reduced \( R_i \) can now be shipped to the query site with savings on transmission costs. Figure 2-1 gives an example to show how the semijoin works.
Figure 2-1 The semijoin $R_i \bowtie R_j$ over attribute B

Besides the SDD-1 optimizer [EW81, EW77], the AHY algorithm is another semijoin based query processing algorithm. This is a collection of algorithms, presented by Apers, Hevner and Yao in [AHY83]. Two algorithms called algorithms PARALLEL and SERIAL are presented first. These two algorithms are developed for a special class of simple queries. Algorithm PARALLEL is for the minimization of response time and Algorithm SERIAL is for minimization of total time. In order to process general distributed queries, Hevner and Yao extended the previous two algorithms to a new algorithm called Algorithm GENERAL
[AHY83], which is later called AHY algorithm by other database researchers. This algorithm investigates two optimization objectives: the minimization of response time and the minimization of total time. The response time of a schedule is the time that elapsed between the start of the first transmission and the time at which the relation arrives at the required computer. The total time of a schedule is the sum of the costs of all transmissions required in the schedules. In [AHY83], three versions of Algorithm GENERAL are proposed: one for minimizing response time and two for minimizing total time.

In [VV84], a new algorithm called Strategy V is presented. It is improved from algorithm Method-D [CTY80] and Algorithm GENERAL [AHY83]. Unlike the AHY algorithms, it takes into account the local processing cost. The total cost includes both the communication cost and the local processing cost.

Most semi-join algorithms favor executing semi-joins sequentially such that the reduction effect of a semi-join can be propagated to reduce the costs of other semi-joins. For example, the cost of $R_i \rightarrow R_j$ may be lowered if another semi-join $R_k \rightarrow R_i$ is executed first. Such semi-join processing strategies are sometimes inefficient for some reasons, such as loss of parallelism, processing overhead, loss of global semi-join optimization, and inaccurate semi-join reduction estimation [WLC91]. To alleviate the above problems, Wang, Li and Chen propose a new semi-join processing procedure, named one-shot semi-join execution [WLC91]. Under this method, all applicable semi-joins to the relations are executed at one time. That means, the semi-join processing at all sites can be performed simultaneously. As a
result, each relation needs to be scanned only once to process all applicable semi-joins. Moreover, no inaccurate estimation of the semi-join cost and benefit will be propagated.

Later, Chen and Yu presented a series of papers to explore the approach to apply a combination of join and semi-join operations to minimize the amount of data transmission required for distributed query processing [CY90, CY92, CY93]. They defined two important concepts, namely, gainful semi-joins and pure join attributes. Some semi-joins, though not profitable themselves, may benefit the execution of subsequent join operations, and become profitable owing to the use of join operations as reducers. Such a semi-join is termed a gainful semi-join. Also, join attributes, which are not part of the output attributes, are referred to as pure join attributes [CY90]. Based on these two concepts, they use several heuristic searches to determine a sequence of join and semi-join reducers for tree and cyclic queries. Their approach did not only exploit the usefulness of gainful semi-joins, but also utilizes the removevability of pure join attributes to reduce the communication costs required for subsequent operations. Integrated from the work of Chen and Yu [CY90, CY92, CY93], B. Rho and S.T.March gave a comprehensive query optimization model [SSM97]. This model also uses the concepts of gainful semi-joins and pure join attributes. First, the algorithm integrates copy identification, join order, join site selection, and types of joins for a query into a single model. Then a generic algorithm-based solution is developed for the model. This model considers both local join processing costs and communication costs. In [CY94], Chen and Yu
presented another heuristic in which they use join operations as reducers in a
distributed query. They mentioned that judiciously applying join operations as
reducers can lead to further reductions in the amount of data transmission required.
The concept of a complete and feasible (CF) set of cuts to a query graph is given at
first. A sequence of join operations for a query can be transformed to the one of
finding a CF set of cuts to the corresponding query graph. Then, a heuristic
algorithm based on the concept of divide and conquer is used to determine a
sequence of join reducers for query processing.

Dynamic programming, which is a standard optimization technique, has
been employed as a search strategy in query optimization. The dynamic
programming based query optimizer first enumerates possible partial plans and
estimates the corresponding costs, and then prunes costly partial plans that are
equivalent to a cheaper one. This pruning reduces the optimization cost because
partial plans that are not likely to be optimal are pruned as early as possible
[HSH97]. In [HSH97], a new dynamic query optimization algorithm named two-
step pruning was presented to obtain the optimal global plan for a tree query. The
algorithm builds an optimal plan of a sub-query through successively generating
optimal join and transmission plans of the sub-query. When building an optimal
plan of the sub-query, it applies a pruning step twice to each sub-query by using
two equivalent criteria so that the plans except the optimal one are pruned
efficiently. Dynamic programming is traditionally used for finding the exact
optimal solution to query optimization problems but this strategy faces a
combinatorial explosion for complex queries. In order to investigate larger spaces, heuristic search strategies have been proposed in the literature to improve an initial solution until obtaining a local optimum such as simulated annealing [WI95] and iterative improvement. Tabu search is an adaptive procedure for solving combinatorial optimization problems. Considering the query optimization problem in distributed relational databases, tabu search is used to build one or more initial solutions and attempt to improve them by applying transformations (or moves) which lead to neighbor solutions of the current one, until some termination condition is attained [RRR97]. In [RRR97], using tabu search, a new attempt was proposed to solve the more difficult instances of the distributed query optimization problem, in which bushy execution plans and Cartesian products are accepted. In this paper, the authors suggest tabu search as an alternative approach for finding good approximate solutions for query optimization.

In [MB96], three new algorithms are presented. They are Algorithm W, Algorithm D1 and Algorithm D2. Algorithm W generates static strategies relying on accurate estimates to function properly. It uses the concepts of profit, marginal profit and gain to determine a sequence of semi-joins to minimize the total volume of data transferred over the network. Algorithm D1 is a simple dynamic algorithm, which selects the cheapest semi-join as the starting point for the optimization. Algorithm D2 is a more sophisticated dynamic strategy that has a "Look ahead" phase to determine the best first semi-join and to establish the order in which the join attributes will be considered. By evaluating the performance the authors found
the algorithms were better than the AHY algorithm, but their performances were not as good as they expected.

Optical communications technology allows for very high-speed data transfer. In [MBB95], Morrissey and Bandyopadhyay proposed a new algorithm to consider high-speed data transfers. Before this, typically query optimization algorithms only considered that the costs of semi-joins are determined by the amount of data to be transferred. In this paper, the cost and benefit of the semi-join also are determined by the actual communication paths. For packet switched systems, different paths may require a different number of edges and the delay, in generally, differs from edge to edge.

2.5 Filter-Based Approaches

The term “Bloom filter” is credited to Bloom [BBH70]. The concept of a Bloom filter as used in database systems was first introduced by Babb [BAE79] in their CAFS (Content Addressed File Store) device, there it was called a bit array store. The CAFS device uses a bit array store, which is accessible by the key fields in tuples from relations, for efficiently performing projection and join operations. A Bloom filter 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 semi-join but at a much lower cost, if a perfect hash function is assumed. In [WO98], Osborn presented the processing of queries using hash-based Bloom filters.
To encode attribute \( j \):

1) A bit array of some arbitrary length is allocated and initialized by setting all bits to zero.

2) For each attribute value in the relation, use a hash function to produce an address in the array.

3) For each address produced, set the corresponding bit to 1.

To reduce a relation \( R \) containing joining attribute \( j \), a filter for attribute \( j \) from another relation, denoted as \( h(j) \), is applied in the following manner:

1) For each tuple in \( R \), hash on the value for attribute \( j \).

2) For each address produced, test for the presence of a 1 bit in \( h(j) \).

3) If a 1 bit is found, the tuple is kept for further processing.

4) Otherwise, it is discarded.

From the above definition, we can see that the Bloom filter achieves the same reduction as the semi-join under the assumption of perfect hash function. However, because the filter is very much smaller, using the filter is much cheaper than using a semi-join.

Bloom filters have primarily been used to improve the efficiency of the relational join operation [JKM93, DGS88]. Bloom filters have also been applied to other relational operations [GG89, GG93] and file processing operations [DG76]. Mullin [JKM90] employs filters to improve the semi-join operation. A sequence of small Bloom filters is used to decide which semi-join operations should be
performed. This algorithm only considers two relations residing at different sites with one common joining attribute. The goal of this algorithm is to reduce communications costs to process a distributed natural join as much as possible with a filter approach. Filter information is used both to recognize when the semi-join will cease to be effective and to optimally process the semi-join. During each iteration, an optimally information dense Bloom filter is sent from the master site to the apprentice site to determine whether the semi-join will be effective. The information is used to estimate the number of tuples which need to be returned. If it is decided that the semi-join will be effective, this same information will be used in actually processing the semi-join. This processing also is used to obtain an estimate of the effectiveness of the semi-join, at a lower cost. At the end of [JKM90], Mullin points out that the multiple filter semi-join can also be valuable in reducing costs where local processing costs cannot be ignored.

In [JKM93], Mullin proposes how to estimate the size of the natural join of two relations. Four related methods are discussed in this paper. The first method uses a full filter of the joining attribute of one relation and applies it to the other relation. The second method uses a portion of the Bloom filter of one relation and applies it to the other relation. The third method uses the full filters for both relations. The last method uses partial Bloom filters to be available from both relations. Partial filters do not provide as accurate an estimate as full filters. However in many applications, high accuracy is not needed, and they have less storage demands. In this paper, a reliable estimate of the size of the final joined
table gives sufficient information to determine whether the semi-join will provide a net gain or net loss in the amount of work required to do the join.

Morrissey et al [MBB95] propose the use of filters for more accurately estimating the cost and benefit of semi-join when deriving a semijoin executing strategy. Filters are applied in various estimation techniques, such as semi-join benefit and selectivity estimation, for both a static heuristic and dynamic heuristic, which attempt to minimize total cost. It was found, however, that the use of filters did not improve the accuracy of estimation over the traditional cost and benefit estimation technique. Future work suggested by the authors includes the use of the filter as an actual reducer instead of an estimator, and the concurrent application of several filters to a relation [WO98].

Chen et al [CCY92] gave Bloom Filter a new name called the hash semi-join. It uses a bit array to represent the semi-join projection when processing a semi-join. The authors point out that given a filter of a suitable size, the hash semi-join outperforms the traditional semi-join. In this paper, the hash semi-joins are not integrated into a query processing strategy. Tseng and Chen [TC92] developed a different version of the hash semi-join. The use of the search filter in the hash semi-join achieves considerable savings in the cost of a semi-join operation. A replacement algorithm is devised in this paper [TC92]. When the replacement algorithm is used, it takes an existing semi-join strategy and replaces certain semi-joins with more cost-effective hash semi-joins. The replacement algorithm devised here, however, only works on some special cases where the execution graph is
actually a tree. Therefore, the authors advise that devising a replacement algorithm for general execution graphs be the subject of further investigation.

In [MAX97, MMA98], Morrissey and Ma propose an algorithm named Algorithm X for processing general queries. General queries involving numerous relations, attributes and sites are considered in this research. Algorithm X uses filters to reduce query response time as well as local processing costs. In this algorithm, no semi-joins are executed. It is completely based on Bloom filters. The basic idea is to apply all filters to all relations, concurrently. The rationale is that the filter sizes are small and the cost of shipping the filters to all sites is relatively low. The concurrent application of all filters decreases the response time and processing costs. Each relation is processed twice: once to construct the filters (this can be done during initial local processing) and once to reduce the relation. An evaluation of this algorithm shows that Algorithm X performs better than AHY Algorithm. But a limitation of this algorithm is that it assumes a perfect hash function.

In [MOR96, WO98], Morrissey and Osborn propose another heuristic for processing general queries by using reduction filters to accomplish the same reduction effects as semi-joins, but at a lower cost. In this algorithm, queries, to be processed by the algorithm, are represented by a query graph stored as an adjacency list. This list is used to determine which relation to process and the corresponding filters that need to be created. A queue is used to keep track of which relations need further processing. To determine the candidate relations for possible further processing, an inverted list is used to keep track of which relations contain a
specific attribute. Each query is processed in two phases: construction of reduction filters and processing of queue. A limitation of this algorithm is the assumption of the perfect hash function. In fact, the assumption of a perfect hash function is an unrealistic assumption. All hash functions suffer from collisions. A collision happens when two or more attribute values hash to the same address. This is one of the reasons why the use of filters for processing distributed queries has not been widely accepted. The previous algorithm was improved in [YL99]. Morrissey and Liang investigated how the collisions affect the performance of the algorithm. Then they designed a method to reduce the collisions by using two sets of filters. It means using two filters for each joining attribute. The result showed that the performance of the algorithm is much better than using a single set of filters under the assumption of collisions.

### 2.6 Specialized Semi-joins

There are specialized semi-joins developed to improve query optimization. Some of them are briefly introduced as follows.

#### 2.6.1 Two-way Semijoins

The two-way semijoin is an extended version of the semi-join operation [RK91, KP87]. Let us denote the two-way semijoin of $R_i$ and $R_j$ on attribute $A$ by $R_i \bowtie A R_j$ [RK91]. Then
\[ R_i \bowtie A R_j = \{ R_i \bowequiv A R_j, R_j \bowequiv A R_i \} \]

The steps in the computation of a two-way semijoin are as follows [RK91]:

Suppose that a two-way semijoin is to be computed between relation \( R_i \) at site \( i \) and relation \( R_j \) at site \( j \) on attribute \( A \), \( i \neq j \).

1) send \( R_i[A] \) from site \( i \) to \( j \),

2) reduce \( R_j \) by eliminating tuples whose attributes \( A \) are not matching any of \( R_i[A] \). This is the forward reduction of the semi-joins. During the forward reduction of \( R_j \), partition \( R_i[A] \) into \( R_i[A]m \) and \( R_i[A]nm \) where \( R_i[A]m \) is the set of values in \( R_i[A] \) which match one of \( R_j[A] \) and, \( R_i[A]nm \) is \( R_i[A]m - R_j[A]m \),

3) send either \( R_i[A]m \) or \( R_i[A]nm \), whichever is less in size, from site \( j \) back to \( i \),

4) reduce \( R_i \) using either \( R_i[A]m \) or \( R_i[A]nm \). This is the backward reduction of the two-way semijoin. If \( R_i[A]m \) is used, then tuples whose attributes \( A \) are not matching any of \( R_i[A]m \) are eliminated. If \( R_i[A]nm \) is used, then tuples whose attributes \( A \) are matching one of \( R_i[A]nm \) are eliminated.

The performance of a two-way semijoin is evaluated on the basis of the amount of reduction in the relations on which it operates. In [RK91], it has been
proved that a two-way semijoin is more powerful than ordinary semi-joins because of the backward reduction. Most query processing algorithms based on semijoins can be modified to use two-way semijoins instead. Simulation experiments indicate that the response time improves significantly [KP87].

2.6.2 Domain Specific Semi-joins

A database is a set of relations. Tuples of a relation can be horizontally partitioned into disjoint subsets called horizontal fragments, where the tuples in each subset have at least one common property. A relation \( R_i \) is fragmented by attribute \( A \) if the domain of attribute \( A \) in the \( m \)-th fragment, \( D_{om}[R_{im}.A] \), and the domain of attribute \( A \) in the \( n \)-th fragment, \( D_{om}[R_{in}.A] \), are disjoint for each pair of \( R_{im} \) and \( R_{in} \) in \( R_i \), \( m \neq n \) [CL90]. When semi-joins are employed in fragmented databases to reduce the size of the fragments of joining relations, they have to be performed in a relation-to-relation or a relation-to-fragment manner to avoid eliminating contributive tuples. In [15], Chen and Li introduce a new operation called domain-specific semijoin, a domain-specific semijoin is defined as follows:

\[
R_{ik}(A = B)R_{jm} = \{r \mid r \in R_{ik}; r.A \in R_{jm}[B] \cup (Dom[R_j.B] - Dom[R_{jm}.B])\}
\]

where \( A, B \) are the joining attributes and \( R_{ik}, R_{jm} \) are two fragments of the joining relations \( R_i \) and \( R_j \). Especially, \( Dom[R_j.B] - Dom[R_{jm}.B] \) is used to keep the contributive tuples that are not in fragment \( R_{jm} \) from lost.
When a query is executed, every semijoin is divided into several parts (domain-specific semijoin). Only the profitable domain-specific semijoins are performed. For any query processing strategy, which uses only semijoins, there is always a query-processing strategy, using both semijoins and domain-specific semijoins. Which is at least as good in a network environment in which data communication costs dominant.

2.6.3 Composite Semi-join

In [PC90], semi-join-specific techniques are classified into four categories: single attribute semi-joining, relation semi-joining, composite semi-joining and composite relation semi-joining.

The composite semi-join says whenever multiple joining attributes occur in two relations, a composite attribute semi-join can be used to replace individual single attribute semi-joins. In the combination of composite semi-joining and relation semi-joining, the entire relation is sent whenever a transmission is made, and whenever there are multiple joining attributes involved, they are joined as one composite.

It has been observed that composite semi-joining allows for low actual response times when applied to queries involving relations with multiple joining attributes [PC90].
3. THE ALGORITHM

3.1 Motivation

3.1.1 Query Graph and Full reduction

A query consists of two components: the target list and the qualification. The target list contains target attributes and target relations. The target attributes are of interest to the query. A target relation is defined as one that contains at least one target attribute. The qualification, for simplicity, is assumed as a conjunction of join clauses, which describe the query.

It is clear that if the joins of all the relations in the qualification are taken, and the resulting relation is then projected back onto the attributes of the original relations, then the projected relations will then be reduced fully. It is because that any tuple of each projected relation not satisfying the qualification would have been eliminated by the joins. The joins usually are used as an approach to check if a relation is reduced fully after reduced by a query optimizer [YC84].

The problem of optimal query processing in distributed database systems was shown to be NP-hard [CL84, YHL89]. Semijoin tactics are applied for query processing [AV85]. If relations are reduced fully using a semijoin-based algorithm before they are shipped to join site, less communication cost may be incurred when reduced relations are sent to result site. However, depending on the type of query, the relations appearing in the query may not be reduced fully. As a result, the communication cost in assembling the relations can still be high [YC84]. A precise
characterization of the type of queries whose referenced relations can be reduced fully by semijoins is therefore desirable [YC84]. The characterization is facilitated by defining a query- graph $G = (V, E)$. It is an execution graph to represent the semijoin programs associated with the distributed processing of the queries. The vertices of the query graph are the relations appearing in the qualification. An edge $(R_i, R_j)$ with label $A_k$ exists in the query- graph if $(R_i, .A_k = R_j, A_k)$ is a clause in the qualification. If $(R_i, A_i = R_j, A_i)$ also appears in the qualification, the label of the edge in the query graph is $\{A_k, A_i\}$; that is, the label is to include all attribute names that participate in the clauses involving the relations $R_i$ and $R_j$. Figure 3-1 illustrates a query-graph. The qualification is equivalent to $(R_1.A_1 = R_2.A_1)$ AND $(R_1.A_4 = R_4.A_4)$ AND $(R_1.A_2 = R_3.A_2)$ AND $(R_1.A_3 = R_3.A_3)$.

![Query Graph](image)

**Figure 3-1.** A example of query graph

If a query graph is not cyclic, we call it query tree, and the corresponding query, a tree query. A query that is not a tree query is called a cyclic query [CL84].

Usually when semijoins were used to reduce relations, each proposed algorithm was expected to get good reduction. It has been proved that relations
involved in a tree query can be reduced fully by using semijoins [BC81]. Previous research efforts also recognized the fact that cyclic query graphs can not be reduced fully by using semijoins only. General queries, which involved in our algorithm, include not only tree queries but also cyclic queries. In our thesis, we use query graph to determine the sequence of semi-joins for general query. We expect our work can find near-optimal solutions for general queries. We will provide a quantitative comparison between our near-optimal solutions and full reduction.

3.1.2 Bloom Filter and Its Collision

The use of a filter can achieve the same result as a semi-join but at a much lower cost. Although most research based on filters varies in how the filters are used, the majority encode them using hashing. Hashing is the procedure of applying a special function, called a hash function, to a key or attribute value to produce an address in a data structure. This data structure can be a hashed index or, for the purposes of attribute encoding, a bit array. The hash function applies one or more "transformations" to the value to produce the address. This ensures that a key will always hash to the same address.

In [WO98], Osborn proposed a filter-based algorithm to process general queries. The result showed that the algorithm achieves both significant reductions in relation sizes and an acceptable percentage of fully reduced queries. But this algorithm assumed a perfect hash function. It is an unrealistic assumption. Filters always suffer from collisions. A collision happens when two or more attribute
values hash to the same address. 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. This is one of the reasons why the use of filters for processing distributed queries has not been widely accepted. In [YL99], Liang extended Osborn’s algorithm. She investigated the effect of the collisions and used two sets of filters, each with a separate hash function, instead of using a single set of filters. The results showed that the performance of the algorithm is much better than using a single set of filters under the assumption of collisions. But on the other hand, it still can not avoid the problem incurred by collisions. Another disadvantage of filters is that it is never possible to do the equivalent of a composite semi-join using filters.

3.1.3 Hypothesis

Both Bloom filters and semijoins assume the independence of attributes. This assumption is one disadvantage of Bloom filters and semijoins. Sometimes two or more attributes, each with poor selectivity, can be combined to form a composite semi-join with a better selectivity. The composite semi-join says whenever multiple joining attributes occur in two relations, a composite attribute semi-join can be used instead of individual single attribute semi-joins. In figure 3-2, there are two relations R1 and R2 with two common attributes A and B. From this example, it is clear that composite semi-joins produce more reduction than separate semi-joins. So it is possible that combining composite semi-joins into a semi-join
based algorithm may obtain more efficient performance than a pure semi-join based algorithm.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

a. After two separate semijoins R2 $\bowtie$ R1 on attributes A and B

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

b. Using composite semijoin R2 $\bowtie$ R1 on attributes A and B

Do projection on attributes A and B at relation R1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Transfer the projection to R2 and join the projection with R2

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 3-2. Comparison between semijoin and composite semijoin
So far discussed above, we give the objective of this thesis here. In the thesis, we investigate a semijoin-based algorithm, which allows the combination of semijoins and composite semijoins. This algorithm will reduce the relations involved in a query significantly and it will reduce more queries fully under feasible costs.

3.2 The Algorithm

In this section, we present our proposed algorithm. This algorithm uses semijoins and composite semijoins as reducer. Its primary goal is to reduce the size of all relations, while incurring lower data transmissions. The secondary goal is to minimize the query processing cost by processing each relation the least number of times possible. The algorithm is presented in detail and illustrated with a running example below.

We assume a point-to-point network and there is no fragmentation or replication in the database. We will only consider select-project-join (SPJ) queries. Since most queries can be stated in this format, this restriction will not limit the effectiveness of the algorithm as an optimizer of general queries. We also assume that selects and projects are carried out on the relations during the local processing phase before the algorithm takes place.
3.2.1 Details of the algorithm

Queries, to be processed by the algorithm, are represented by a query graph stored as an adjacency list. This list is used to determine the sequence of relations to be processed using semijoins and composite semijoins. A queue is used to keep track of which relations need further processing. A “semijoin rule”, given below, is used to determine which relations to add to the queue. Each query is processed in two phases.

**Phase one:**

1) From the query graph, select the relation with lowest degree for processing. In other words, select the relation with the lowest number of edges incident to it. We will denote this relation as $R$.

2) Scan adjacency list to determine if projections for any joining attributes exist, and apply them to $R$ to reduce it using semijoins or composite semijoins. Then construct projections for all joining attributes contained in $R$.

3) Determine which relations to place back on the queue. The “semijoin rule” states that a relation is placed on the queue if:
   
   a. The relations for any projections of its joining attributes have been changed after being applied to $R$,
   
   b. it is not $R$,
   
   c. it is not already on the queue,
d. it has been processed already.

4) Reduce the degree of adjacent relations in the list by 1.

5) Mark relation as processed.

6) Repeat step 1 to step 5 till all relations are marked as processed.

Phase two:

1) Remove one relation from queue.

2) Reduce relation using all appropriate projections.

3) If a projection changes then use the “semijoin rule”.

4) Repeat all steps until the queue is empty.

3.2.2 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 3-3.

The SQL formulation of this query is:

```sql
SELECT * FROM R1, R2, R3, R4, R5
```
Figure 3-3 The five relations of the example

First, we will construct the query graph for this query example. The query graph is depicted in figure 3-4. Then the query graph is stored in adjacency lists, which are depicted in figure 3-5. The first node contains the relation name and the degree of this relation. The rest of the nodes contain the relevant relation name and joining attribute name.
Figure 3-4 Query graph of the query example

Figure 3-5 Adjacency List
Phase one:

1) In the adjacency lists, R5 has the lowest degree that is 1. We select R5 and construct a projection for attribute F. The adjacency lists are updated. The degree of R1 decreases by 1. The relation R5 is marked as processed.

2) We select R1 whose degree is 2 now. Because R5 has been processed, we reduce R1 using projection F and produce projections for attribute A, E and F at the same time. Projection F was changed. Put R5 into the queue. The degree of R2 decreases by 2. The relation R1 is marked as processed.

| R1 | -1 |
| R2 | 2  |
| R3 | 2  |
| R4 | 2  |
| R5 | -1 |

Projections:  
[A]: 1,2,3  
[E]: 2,4  
[F]: 4,5

Queue: R5

| R1 |  
|----|---|---|
| A | E | F |
| 1 | 2 | 4 |
| 2 | 4 | 4 |

Figure 3-6 After step 2.
3) We Select R2 whose degree is 2. There are two common join attributes A and E between R1 and R2, so do composite semijoin R2 ⧵ R1 on attributes A and E.

Build projections A, B, D and E. Projection A and E are changed, so we push related R1 into the queue. Decrease the degrees of R3 and R4 each by 1. Mark R2 as processed.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Projections:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>[A]: 2</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>[B]: 3</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>[D]: 5</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>[E]: 4</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>-1</td>
<td>[F]: 4,5</td>
<td></td>
</tr>
</tbody>
</table>

Queue: R5, R1

![Table of R2]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>A</td>
<td>B</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 3-7 After step 3.

4) We Select R3 whose degree is 1. Because R2 has been processed, we reduce R3 using projection D and produce projections for attribute D and C. Decrease the degree of R4 by 1. R3 is marked as processed.
Figure 3-8 After step 4.

5) Select R4 to process. Using R2 and R3 to reduce R4 on attributes B and C. Do projection on B and C. Projection C was changed, so push R3 into the queue. Mark R4 as processed. Because all relations are marked as processed, go to phase two to do further processing.
<table>
<thead>
<tr>
<th>R1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>-1</td>
</tr>
<tr>
<td>R3</td>
<td>-1</td>
</tr>
<tr>
<td>R4</td>
<td>-1</td>
</tr>
<tr>
<td>R5</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Projections:**

- [A]: 2
- [B]: 3
- [C]: 1
- [D]: 5
- [E]: 4
- [F]: 4,5

**Queue:** R5, R1, R3

<table>
<thead>
<tr>
<th>R4</th>
<th>B</th>
<th>C</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

**Figure 3-9 After step 5.**

**Phase two:**

6) We remove R5 from the queue and process it. We reduce R5 with projection F. Projection F was not changed. Nothing goes into the queue.
7) We remove R1 from the queue and process it. We reduce R1 with projections A, E and F. Then we do projections on A, E and F. Projection F was changed, so we push R5 into the queue.

8) We remove R3 from the queue and process it. We reduce R3 with projections D and C. Then we do projections on D and C. No projection was changed, so no related relation is pushed into the queue.
9) We remove R5 from the queue and process it. We reduce R5 with projection F. Then we do projections on F. Projection F was not changed, so no related relation is pushed into the queue. The queue becomes empty. The algorithm stops.
Result relations:

The reduced relations are given in figure 3-14. All relations are reduced fully.

\[
\begin{array}{c|c|c|c}
R1 & A & E & F \\
& 2 & 4 & 4 \\
\end{array}
\quad
\begin{array}{c|c|c|c|c}
R2 & A & B & D & E \\
& 2 & 3 & 5 & 4 \\
\end{array}
\quad
\begin{array}{c|c}
R3 & C & D \\
& 1 & 5 \\
\end{array}
\quad
\begin{array}{c|c|c}
R4 & B & C & H \\
& 3 & 1 & 8 \\
\end{array}
\quad
\begin{array}{c|c}
R5 & F & G \\
& 4 & 6 \\
\end{array}
\]

Figure 3-14 The five reduced relations after the algorithm
4. EXPERIMENTS

Besides analytical discusses of algorithm, it is very important to provide empirical comparisons as well. Such an empirical comparison can provide a way to evaluate the validity of assumptions made and the techniques used.

In this chapter, we describe the framework that we are going to evaluate our new algorithm. The aims of the evaluation are:

1) To compare the algorithm against the effects of a full reducer. The cardinalities of the fully reduced relations are used to determine the amount of full reduction. It can help determine how close the algorithm come to achieving full or optimal reduction in relations.

2) To measure the performance enhancement of the algorithm over IFS in terms of total cost. IFS is called the Initial Feasible Solution. The cost of the algorithm includes the cost for each projection shipped and the size of reduced relations.

4.1 The Queries

The test database we are using is the one developed by the Database Research Group at School of Computer Science at University of Windsor [BWT95]. It is basically a modified version of the Wisconsin benchmark database [BDT83], which proposed by Bitten, Dewitt and Trubyfill.
Based on the test database, we constructed a set of different test queries and relations. Each query consists of an arbitrary number of relations and an arbitrary number of joining attributes after all local site processing. The queries and relations vary in the following characteristics.

1) The number of relations involved in a given query,
2) The number of possible attributes involved in a given query,
3) The selectivity of the attributes,
4) The size of relations,
5) The domain size of attributes,

In our experimental system, the actual query construction is handled by the C programs “create_query.c” and “relbuilder.c”. While it is unrealistic to construct explicit queries as in [Bod85], the statistical representation on the relations and attributes is used to facilitate the construction of a wide variety of test queries. Given the desired number of relations and the maximum number of join-attributes, the program will produce a query statistics table as well as the input parameters that are required for constructing the actual relations. Then the statistical information is used subsequently to construct the relations that are described in the statistical table.

For simplicity, only integer attribute values are considered in our experimental system. The values for a particular attribute are randomly selected from the domain pool of values for the attribute. For example, suppose the cardinality of the domain for attribute A is 1000, which implies that possible values are 0 – 999. If attribute A of relation R1 is required to have a cardinality of 500 (or
equivalently having a selectivity of 0.5), 500 different values will be randomly
elected from domain, thus constituting the actual values for attribute A.

The programs that are used to generate queries are available via the World
Wide Web at the following URL:

http://www.cs.uwindsor.ca/meta-index/research/dbrg/

The programs are given in the following table.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_query.h</td>
<td>The header file for the create_query.c program.</td>
</tr>
<tr>
<td>create_query.c</td>
<td>Program for Creating the query statistics.</td>
</tr>
<tr>
<td>relbuilder.h</td>
<td>The head file for the relbuilder.c program.</td>
</tr>
<tr>
<td>relbuilder.c</td>
<td>This program uses the query statistics to construct ac</td>
</tr>
<tr>
<td></td>
<td>relations that match the statistical characteristics.</td>
</tr>
</tbody>
</table>

Figure 4-1 The query generator program

4.2 Individual Runs

The algorithm is evaluated with over 3600 queries that vary in many ways
including the number of relations and joining attributes, and the selectivity. In order
to effectively evaluated this algorithm with such a large number of diverse queries it
is necessary to split up the queries into runs.

Basically, each query in our experiments has between 3 and 6 relations,
between 2 and 4 attributes, the domain size is between 150 and 250, the
connectivity is 75% and the number of tuples in each relation is between 200 and
600. Then for the purposes of this evaluation, each combination of a relations count and attribute count make up what will be referred to as a query type. For example, query type $3 - 2$ represents three relations and two joining attributes, while query type $6 - 4$ represents six relations and four joining attributes. In total, twelve query types ranging from $3 - 2$ to $6 - 4$ will be represented in the experiments.

Furthermore, each type of query is split into three runs varying in different selectivity. The selectivity ranges from 2% to 39%, from 40% to 69% and from 70% to 95%. Totally, we will have 36 runs and each run includes 100 queries.

A C++ program, executerun.exe, is used to execute a run. The output from each run consists of a file, which includes the unoptimized relation cardinalities, the final cardinalities from the full reducer, the final cardinalities from the algorithm, the total time cost using the algorithm and the total time cost using initial feasible solution (IFS). Then the average experimental data for each run can be calculated from the file.

### 4.3 Full Reducer

To evaluate our algorithm objectively, we compare the performance of the 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.

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.

Using the previous example of Figure 3-1, we explain how the full reducer works.

First, we join the five relations. Figure 4-2 describes the result of joining of five relations.

\[ \text{R1 } \Join \text{ R2 } \Join \text{ R3 } \Join \text{ R4 } \Join \text{ R5 :} \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

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 4-3. Notice that in each relations we only have contributive tuples, all non-contributive tuples have been eliminated.
<table>
<thead>
<tr>
<th>R1</th>
<th>A</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R2</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R3</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R4</th>
<th>B</th>
<th>C</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 4-3. Results of fully reduced relations

Then our program will calculate the average percentage of reduction achieved by the algorithm and the percentage of queries that achieve full reduction in each run. The calculation of reduction of each query is based on the following formula.

Reduction (%) = \((\text{total size} - \text{reduced size}) \div (\text{total size} - \text{full size})\)*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 = 49 units

Reduced size = 14 units

Full size = 14 units

Reduction (%) = [(49 - 14) / (49-14)]*100 = 100%

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

4.4 Initial Feasible Solution (IFS) and Total Time Cost

A simple way to process a query is to perform initial local processing, directly followed by the transmissions of all remaining data to the query site, where centralized query processing builds the result. This is the so called the Initial Feasible Solution (IFS).

In our thesis, a comparison against IFS will be done in order to evaluate the algorithm in term of total cost. This can help us to determine if the algorithm reduces the total cost of a query significantly. The total of IFS is the sum of the costs of transferring all relations to the joining site.

In the previous example of Figure 3-1, we assume that the cost of transferring one value of an attribute to the final joining site costs one unit. So the total cost of IFS is:

Total Cost = 12+16+6+9+6 = 49 units
So in our program, we can obtain the total cost of IFS by add the sizes of all relations participating query.

Also in our program, in order to calculate the total cost of the algorithm, we need consider both the cost of transferring the reduced relations to the joining site and the costs are used to do semijoins and composite semijoins. The cost in one semijoin or one composite semijoin is the size of projection transferred from the source relation site to the reduced relation site. Finally, the total cost of the algorithm over IFS is calculated as following.

\[
\text{Total Cost of The Algorithm over IFS} = \\
\frac{\text{Total Cost of IFS} - \text{Total Cost of The Algorithm}}{\text{Total Cost of IFS}}
\]
5. RESULTS OF EVALUATION

In this chapter, we present the results of the performance evaluation. The observations based on the results of comparison against full reduction will be presented first, followed by the observations based on the results of comparison against the initial feasible solution (IFS). It is very important to provide such empirical comparisons. Such empirical comparisons can give a way to evaluate the validity of assumptions made and the techniques used in our thesis.

5.1. Comparison against Full Reduction

The main purpose of this comparison is to determine how well the algorithm performs, while the secondary purpose is to determine how the selectivity, the number of relations involved in queries, and the number of attributes in queries effect the performance of the algorithm. The following figure 5-1 shows the actual data. In the table, “Reduction” means the average percentage reduction produced in each type of queries by the algorithm, with respect to full reduction, and “Fully Reduced” means the percentage of queries that achieve full reduction in each type of queries.

- The actual data shows that, in all cases, the algorithm achieves substantial reductions in the sizes of the relations. On average, approximately 99.51% of all tuples not required for the final results are eliminated from the relations
<table>
<thead>
<tr>
<th>Type</th>
<th>selectivity 0.02--0.4</th>
<th>selectivity 0.4--0.7</th>
<th>selectivity 0.7--0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reduction</td>
<td>Fully Reduced</td>
<td>Reduction</td>
</tr>
<tr>
<td>3-2</td>
<td>99.09</td>
<td>89</td>
<td>99.07</td>
</tr>
<tr>
<td>3-3</td>
<td>99.52</td>
<td>94</td>
<td>98.37</td>
</tr>
<tr>
<td>3-4</td>
<td>99.99</td>
<td>97</td>
<td>99.28</td>
</tr>
<tr>
<td>4-2</td>
<td>99.63</td>
<td>87</td>
<td>99.58</td>
</tr>
<tr>
<td>4-3</td>
<td>99.82</td>
<td>94</td>
<td>98.20</td>
</tr>
<tr>
<td>4-4</td>
<td>99.98</td>
<td>96</td>
<td>99.35</td>
</tr>
<tr>
<td>5-2</td>
<td>99.96</td>
<td>98</td>
<td>99.85</td>
</tr>
<tr>
<td>5-3</td>
<td>99.98</td>
<td>99</td>
<td>99.27</td>
</tr>
<tr>
<td>5-4</td>
<td>99.56</td>
<td>97</td>
<td>99.99</td>
</tr>
<tr>
<td>6-2</td>
<td>99.72</td>
<td>98</td>
<td>99.86</td>
</tr>
<tr>
<td>6-3</td>
<td>99.27</td>
<td>97</td>
<td>100.0</td>
</tr>
<tr>
<td>6-4</td>
<td>100.0</td>
<td>100</td>
<td>100.0</td>
</tr>
<tr>
<td>Avg</td>
<td>99.71</td>
<td>96</td>
<td>99.40</td>
</tr>
</tbody>
</table>

* The connectivity is 75%.

Average Reduction = (99.71 + 99.40 + 99.41)/3 = 99.51%  
Average Fully Reduced = (96 + 94 + 95)/3 = 95%

Figure 5-1. Comparison with Full Reducer

involved in the queries. The actual data also shows that, on average, the algorithm fully reduces the relations in 95% of all queries. Therefore, we can find that the shipment of large volumes of useless data can be avoided when the algorithm is used as query reducer.

- The actual results of varying that selectivity of the joining attributes show that the algorithm almost has the same reduction on each type of queries. The difference in the best average reduction (99.71%, 0.02—0.4) and the worst average reduction (99.40%, 0.4—0.7) is kept with 0.3%. On the other hand, the
difference in the best average percentage of fully reduced queries (96%, selectivity of 0.02—0.4) and the worst (94%, selectivity of 0.4—0.7) is kept with 2%.

- Queries of types 3—2 have little lower amounts of data reduction and less fully reduced queries than queries of other types. Although in such case, we can find that 98.86% of these data that do not participate in the final result are eliminated from query relations. So the performance of the algorithm is more efficient than we expected in the least case.

- The actual results of varying the number of relations and the number of attributes involved in queries show that the difference between the best reduction and the worst reduction is not substantial. Usually, tuples, which do not participate in the final result are eliminated sufficiently. At least, we can obtain 97.12% reduction in the worst case (3—3, selectivity of 0.7—0.95). However, the difference in the best percentage of fully reduced queries and the worst percentage of fully reduced queries (selectivity of 0.7—0.95) is approximately 18%, which is substantial. The percentage of queries that achieve full reduction increases as the increase of the number of attributes involved in queries increases and the number of relations participating in the join.
5.2. Comparison against IFS

The main purpose of this comparison is to determine how well the algorithm performs in terms of total cost. The comparison is done against the initial feasible solution (IFS) under a wide variety of type queries. The queries vary on the selectivity, the number of relations involved in queries, and the number of attributes in queries. We also wish to see how these factors effect the performance of the algorithm. The following figure 5-2 shows the actual data.

<table>
<thead>
<tr>
<th>Type</th>
<th>selectivity 0.02-0.4</th>
<th>selectivity 0.4-0.7</th>
<th>selectivity 0.7-0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Our algorithm over IFS</td>
<td>Our algorithm over IFS</td>
<td>Our algorithm over IFS</td>
</tr>
<tr>
<td>3--2</td>
<td>80%</td>
<td>54%</td>
<td>27%</td>
</tr>
<tr>
<td>3--3</td>
<td>81%</td>
<td>66%</td>
<td>58%</td>
</tr>
<tr>
<td>3--4</td>
<td>76%</td>
<td>70%</td>
<td>69%</td>
</tr>
<tr>
<td>4--2</td>
<td>91%</td>
<td>65%</td>
<td>37%</td>
</tr>
<tr>
<td>4--3</td>
<td>90%</td>
<td>72%</td>
<td>61%</td>
</tr>
<tr>
<td>4--4</td>
<td>88%</td>
<td>78%</td>
<td>75%</td>
</tr>
<tr>
<td>5--2</td>
<td>94%</td>
<td>74%</td>
<td>43%</td>
</tr>
<tr>
<td>5--3</td>
<td>93%</td>
<td>80%</td>
<td>70%</td>
</tr>
<tr>
<td>5--4</td>
<td>93%</td>
<td>85%</td>
<td>80%</td>
</tr>
<tr>
<td>6--2</td>
<td>96%</td>
<td>79%</td>
<td>58%</td>
</tr>
<tr>
<td>6--3</td>
<td>94%</td>
<td>86%</td>
<td>74%</td>
</tr>
<tr>
<td>6--4</td>
<td>95%</td>
<td>88%</td>
<td>81%</td>
</tr>
<tr>
<td>Avg</td>
<td>89%</td>
<td>75%</td>
<td>61%</td>
</tr>
</tbody>
</table>

* The connectivity is 75%.

Average Cost over IFS=(89+75+61)/3=78%

Figure 5-2. Comparison with IFS
• The actual data shows that, on average, the algorithm makes the significant reduction in total cost compared to IFS. The algorithm outperforms IFS by approximately 78%.

• The actual data shows that the improvement of the algorithm over IFS will be more significant when the number of relations in the query goes up. When the number of relations is three, the average total cost over IFS is 63% (selectivity of 0.4—0.7). However, when the number of relations is six, the average total cost over IFS goes up to 84% (selectivity of 0.4—0.7).

• When the selectivity is high, the actual data shows that the improvement of the algorithm over IFS will be more significant when the number of attributes in the query increase. But when the selectivity ranges from 0.02—0.4, the number of attributes will not effect the performance of the algorithm.

• Obviously, whenever the selectivity goes up high, for example ranging from 0.7 to 0.95, the algorithm will not be as good as enough for reducing total cost. The average total cost over IFS is 61%. In the least case (type of 3—2), the algorithm only outperforms IFS 27%. However, when the selectivity ranges from 0.02 to 0.4, the algorithm will generate very attractive performances since the average total cost over IFS is 89%.
5.3. Discussion

The objective taken in the algorithm is mainly to reduce the communication cost required for data transmission. The performance evaluation shows that, on average, the algorithm gives substantial reductions of relation sizes under an acceptable total cost. When we compared the algorithm against full reduction, we found that the variation of the selectivity, the number of relations involved in a query and the number of attributes in a query do not have obvious effects on the reduction ability of the algorithm. However, these variations effect the total cost of the algorithm significantly when the selectivity is high. A further analysis of the results and the algorithm has revealed the following.

First of all, we want to discuss the use of composite semijoins again. In our algorithm, composite semijoins are used as relation reducers to reduce the amount of intermediate data to be shipped. Just as we discussed before, composite semijoins are much powerful. Composite semijoins can efficiently eliminate many unneeded tuples during the reduction phase. They can also help to reduce more queries. Hence, on each type of query, the algorithm gave good reduction. The variations of selectivities, the number of relations involved in queries, and the number of attributes in queries do not obviously effect the reduction power of the algorithm.

Another important feature of composite semi-join is that composite semijoins propagate bigger benefits than simple semijoins and Bloom filters do. It benefits the remaining relational data manipulation operations in a query. Figure 3 gives an example to illustrate how a composite semijoin propagates the benefits.
R1 | A | B | C |
---|---|---|---|
2  | 4 | 4 |
2  | 5 | 3 |
3  | 6 | 5 |
4  | 7 | 7 |

R2 | A | B |
---|---|---|
2  | 7 |
2  | 8 |
3  | 6 |
3  | 7 |
4  | 5 |

a. After two separate semijoins R2 $\bowtie$ R1 on attributes A and B

Projection on A

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Projection on B

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

b. After composite semijoin R2 $\bowtie$ R1 on attributes A and B

Projection on A

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Projection on B

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Figure 5-3. Propagation of benefits

As depicted in the figure above, if there is another semi-join R3 $\bowtie$ R2 on attribute A following composite semi-join R2 $\bowtie$ R1, then R3 $\bowtie$ R2 will only cost 1 unit to send projection A to the site R3. It is very gainful to use composite
semijoin on R2 $\bowtie$ R1 instead of simple semijoin R2 $\bowtie$ R1. So composite semi-join always propagates bigger benefits to the followed relation operations.

Meanwhile, a composite semi-join often generates bigger projection. A separate composite semijoin may require more data transmission than a single simple semi-join. So a composite semi-join itself is not more beneficial than a simple semi-join. But when it is put into a sequence of relation operations in a query, it is more gainful than a semijoin.

Back to the experiment data, when selectivity is high, total cost is not efficient. Especially when these types of queries (type of 3–2, type of 4–2), they get the worst cost of intermediate data transmission. The reason is that high selectivity causes bigger projections in a composite semi-join, more data transmissions are needed. Further more, these types of queries with less number of relations and number of attributes include less relational operation in their operation sequences. Consequently, the queries do not include sufficient relational operations benefited from previous composite semijoins. So the percentage of the total cost of the algorithm over IFS are not good enough in these types of queries.

But when the number of attributes and the number of relations increase, there are more relational operations which benefit from previous composite semijoins. So the queries using the algorithm as reducer become more efficient comparing with IFS. Especially, when selectivity is lower, the queries have attractive total costs. It is mainly because that the projections generated by composite semijoins are not big again. So the costs of composite semijoins are
reduced. Consequently, the queries cost less than these that selectivities are high and the performance of the algorithm become better. The difference of performances varying in the number of attributes and the number of relations in queries is not substantial.
6. CONCLUSIONS

In this thesis, a new semi-join based algorithm is proposed. The algorithm can process general queries consisting of an arbitrary number of relations and joining attributes. More specifically, in order to improve the reduction ability of semi-joins, we combine composite semi-joins into our algorithm. Composite semi-joins may produce more reduction than separate semi-joins in our algorithm. The primary goal of our algorithm is to reduce relation sizes while incurring minimum intermediate data transmission during query processing. A secondary goal is to incur minimum processing costs in term of total time cost. The lower the amount of data transferred, clearly the lower the total time for the query will be.

Our proposed algorithm has been evaluated to determine how close it comes to achieving full reduction of relations under various conditions. The test data used to evaluate the algorithm consists of many select-project-join (SPJ) queries, which vary in many ways. The algorithm was compared with full reducer and initial feasible solution (IFS). Using the actual data of the evaluation, we now get the following results:

- On average, approximately 99.51% of all tuples not required for the final results are eliminated from the relations involved in the queries the algorithm fully reduces the relations in 95% of all queries.

- On average, the algorithm makes the significant reduction in total cost compared to IFS. The algorithm outperforms IFS approximately 78%.
• The actual results of varying that selectivity of the joining attributes show that the algorithm almost has the same reduction on each type of queries. The difference in the best average reduction and the worst average reduction is kept in 0.3%.

Consequently, the algorithm proposed in our thesis performs well in comparison to the IFS and the full reducer program, with respect to both the average percentage reduction of query relations and the percentage of queries that achieve full reduction. And the data transmission cost is reduced significantly in comparison with IFS.
BIBLIOGRAPHY


[KP87] H. Kang and N. Poussopoulos, “Using 2-way semi-joins in distributed query processing” In Proc. 3rd Int. Conf. on Data Engineering, 1987


VITA AUCTORIS

Ma Lei was born in 1970 in Hexian County, Anhui Province, P.R.China. He obtained a B. Eng, in Computer Engineering at the Eastern China Institute of Metallurgy, P.R.China in 1991. He is currently a candidate for the Master's degree in Computer Science at the University of Windsor and will graduate in the Fall of 2000.