

University of Windsor

Scholarship at UWindor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

1998

The use of reduction filters in distributed query optimization

Wendy Kathleen Osborn
University of Windsor

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

Recommended Citation

Osborn, Wendy Kathleen, "The use of reduction filters in distributed query optimization" (1998). *Electronic Theses and Dissertations*. 4336.
<https://scholar.uwindsor.ca/etd/4336>

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

INFORMATION TO USERS

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

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

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

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

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

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

UMI[®]

NOTE TO USERS

This reproduction is the best copy available.

UMI

THE USE OF REDUCTION FILTERS IN DISTRIBUTED QUERY OPTIMIZATION

by

Wendy K. Osborn

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
1998**



National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

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

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

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

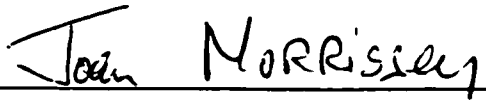
L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-52626-7

Canada

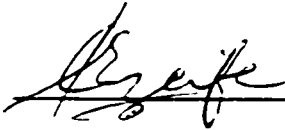
Wendy K. Osborn 1998
© All Rights Reserved

APPROVED BY:

_____

Dr. J.M. Morrissey, School of Computer Science

(advisor)

_____

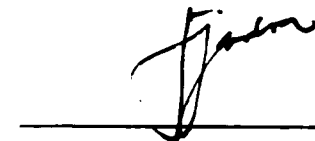
Dr. C.I. Ezeife, School of Computer Science

(internal reader)

_____

Dr. D. Kellenberger, Faculty of Education

(external reader)

_____

Dr. I.A. Tjandra, School of Computer Science

(chair)

Abstract

A major issue that affects the performance of a distributed database management system is the optimal processing of a query involving data from several sites. The problem of distributed query processing is to determine a sequence of operations, called an execution strategy, with the minimum cost. This has been shown to be an NP-Hard problem [Hen80, WC96]. Therefore, most proposed algorithms for processing distributed queries are heuristic, and focus on producing efficient (but suboptimal) strategies that minimize some particular cost of the query. Many proposed solutions use joins, semijoins, a combination of joins and semijoins, and dynamic methods. Solutions that use a filter-based approach have also been proposed. However, the limitations of such approaches include the assumption of a perfect hash function, the restriction of the algorithm to specific query types, and the restriction of the algorithm to a specific number of relations and joining attributes.

Therefore, we propose a new filter-based algorithm that can process general queries consisting of an arbitrary number of relations and joining attributes. Also, it does not assume the use of a perfect hash function. The proposed algorithm accomplishes the same reduction effects as semijoin-based algorithms, but at a lower cost. The primary goal of our algorithm is to reduce relation sizes while incurring minimum data transmission costs. The secondary goal is to incur

minimum processing costs by processing each relation as little as possible.

Our proposed algorithm is evaluated against the effects of a full reducer, to determine the following: 1) How close does the algorithm come to achieving full reduction of the query relations? and 2) How do collisions affect the performance of the algorithm? The results of the evaluation show that: 1) On average, our algorithm eliminates over 90% of the unneeded data from the query relations, 2) Our algorithm fully reduces the relations of over 80% of the queries. 3) Collisions do not substantially affect the amount of full reduction being achieved by our algorithm, and 4) A low percentage of collisions does not substantially affect the percentage of fully reduced queries.

To the women who inspire me

Acknowledgments

I wish to thank the following people, both for their valuable assistance and for helping me keep my sanity. First of all, I give thanks to a great supervisor, Dr. Joan Morrissey. Her comments, encouragement, and patience were invaluable to the completion of this thesis. I thank Dr. Christie Ezeife and Dr. David Kellenberger for their comments on my thesis. I also wish to thank my committee for being accommodating when I needed it most. I wish to thank Zongli Jiang for running many of the evaluation runs. I also wish to thank my family for support. Finally, I wish to thank Hong for the margaritas and the Sleemans Honey Brown, Sandy for the super nachos, Phil for the home brew (hey..what do you mean you're too busy to make more??), Steph for her wonderful hospitality, Lenni for the free, much needed 'beverages' at her wedding, and Terry for putting up with me... :). Thank you all very much.

Contents

Abstract	iv
Acknowledgments	vii
List of Figures	x
List of Tables	xii
1. INTRODUCTION	1
Outline of Thesis	3
2. BACKGROUND	5
Cost Models	5
Join-Based Approaches	6
Semijoin-Based Approaches	9
Filter-Based Approaches	14
Dynamic Approaches	21
Combination Approaches	23
Conclusions	25
3. THE ALGORITHM	26
Details of the Algorithm	26
Construction of Reduction Filters	27
Processing of Queue	33
Conclusion	35

4.	EVALUATION	36
	Experimental Rationale	36
	Evaluation Framework	38
	The Experiments	43
	Conclusion	46
5.	RESULTS	47
	Results of Initial Runs	47
	Results of Collision Runs	50
	Discussion	55
	Conclusion	58
6.	CONCLUSIONS	59
	Future Research Directions	61
	SELECTED BIBLIOGRAPHY	62
	VITA AUCTORIS	70

List of Figures

Figure 1	A Distributed Database Management System	1
Figure 2	The Join $R_1 \bowtie R_2$ over Attribute B	7
Figure 3	The Semijoin $R_1 \ltimes R_2$ over Attribute b.	9
Figure 4	The Reduction of R_i by the Reduction Filter for Attribute b	15
Figure 5	Example Database	27
Figure 6	Query Graph for Example Database	29
Figure 7	Query Graph After Removing R_1 and A	30
Figure 8	The Reduced Relation R_2	30
Figure 9	Query Graph After Removing R_2 , B and E	31
Figure 10	The Reduced Relation R_5	31
Figure 11	Query Graph After Removing R_5 and F	32
Figure 12	Query Graph After Removing R_3 , C and D	32
Figure 13	The Reduced Relation R_4	32
Figure 14	The Reduced Relation R_1	34
Figure 15	The Reduced Relation R_3	34
Figure 16	The Set of Reduced Relations	35
Figure 17	The Join of the Reduced Relations	35
Figure 18	Example Files for a Query	40

Figure 19	Example Relations	56
-----------	-----------------------------	----

List of Tables

Table 1	Initial Runs	44
Table 2	Results of the Initial Runs	48
Table 3	Average Percentage Reduction at 0%-10% Collisions	51
Table 4	Average Percentage of Fully Reduced Queries at 0%-10% Collisions	52
Table 5	Average Percentage Reduction at 10%-60% Collisions	53
Table 6	Average Percentage of Fully Reduced Queries at 10%-60% Collisions	54

Chapter 1 INTRODUCTION

A distributed database management system (DDBMS) [CP84, OV91, YC84] consists of several autonomous sites that are remotely or locally distributed and connected by a network. A DDBMS has several advantages over a centralized DBMS, including: 1) increased accessibility to remote data, 2) reliability, since the local operation of one site does not affect the operation of other sites, and 3) efficiency, since each site contains its own data and can process its own transactions and queries. However, a major issue that affects the performance of a DDBMS is the optimal processing of a query involving data from several sites. The problem of distributed query processing is to determine a sequence of operations, called an execution strategy, with the minimum cost. This has been shown to be an NP-Hard problem [Hen80, WC96]. Therefore, most proposed algorithms for processing distributed queries are heuristic, and focus on

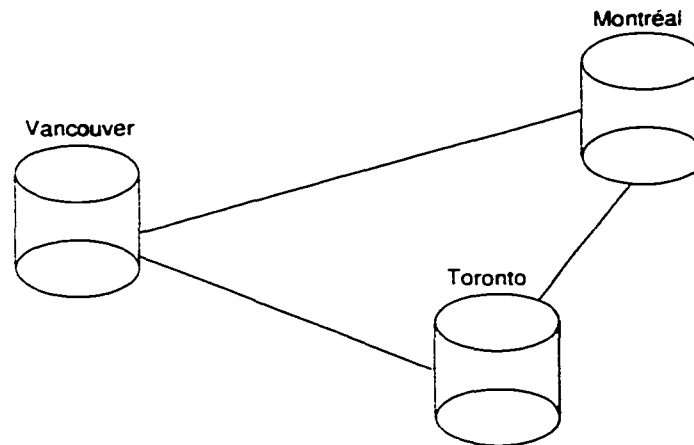


Figure 1 A Distributed Database Management System

producing efficient but suboptimal strategies that minimize some particular cost of the query. Many approaches use joins [LMH⁺85, LPP91, CY90b], semijoins [BGW⁺81, AHY83, CL84, MBB95b, KR87, WCS92, WLC91, CL90, PC90, Bea95, YL89, RK91, MB97], a combination of joins and semijoins [CY93, CY92, CY91, CY90a, CY94], and dynamic methods [YLG⁺86, BRP92, BR88, BRJ89, MBBK95, MBB95a].

Recently, some approaches that use filters [Blo70, Mul83] have been proposed in the literature [MO97, Osb96, CCY92, Mul90, Mul93, Mor96, TC92, Ma97, MO98, MM98, VG84, Bra84, MBBK95]. However, the limitations of such algorithms include the assumption of a perfect hash function (in other words, the assumption of no collisions), the restriction of the algorithm to a specific query type such as tree queries, and the restriction of the algorithm to a specific number of relations and joining attributes.

The main contribution of this thesis is a new algorithm, which uses filters to accomplish the same reduction effects as semijoins, but at a lower cost. The primary goal of our algorithm is to reduce relation sizes while incurring minimum data transmission costs. The secondary goal is to incur minimum processing costs by processing each relation as little as possible. This algorithm can process general queries consisting of an arbitrary number of relations and joining attributes, but does not assume the use of a perfect hash function.

Our proposed algorithm is evaluated, not against other algorithms, but instead against the effects of a full reducer. Our algorithm is evaluated to determine how close it comes to achieving full reduction of query 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. Using the results of the evaluation, we answer the following questions:

- On average, how much reduction, with respect to the full reducer, is achieved?
- Full reduction of query relations is achieved in what percentage of queries?
- What effect do collisions have on the amount of reduction?
- What effect do collisions have on the percentage of queries achieving full reduction?

1.1 Outline of Thesis

The remainder of this thesis is structured as follows. In chapter 2, other proposed approaches to distributed query processing will be summarized. The concepts related to these approaches will also be presented. In chapter 3, our proposed algorithm is presented in detail. An illustrated example of how the algorithm works is also presented. In chapter 4, the evaluation framework is discussed in detail. An outline of the experiments and the rationale for the evaluation will also be presented. In chapter 5, the results of the evaluation

are presented and discussed. Finally, in chapter 6, conclusions are made and some future research directions are given.

Chapter 2 BACKGROUND

The goal of processing a distributed query involving relations from several sites, is to derive an execution strategy that incurs the minimum cost. It has been shown that determining an optimal strategy is NP-Hard [Hen80, WC96]. Therefore, research focuses on developing algorithms that generate efficient, near-optimal strategies.

Several approaches proposed in the literature use relational operators such as joins [LMH⁺85, LPP91, CY90b], semijoins [BGW⁺81, AHY83, CL84, MBB95b, KR87, WCS92, WLC91, CL90, PC90, Bea95, YL89, RK91, MB97], and a combination of joins and semijoins [CY93, CY92, CY91, CY90a, CY94]. Other approaches include the use of dynamic methods [YLG⁺86, BRP92, BR88, BRJ89, MBBK95, MBB95a], the improvement of suboptimal execution strategies [CL84], and, more recently, the use of filters [Blo70, Mul83] for further minimizing cost [MO97, Osb96, CCY92, Mul90, Mul93, Mor96, TC92, Ma97, MO98, MM98, VG84, Bra84, MBBK95]. In this chapter, some concepts related to the above approaches will be presented. Then, several of the above approaches will be presented.

2.1 Cost Models

The goal of processing a distributed query is to derive a sequence of relational

operations, or an execution strategy, that 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.

2.2 Join-Based Approaches

One of the most basic relational operations used in distributed query processing is the join. Given two relations, R_1 and R_2 , both containing joining attribute B , the join of R_1 and R_2 is performed by concatenating tuples of R_1 and R_2 where the value of attribute B is equal for both tuples.

Although the join has the advantage of simplicity, it suffers from several problems. One is that the result relation can be much larger than the relations participating in the join. This increases the data transmission cost. It is computationally expensive. There is also the fact that we have to ship a large (and possibly unneeded) amount of data to the join site before performing the join.

R1		R2	
A	B	B	C
a1	b1	b1	c2
a2	b1	b3	c3
a3	b2	b4	c4
a4	b3	b4	c5

R1 ↔ R2		
A	B	C
a1	b1	c2
a2	b1	c2
a4	b3	c3

Figure 2 The Join $R_1 \bowtie R_2$ over Attribute B

Therefore, the use of the join for distributed query processing has not been shown to be a popular approach in the literature. However, for the sake of completeness, we will summarize some research that uses a join-based approach.

The R^* optimizer [LMH⁺85] aims to minimize the total cost of a distributed join query. The total cost model includes weighted measures of both the data transmission costs and the local processing costs. The R^* optimizer uses a dynamic programming approach to generate new join sequences of n relations from join sequences of $n-1$ relations. It considers several factors, including the relation access method, the join method, and the join site, when determining the optimal join sequence. Pruning of suboptimal sequences at each step attempts to minimize the full enumeration of strategies. Although this algorithm produces

optimal strategies, it has exponential complexity, and therefore is not a feasible approach.

The algorithm of Legato *et al.* [LPP91] also uses a dynamic programming approach and a dominance property, defined by the authors, to generate new join sequences from join subsequences. When generating a join sequence, the algorithm produces a binary join tree which is augmented with information such as the location of the relations and intermediate results, the join locations and the query site. This information is determined in such a way that both the local processing cost and data transmission cost of a query is minimized. As with the R* algorithm, the complexity of this algorithm is exponential.

Chen and Yu [CY90b] focus solely on minimizing the cost of data transmissions for a distributed join query. Given a query graph, the authors define the concept of a complete and feasible set of cuts to the graph, and prove that the set of cuts can be mapped to a join sequence. The cost of a set of cuts is equivalent to the sum of the sizes of the resulting joins in each cut, and is determined so that the sizes (or alternatively, the data transmission costs) are minimized. This algorithm has the advantage of polynomial time complexity – making it superior to the previous works. However, since the algorithm only applies to tree queries, further research is needed to apply this algorithm to general queries.

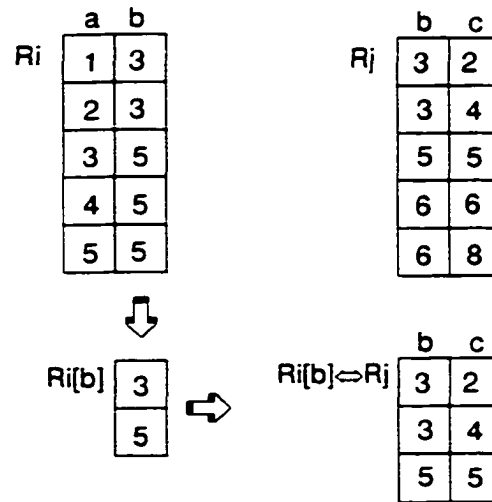


Figure 3 The Semijoin $R_1 \bowtie R_2$ over Attribute b .

2.3 Semijoin-Based Approaches

A more popular approach for processing a distributed query is to use the semijoin. Given two relations R_i and R_j , and a common join attribute b , the semijoin $R_1 \bowtie R_2$ over b is executed as follows:

1. Project R_i over b to get $R_i[b]$
2. Send $R_i[b]$ over to the site of R_j
3. Perform $R_i[b] \bowtie R_j$

The purpose of the semijoin $R_1 \bowtie R_2$ is to reduce the relation R_2 before any joining takes place by removing tuples which will not be part of the final result. The semijoin has the advantage that the data transmission cost is reduced since only an attribute projection, not the entire relation, needs to be shipped to the query site. It also has the advantage of never producing a larger relation than those

participating in the semijoin. Unfortunately, the semijoin has two disadvantages. Semijoins incur higher local processing costs since a project, as well as a small join, must be executed. Also, in many cases, they incur a higher data transmission cost, with respect to the size of the attribute projection, than necessary. However, the semijoin is still considered by many researchers to be an attractive approach for processing distributed queries. In the remainder of this section, I will summarize several approaches that use semijoins.

One of the first distributed query processing algorithms to be developed, the SDD-1 optimizer [BGW⁺81], aims to minimize the data transmission cost of a distributed query. SDD-1 transforms a query into a set of relational calculus expressions, which specifies a superset of the data needed for the query. Using this set of expressions, a greedy algorithm is used to derive the sequence of semijoins that will retrieve the set of data needed for the query. A major limitation of this algorithm is that it may produce suboptimal strategies due to the failure to consider other semijoins at each step of the strategy generation. Some future research directions of this work include removing unprofitable semijoins, reordering semijoins, incorporating joins, creating a dynamic reducer, and considering other cost factors.

Apers, Henver and Yao [AHY83] propose a collection of algorithms which process general queries – queries involving an arbitrary number of relations and joining attributes – and attempt to minimize either the response time or the data

transmission cost of a query. All three algorithms in Algorithm GENERAL follow the following basic framework. First, the general query is decomposed into simple queries by isolating each joining attribute. Second, a minimum cost schedule is derived for each simple query. Then, either the response time or the data transmission cost, is minimized depending on the algorithm used. Finally, the minimum cost schedules are integrated into an overall strategy for the general query. Although all versions of Algorithm GENERAL have polynomial time complexity, the lack of consideration of global conditions and many simplifying assumptions concerning the network may result in suboptimal strategies being generated. Therefore, research directions from this work include consideration of global conditions and network factors such as queueing delays and line contention when processing a query.

Morrissey *et al.* [MBB95b, Bea95, MB97] propose a semijoin algorithm that takes global conditions into account when estimating the cost effectiveness of a semijoin. The goal of Algorithm W is to minimize the data transmission cost of a query. For each common join attribute, a reducer is created from cost effective semijoins. In addition to using cost and benefit, the authors use two additional concepts, marginal profit and gain, which consider global factors in determining the cost effectiveness of a semijoin. After creating the schedule for the construction and application of the reducers, the schedule is executed by constructing the reducers in parallel, applying the reducers in parallel, and shipping

the reduced relations to the query site in parallel. Empirical testing shows that W consistently performs better than the AHY Algorithm GENERAL (total cost) [AHY83]. Some proposed research directions include the use of filters and the proposal of a new dynamic heuristic which constructs multi-attribute reducers.

Yoo and Lafortune [YL89] propose the use of the A* heuristic search for determining a semijoin execution strategy. The goal of the A* algorithm is to find the best path from the initial state (unreduced relations) to the final state (fully reduced relations). The authors define two concepts, admissibility and consistency, which the heuristic function must satisfy to ensure that an optimal solution is found efficiently. Also, during the search, a pruning strategy eliminates states that will lead to a non optimal solution. This algorithm has the advantage of being not only cost-model independent, but also very efficient at determining optimal solutions. However, since this algorithm only applies to tree queries, a necessary research direction is to generate a similar algorithm for handling general queries.

Chen and Li [CL84] propose an approach for taking existing semijoin execution strategies and testing them for optimality. The authors have identified several properties to which an optimal execution strategy must adhere. Given these properties, several improvement algorithms are proposed that test an execution strategy for optimality based on the identified properties and improve it if necessary. Since this method only applies to tree queries, further work is needed to extend this approach to handle general queries.

Wang *et al.* [WCS92, WLC91] present a framework for processing general queries called the one-shot semijoin execution. This framework has three phases. The first involves the projection of all required semijoin projections in parallel. The second involves the parallel transmission of the semijoin projections. The third involves the execution of the semijoins in parallel. This framework is used to develop algorithms – two which minimize the data transmission cost [WLC91] and one which minimizes the response time [WCS92]. It has the advantages of increased parallelism, reduced processing overhead, the opportunity to apply global optimization techniques, and no propagation of errors which is inherent in a sequential processing strategy.

Roussopoulos and Kang [RK91, KR87] propose the two-way semijoin. It extends the traditional semijoin to include backward reduction, which results in the reduction of both relations. A comparison of the two-way semijoin and the traditional semijoin shows that the two-way semijoin achieves more reduction and greater reduction propagation effects. The authors also propose an algorithm that attempts to minimize the local processing cost of a query. This algorithm uses the 2-way semijoin framework and pipelining to eliminate the process of creating, storing and transmitting intermediate results. This gives good I/O savings.

Chen and Li [CL90] propose a relational operator called the domain-specific semijoin, for performing semijoins between fragments in a fragmented database. The domain-specific semijoin uses the domain knowledge of a joining attribute

to ensure that no data that is required for the final result is lost as a result of performing semijoins between fragments. The authors also propose a simple algorithm which integrates domain-specific semijoins into an existing semijoin strategy and show that a strategy incorporating both operators has a lower total cost than a strategy only containing semijoins.

2.4 Filter-Based Approaches

Filters can be used to achieve the same benefits of a semijoin but with lower data transmission and local processing costs. A filter is a bit array which is a compact representation of the values in an attribute. 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 hash table) 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. Therefore, to encode attribute b:

1. A bit array of some arbitrary length is allocated and initialized by setting all bits to zero.
2. For each attribute value in b, use a hash function to produce an address in the array.

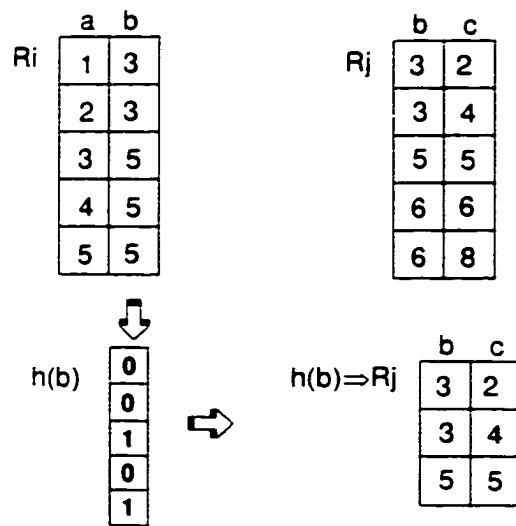


Figure 4 The Reduction of R_i by the Reduction Filter for Attribute b

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

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

1. For each tuple in R , hash on the value for attribute b .
2. For each address produced, test for the presence of a 1 bit in $h(b)$.
3. If a 1 bit is found, the tuple is kept for further processing.
4. Otherwise, it is discarded.

A filter has the following advantages: lower data transmission costs since the filter is small, and lower local processing costs since the filter is created during the processing of a relation or intermediate result. However, because hashing is utilized, a filter suffers from the problem of collisions. A collision is the event of two or more attribute values hashing to the same address. This may result

in data that is not required for the final result being shipped to the query site. For the remainder of this section, research that utilizes some type of filter-based approach will be summarized.

Bratbergsengen [Bra84] presents an joining algorithm that uses hash filters. The purpose of the filters is to reduce both relations before performing a join. During each iteration, the algorithm creates a filter for the joining attribute A of R_1 , reduces R_2 with the filter, creates a filter for the reduced joining attribute A of R_2 and applies it to R_1 to reduce it. This method significantly improves the response time of a join of medium sized relations, with a decrease in improvement for large relation joins. However, the approach needs to be extended to handle the join of multiple relations.

Valduriez and Gardarin [VG84] propose two “divide and conquer” algorithms – one for joins, one for semijoins – that utilize bit arrays. In the joining algorithm, the smaller relation is used to produce a hash table of tuples and a bit array to represent the joining attribute. The larger relation is divided and allocated among all processors. At each site, each tuple is hashed and tested against the bit array. For each tuple passing the filter, its corresponding tuples in the hash table are retrieved, tested and joined with the tuple. In the semijoin algorithm, both a bit array and a local data structure of attribute values are created from a semijoin projection. The relation to be reduced is divided and allocated to all processors. At each processor, the tuples are hashed and tested in the bit array. For all

tuples that pass the filter, they are tested in the local data structure. It is shown that, under certain conditions, the join and semijoin algorithms using bit arrays outperform other join and semijoin operators. It was also found that performing semijoins on two relations before joining them decreases the response time of a join. As with the above, more work must be done to extend the join algorithm to handle multi-relation joins. Also, more work must be done to integrate this approach into a query processing strategy.

Mullin [Mul93] presents four techniques for estimating the size of a relation resulting from a join. The purpose is to determine if the amount of reduction obtained from performing the join justifies applying semijoins before shipping the reduced relations to the joining site. These techniques utilize full or partial filters in various ways. The first technique uses a full filter of the joining attribute of one relation and applies it to the other relation. The fraction of rejected tuples and the fraction of set bits in the filter are used to estimate the join cardinality. The second technique uses a partial filter for the same purpose. The third and fourth techniques use full filters (partial filters, respectively) from both relations. The fraction of bits common to both filters is determined and used in estimating the cardinality of the resulting relation. Mullin found that these techniques to produce very accurate estimations. However, these estimation techniques only apply to joins of two relations. Further research is needed to apply these techniques to joins containing an arbitrary number of relations.

Mullin [Mul90] also proposes a filter based semijoin algorithm that serves two purposes: to calculate the cost effectiveness of a semijoin and to perform the semijoin if it is found to be cost-effective. The goal of this algorithm is to reduce the data transmission cost – at the possible expense of extra local processing. During each iteration, the algorithm creates a filter for the joining attribute of one relation, ships it to the site of the second relation, and applies it to the relation. This continues until the cost of shipping the filters outweighs the amount of reduction achieved by using them. This algorithm is successful at reducing the data transmission costs of a semijoin. However, the main limitation of this work is that the algorithm only considers two relations residing at different sites with one common joining attribute. Therefore, research is needed to extend this technique to several relations with many joining attributes.

Chen *et al.* [CCY92] propose a new relational operator called the hash semijoin. The goals of the hash semijoin are to reduce the data transmission and high local processing costs of a semijoin. The hash semijoin utilizes a bit array to represent the semijoin projection when processing a semijoin. The authors show that, given a filter of a 'suitable' size, the hash semijoin outperforms the traditional semijoin. Further work is needed to integrate the hash semijoin in a query processing algorithm.

Tseng and Chen [TC92] propose a different version of the hash semijoin, as well as a replacement algorithm for its application. The authors focus on

minimizing the semijoin cost – at the possible expense of decreased reduction effects. When constructing a filter, each attribute value is encoded with d bits by using d independent hash functions. The replacement algorithm takes an existing semijoin strategy and replaces certain semijoins with more cost-effective hash semijoins. One limitation of the replacement algorithm is that it only applies to tree queries. Another is that, along with many other proposed algorithms, the performance has not been extensively evaluated. Therefore, a future direction of research proposed by Tseng and Chen is to adapt the replacement algorithm to general queries. Also, a performance evaluation is necessary.

Morrissey *et al.* [MBBK95] propose the use of filters for more accurately estimating the cost and benefit of a semijoin when deriving a semijoin execution strategy. Filters are applied in various estimation techniques, such as semijoin benefit and selectivity estimation, for both a static heuristic and a dynamic heuristic that attempt to minimize total cost. It was found, however, that the use of filters did not improve the accuracy of estimations over the traditional cost and benefit estimation technique. Future work suggested by the authors include the use of the filter as an actual reducer instead of an estimator, and the concurrent application of several filters to a relation.

Morrissey and Osborn [Mor96, Osb96, MO97] propose a heuristic for processing general queries. The heuristic utilizes reduction filters to further reduce data transmission costs, but at the same time provide the same reduction benefits

as the semijoin. After initially constructing reduction filters for each common joining attribute and determining the processing order of the relations, the relations are reduced in three phases. The first phase applies all relevant filters to the relations in the given order. The second and third phases apply only 'changed' filters to the relations. The second phase does this in the reverse processing order while the third phase does so in the original order. They found a significant improvement in performance of the filter-based algorithm over a traditional semijoin-based algorithm [MBB95b]. Two limitations of this work, however, are the assumption of the perfect hash function and the unnecessary shipment and application of some filters, which in turn results in a higher data transmission cost than necessary. Therefore, future work includes determining ways to eliminate unnecessary data transmissions and local processing, and investigating the effect of collisions on the performance of the algorithm.

Morrissey and Ma [Ma97, MM98] propose a heuristic for processing general queries. Algorithm X uses filters to reduce query response time as well as local processing costs. This simple algorithm involves the parallel transmission and application of all relevant filters to all relations. An evaluation of their algorithm against the AHY Algorithm GENERAL (Response Time) [AHY83] shows that Algorithm X performs significantly better. A limitation of this work is the assumption of the perfect hash function. Therefore, a necessary direction of research is to investigate the effect that collisions will have on the performance

of the algorithm.

2.5 Dynamic Approaches

Many static strategies have been found to be non-optimal due to errors in estimations and the assumption of a uniform distribution of values in a joining attribute. One proposed solution to this problem is to use a dynamic approach to query processing.

Bodorik *et al.* [BRJ89] propose a three-phased framework for the dynamic execution of a strategy. During the **Monitoring** phase, information on the progress of a strategy, usually with respect to intermediate results sizes, is gathered. During the **Decision to Correct** phase, a decision can be made to alter the current strategy based on the suboptimal results produced by the strategy. In the **Corrective Action** phase, a new execution strategy is generated for the remaining query. An investigation of this framework shows that dynamic query processing has high overheads, mainly due to the delay when correcting a strategy. Two proposed research directions, the *a priori* generation of a strategy in the background during query execution and sampling of a partial intermediate result to estimate the join size, are investigated in [BRP92].

In [BRP92], Bodorik et al propose a dynamic processing method that attempts to minimize query response time, including the delay from correcting a strategy. The basic idea of the Aborted Join Last (AJL) method is to postpone expensive

joins until the end of the strategy. While the query is executing, an alternative strategy is generated in the background. Sampling of partial intermediate results is performed to determine if the delay from the current strategy is greater than the estimated delay of the alternative strategy. A performance evaluation of the AJL method shows that it produces minimal delays. Since this method only applies to tree queries, further work is needed to apply the AJL method to general queries. Another limitation is the assumption of only one estimation error occurring at a time. Bodorik *et al.* plan to investigate the effect of simultaneous estimation errors on the response time of a query.

Yu *et al.* [YLG⁺86] propose three categories of techniques for adaptive query processing. The first, direct improvement of query execution efficiency, contains techniques that directly manipulate an execution strategy by removing redundant attributes and relational operations. The second, indirect improvement of query execution efficiency, contains techniques for manipulating cost estimation formulas for join and semijoin result sizes, data transmission costs and local processing costs. The third, knowledge acquisition, involves obtaining information about the decisions of a user interacting with the execution of a query if the user can derive a better execution strategy than the system. Necessary future work includes the integration of the proposed techniques into one system, and the extension of these techniques to handle fragmented and replicated relations.

Morrissey *et al.* [Bea95, MBB95a] propose two dynamic algorithms, Dyna-

1 and Dyna-2, which use semijoins to minimize the data transmission cost of a query. In both algorithms, each reducer is created and applied one at a time. Global estimation techniques and up-to-date information on relation sizes and attribute selectivity, are used to determine cost effective semijoins for the reducers. The main difference between the two algorithms is that, during each iteration, Dyna-1 uses the smallest attribute to determine the next reducer to be created, while Dyna-2 estimates all reducer sizes and selects the smallest as the next reducer. Another difference is that Dyna-1 contains no monitoring of the reducer creation, while Dyna-2 monitors each step of this process. A performance evaluation shows that Dyna-2 outperforms Dyna-1 but only marginally performs better than the static semijoin algorithm W [Bea95, MBB95b]. In conclusion, the authors state that information about attribute values and their distribution is needed for a dynamic algorithm to produce schedules that are superior to those of a static algorithm.

2.6 Combination Approaches

Chen and Yu [CY93, CY92, CY91, CY90a, CY94] propose the combination of joins and semijoins in an execution strategy to further reduce the data transmission cost of a query. The authors define two concepts which result from this combination. Gainful semijoins are semijoins that are not profitable individually, but are found to be profitable if they further reduce the cost of a subsequent join.

Pure join attributes are attributes that are required for the processing of a query but not required for the final result, and can be removed when no longer needed by the query. Chen and Yu apply these two concepts in several algorithms [CY93, CY92, CY91, CY90a, CY94]. Two algorithms of interest [CY93, CY92] will be summarized below.

In [CY92], Chen and Yu propose an algorithm for interleaving a join sequence with semijoins. Given a join sequence, both profitable and gainful semijoins are identified based on certain properties of the relations in the join sequence. Then, these semijoins are inserted into the existing join sequence. An illustrative example from their paper shows the benefits of combining joins and semijoins to reduce data transmission costs.

In [CY93], Chen and Yu propose the use of the A^* heuristic search to determine a sequence of joins and semijoins. The heuristic function, $f(x) = g(x) + h(x)$, is derived to calculate the cost of the join/semijoin sequence up to state x ($g(x)$) plus the estimated cost of the join/semijoin sequence from x to the final set of reduced relations ($h(x)$). The data transmission cost calculated by $h(x)$ is estimated by summing the sizes of the intermediate results at state x . The authors also propose rules for selective expansion, which prohibit the expansion of unoptimal states. Limitations of the A^* approach are the assumption of tree queries and the exponential complexity. Future work must be done to apply the A^* technique to general queries.

2.7 Conclusions

The goal of processing a distributed query is to derive an execution strategy that incurs the minimum cost. In this chapter, several approaches have been presented, that utilize joins [LMH⁺85, LPP91, CY90b], semijoins [BGW⁺81, AHY83, CL84, MBB95b, KR87, WCS92, WLC91, CL90, PC90, Bea95, YL89, RK91, MB97], filters [MO97, Osb96, CCY92, Mul90, Mul93, Mor96, TC92, Ma97, MO98, MM98, VG84, Bra84, MBBK95], a combination of joins and semijoins [CY93, CY92, CY91, CY90a, CY94] and dynamic methods [YLG⁺86, BRP92, BR88, BRJ89, MBBK95, MBB95a]. Given the research directions proposed from these approaches, it is apparent that much work needs to be done. In the next chapter, we propose our solution to the distributed query processing problem. We present a new algorithm with an illustrative example.

Chapter 3 THE ALGORITHM

In this chapter, we present our proposed algorithm. This algorithm uses reduction filters to accomplish the same reduction effects as semijoins, but at a lower cost. The primary goal is to reduce the size of all relations, while incurring minimum 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. We also assume that the distributed database management system contains relational data that is neither fragmented nor replicated. 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. The cost function to be used, an inverse cost function, calculates the total reduction achieved by the algorithm, instead of the cost incurred.

3.1 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 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

R1

A	H
1	8
1	7
2	6
3	8

R2

A	B	E
1	2	4
4	5	2
5	4	5

R3

C	D	I
3	4	9
6	9	1
8	6	6
7	7	7

R4

B	C	D	F
2	3	4	6
6	9	5	9
6	3	3	9
7	2	2	9

R5

E	F
4	6
5	8
6	8

Figure 5 Example Database

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: (1) Construction of Reduction Filters and (2) Processing of Queue, which are described in detail below.

Phase 1: Construction of Reduction Filters

During this phase, the query graph, which is represented by an adjacency list, is constructed. It is used to determine the order in which filters are constructed and used. For each relation in the query graph, a reduction filter is created for each joining attribute contained in the relation. During the processing of a relation, any existing filters are applied to the relation to reduce it. Certain relations are added to the queue if they require further processing. A 'filter rule', given below,

is used to determine which relations to add to the queue. The iterative process of constructing the reduction filters is described below:

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

In the following example, the reduction filters will be denoted by the notation $X:n1,n2,...nm$. where X is the attribute label and $n1, n2,...nm$ are the addresses

of the bits in the filter that are currently set to 1. All other bits not listed are assumed to be zero. Using the example database given in Figure 5, the following query graph is constructed.

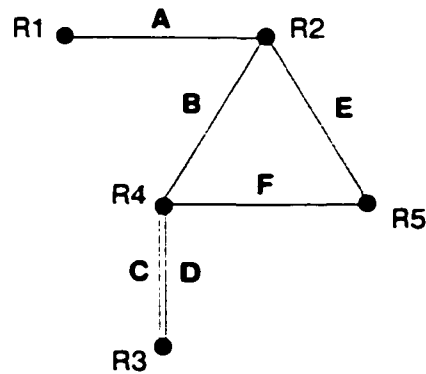


Figure 6 Query Graph for Example Database

The first relation to be processed will be R_1 since it has the lowest indegree. R_1 contains the joining attribute A , but since its reduction filter does not exist already, it is constructed by making a pass through the relation. The resulting filter is $A:1,2,3$. Since this new filter has not changed, no relations are placed on the queue. The vertex for R_1 and the edge for attribute A are removed from the query graph. We then have the following query graph.

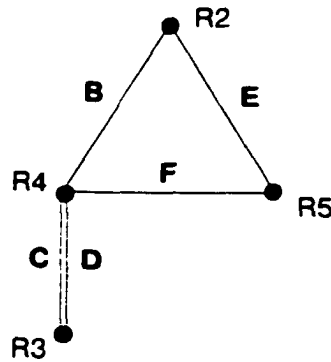


Figure 7 Query Graph After Removing R_1 and A

There are three relations with the lowest indegree of 2. Since two or more relations have the same indegree, the decision as to which of these three relations to reduce next is arbitrary, and depends on the order of the relations in the adjacency list. We will process R_2 next. This relation contains the joining attributes A, B, and E. The reduction filter for attribute A already exists and is applied to R_2 . During this process of reduction, filters for A, B, and E are constructed. The resulting filters are A:1, B:2 and E:4. We also have the following reduced relation R_2 .

A	B	E
1	2	4

Figure 8 The Reduced Relation R_2

Since the reduction filter for A has changed, the relations that contain A, R_1 and R_2 , are tested to see whether they should be placed on the queue. R_1 is not already on the queue, not currently being processed and has been processed

previously, and therefore it is placed on the queue. Since R_2 is the relation currently being processed, it is not placed on the queue. The relation R_2 and the edges for attributes B and E are removed from the graph, which results in the following query graph.

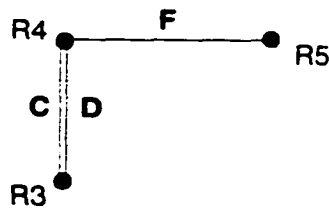


Figure 9 Query Graph After Removing R_2 , B and E

Relation R_5 has the lowest indegree and is thus chosen as the next relation to be processed. The filter for attribute E exists and is applied to R_5 . The filters for attributes E and F are created. The resulting filters are E:4 and F:6. The resulting relation R_5 is shown below.

E	F
4	6

Figure 10 The Reduced Relation R_5

Since the filter for attribute E has not changed as a result of processing R_5 , no relations are added to the queue. After removing R_5 and the edge for attribute F from the query graph, we have the following graph.



Figure 11 Query Graph After Removing R₅ and F

Both remaining relations, R₃ and R₄, both have an indegree of 2. Since both relations have the same indegree, the decision as to which relation to reduce next is arbitrary. We will process R₃ first. Neither of the joining attributes C and D have existing filters. Therefore, a scan is made of the relation to construct their filters. The resulting filters are C:3,6,7,8 and D:4,6,7,9. Since we have no changed filters as a result of processing this relation, no relations are added to the queue. After removing R₃ and the edges for C and D from the graph, we are left with the sole relation R₄.



Figure 12 Query Graph After Removing R₃, C and D

At this point, the reduction filters for all the joining attributes contained in R₄, B, C, D, and F, exist and are applied to R₄. The reduction filters constructed during this process are B:2, C:3, D:4, and F:6. We have the following reduced relation R₄.

B	C	D	F
2	3	4	6

Figure 13 The Reduced Relation R₄

The filters for C and D have changed, and therefore, the relation R_3 is added to the queue since it is not the current relation, is not on the queue and has been processed before. Since R_4 has just been processed, it is not added to the queue. At this point, all relations in the query graph have been processed. We now proceed to Phase 2 to process the relations on the queue.

Phase 2: Processing of Queue

During this phase, the relations in the queue are processed again and any filters that are applied are updated. During this process, more relations may subsequently be added to the queue as the filters change. This phase is repeated until the queue becomes empty. The processing of the queue is described below:

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

From the example query, the queue $Q:R_1, R_3$ is produced in Phase 1. R_1 will be processed first. The reduction filter for attribute A is applied to R_1 and updated. The new filter for A is A:1. The reduced relation R_1 is given below.

A	H
1	8
1	7

Figure 14 The Reduced Relation R_1

There is no change to the filter so no relations are added to the queue. R_3 is taken from the queue next. The filters for attributes C and D are applied to R_3 and updated. The new filters are C:3 and D:4. We have the reduced relation R_3 below.

C	D	I
3	4	9

Figure 15 The Reduced Relation R_3

Since no filters have changed, no relations are added to the queue. The queue is now empty and the algorithm terminates.

The final set of reduced relations is shown in figure 16. The join of the set of reduced relations is shown in figure 17. The join of the original, unreduced relations produces the same result. Notice how the algorithm can fully reduce all relations to only the necessary tuples needed for the final join! Therefore, this example shows that our proposed algorithm achieves significant reduction in the relation sizes.

R1	A	H						
	1	8						
	1	7						

R2	A	B	E					
	1	2	4					

R3	C	D	I					
	3	4	9					

R4	B	C	D	F				
	2	3	4	6				

R5	E	F						
	4	6						

Figure 16 The Set of Reduced Relations

A	B	C	D	E	F	H	I
1	2	3	4	4	6	8	9
1	2	3	4	4	6	7	9

Figure 17 The Join of the Reduced Relations

3.2 Conclusion

In this chapter, we have presented the proposed algorithm in detail and have illustrated it with an example. The main goal of the algorithm is to reduce the size of all relations, while incurring minimum data transmissions. The example demonstrates that the algorithm may achieve a significant reduction in relation sizes. In the next chapter, we present the evaluation framework and experiments used to test this hypothesis.

Chapter 4 EVALUATION

In this chapter we provide the rationale for our experimental work. We describe the framework and give details of our experiments. The aims of the evaluation are:

1. To compare the algorithm against the effects of a full reducer. A full reducer is simulated by joining the query relations and determining which tuples of each relation participate in the join. The cardinalities of the 'fully reduced' relations are used to determine the amount of full reduction. This comparison is done under the assumption of a perfect hash function.
2. To determine how collisions affect the amount of reduction of the algorithm. For each query of the collision evaluation, a percentage of collisions, between 1% and 60% is incorporated. The simulated full reducer used above will be used here to compare the results of the algorithm against full reduction. The results will also be compared against the results that assume a perfect hash function.

4.1 Experimental Rationale

With few exceptions, previously proposed algorithms have not been objectively evaluated. Heuristics have been evaluated for performance by comparison with another heuristic [MBB95b, MO97, Osb96, CCY92, Mor96, PC90,

MBBK95, Ma97, MM98, MO98, Bea95] or not evaluated for performance at all [BGW⁺81, AHY83, CL84, WCS92, WLC91, CY93, Mul90, CL90, TC92]. Although a comparison can determine the improvement in performance of one algorithm over another, it can not determine how close an algorithm comes to achieving full or optimal reduction in relations. Some previously proposed algorithms have been evaluated theoretically by performing a time complexity analysis. However, a theoretical evaluation does not determine how the algorithm will perform when given real-life data with which to work. Complexity alone does not determine how good an algorithm is.

For these reasons, it is preferable to evaluate the algorithm against a full reducer. A full reducer is an algorithm that fully reduces all relations involved in a query by eliminating all non-participating tuples from the relations. Therefore, our algorithm will be evaluated to determine how close it comes to achieving full reduction under various conditions. Our approach to evaluating an algorithm is objective, since its performance is not being compared with the performance of another algorithm. Our approach to evaluation is better since it will allow us to quantify the amount of reduction possible. Also, our approach will provide new insights into the problem of distributed query optimization that are not attainable by other traditional evaluation techniques.

4.2 Evaluation Framework

The evaluation framework consists of a collection of software for generating queries and relations; for executing the queries and compiling the results; and for analyzing the experimental results.¹

4.2.1 Individual Queries

The algorithm is evaluated using select-project-join (SPJ) queries. Each query consists of an arbitrary number of relations, each containing an arbitrary number of joining attributes. The relations vary in the following ways:

- Relation cardinality – the number of tuples or records in a relation.
- Attribute domain sizes – the total number of distinct attribute values an attribute can contain.
- Selectivity – defined as 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 0.01 is considered high while a selectivity of 0.95 is considered low.
- Connectivity – an approximate ratio of the number of joining attributes appearing in all relations of the query over the total number of possible join

¹ This evaluation software was programmed by various members of the Database Research Group of the University of Windsor

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).

Individual queries are generated using *qscript*, a Tcl script. Each relation specified by *qscript* is generated by a C program called *relbuilder*. Both *qscript* and *relbuilder* are described in [Bea95], and are described below for completeness:

- *qscript*. This program generates a query. The input includes the number of relations, the number of common join attributes, the range of relation cardinalities, the range of attribute cardinalities, the range of attribute selectivities, and the connectivity. This data is stored in a 'qspeccs' file which is read in by *qscript*.² The output of *qscript* consists of the database statistics, domain sizes and files containing data for generating relations.

The 'dbstats' file contains the number of relations, the number of common joining attributes, the relation cardinalities, and, for each attribute in each relation, its cardinality and selectivity. For example, consider the 'dbstats' file for a query, given in figure 18. Line 1 contains the number of relations (3) and the number of common joining attributes (2). Lines 2, 3 and 4 contains the statistics for each relation specified in the query. For relation R_0 , represented by line 2 of 'dbstats', the cardinality is 232. R_0 also contains two joining

² Alternatively, the relation count and attribute count can be specified as command line arguments.

dbstats

3 2

232 197 0.9 165 0.75

464 0 0.0 191 0.87

380 129 0.59 174 0.79

Rel 0

232 2 0 197 218 1 165 219

domains

218

219

Figure 18 Example Files for a Query

attributes — attribute 0, with a cardinality of 197 and a selectivity of 0.9, and attribute 1, with a cardinality of 165 and a selectivity of 0.75. Relations R_1 and R_2 are represented by lines 3 and 4 of the dbstats and contain the same statistical information that is contained in line 2.

The 'domains' file consists of the domain cardinality for each common joining attribute. For example, the 'domains' file in figure 18 contains a domain size of 218 for common join attribute 0 and a domain size of 219 for common join attribute 1.

For each relation specified in the query, a 'Rel' file is generated, which consists of the relation cardinality, the number of joining attributes, and for each

joining attribute, the attribute label, the cardinality of the attribute and the cardinality of its domain. For example, we have a 'Rel' file, 'Rel0', for relation R_0 given in figure 18. The cardinality of R_0 is 232. R_0 contains two attributes — attribute 0, with a cardinality of 197 and a domain size of 218, and attribute 1, with a cardinality of 174 and a domain size of 219. The domain sizes are obtained from the 'domains' file. Similar 'Rel' files are created for relations R_1 and R_2 .

- *relbuilder*. The *relbuilder* program generates a relation based on the statistics generated in *qscript*. The input to *relbuilder* is a number indicating the relation to generate. *Relbuilder* uses this number to access the appropriate 'Rel' file, which is generated by *qscript*. The output is a relation, which contains the required number of tuples and the necessary header information, including the number of attributes, the number of joining attributes, and the joining attribute labels. The joining attributes assume either normal or random distribution of attribute values, which depends on which distribution is specified in the *relbuilder* source code. It should be noted that the original *relbuilder* used in [Bea95] does not output the total number of attributes contained by the relation in the header file. For the purposes of these experiments, it was necessary to modify the code to incorporate this count.

It should be noted that both algorithms described above can only generate queries containing three to six relations and two to four joining attributes. Therefore, for

the purposes of this evaluation, each combination of a relation 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.

4.2.2 Individual Runs

The algorithm is evaluated with over 10,000 queries that vary in many ways including the number of relations and joining attributes, relational cardinality, domain cardinality, selectivity, and the percentage of collisions. In order to effectively evaluate this algorithm with such a large number of diverse queries, it is necessary to split up the queries into runs. For the purpose of this evaluation, a run executes 600 queries, comprising 50 queries for each of the query types described above. A C shell script, `runexp`, is used to execute a run.

The output from each run consists of a file for each query type. Each file contains the statistics gathered from the 50 queries, and includes the unoptimized relation cardinalities, the final cardinalities from the full reducer and the final cardinalities from the algorithm.

4.2.3 Analysis of Data Files

The data files created from each run are analyzed using two analysis programs. *Analysis1* determines for each query type, and for the entire run, the average

percentage full reduction that is achieved by the algorithm. *Analysis2* determines for each query type, and the entire run, the percentage of queries that achieve full reduction. Therefore, for each query type, and overall for all 50 queries, the following will be determined:

1. The average percentage reduction produced by the algorithm, with respect to full reduction.
2. The percentage of queries that achieve full reduction.

Given the overall averages for each run, the average percentage of reduction and the overall percentage of queries that achieve full reduction for all runs will be determined.

4.3 The Experiments

The experiments carried out are divided into two sets. Set 1 evaluates the performance of the algorithm under the assumption of the perfect hash function. With the perfect hash function, all attribute values hash to the address specified by the value. For example, an attribute value of 4 will hash to address 4 in the filter. Using this method of 'hashing' ensure that no collisions will occur. Set 2, the collision runs, evaluates the performance of the algorithm given the occurrence of specific percentages of collisions.

Run#	Selectivity	Connectivity
1	0.02-0.4	75%
2	0.4-0.7	75%
3	0.7-0.95	75%
4	0.4-0.7	100%
5	0.4-0.7	75%

Table 1 Initial Runs

4.3.1 Initial Runs

The main purpose of the initial runs is to determine how well the algorithm performs without the effect of collisions. We also wish to determine if varying the selectivity of the joining attributes and the connectivity of the query affects the performance of the algorithm. The initial five runs are shown in table 1. In these five runs, the relations consisted of 200 to 600 tuples, while the attribute domains consisted of 150 to 250 distinct values. In the first three runs, the connectivity is set at 75%. The selectivity ranges of 0.02–0.4, 0.4–0.7 and 0.7–0.95, are chosen because we feel they best represent high, medium and low selectivity respectively. In the final two runs, the selectivity is set at 0.4 to 0.7. The connectivities of 100% and 75% are chosen because we feel they best represent full and average connectivity respectively. A run of low connectivity was attempted as well. Unfortunately, many of the queries generated for this run were invalid, so these results are not being considered.

4.3.2 Collision Runs

The purpose of the collision runs is to determine what effect collisions will have on the performance of the algorithm. In the remaining 16 runs, the relations consisted of 200 to 600 tuples, the attribute domains consisted of 150 to 250 distinct values, the query connectivity is set at 75% and the selectivity is set at 0.5–0.95. The final 16 runs are as follows. The first 11 runs evaluate the algorithm at each percentage of collisions between 0% and 10%. The remaining five runs evaluate the algorithm at 20%, 30%, 40%, 50%, and 60% collisions.

4.3.3 Generating Collisions

Our algorithm is evaluated at specific percentages of collisions. To ensure that a specific percentage occurs, we adopt the following method of simulating collisions. Given a common join attribute j , its active domain is determined. The active domain of a common join attribute j is the set of values from the domain of j that are present in all attributes d_{ij} , $i=0..\#relations$ (in other words, all attributes that take their values from the domain of j). Then, $d\%$ of the values in the active domain are chosen as the values that will result in a collision in the filter. We will call this set X . For each of the values in X , its collision address is determined. The collision addresses are taken from the set $\{active\ domain(j) - X\}$, to guarantee that a collision will occur. This process is repeated for all common join attributes in a query.

For example, suppose for common join attribute j , we have the active domain $\{1,3,4,5,7,8,9,10,12,14\}$. If we want 20% collisions, then we would randomly choose 2 of these 10 values as ones which will collide. Let us say that the values 4 and 10 are chosen. Then, the addresses that are set to 1 in the bit filter are chosen from the remaining values of $\{1,3,5,7,8,9,12,14\}$. Let us choose 1 and 14. Therefore, a value of 4 will hash to the address 1 in the bit filter while a value of 9 will hash to address 14. The remaining attribute values will hash to the address represented by the value.

4.4 Conclusion

In this chapter we provide the rationale for our experimental work. We describe the framework and give details of our experiments. The aims of the evaluation are to compare the algorithm against the effects of a full reducer, and to determine how collisions affect the performance of the algorithm. The results of the proposed experiments are presented in detail in the next chapter.

Chapter 5 RESULTS

In this chapter, we present the results of the performance evaluation. The observations based on the results of the initial runs will be presented first, followed by the observations based on the results of the collision runs. A discussion on some other interesting results will also be presented.

5.1 Results of Initial Runs

The main purpose of the initial runs is to determine how well the algorithm performs without considering collisions, while the secondary purpose is to determine the effect of varying the selectivity and connectivity on the performance of the algorithm.

Type	selectivity 0.02-0.4		selectivity 0.4-0.7		selectivity 0.7-0.95		connectivity 100%		connectivity 75%	
	Avg	Full	Avg	Full	Avg	Full	Avg	Full	Avg	Full
3-2	99.93	94	94.18	52	76.10	40	98.05	84	94.32	46
3-3	99.97	98	98.33	90	89.62	78	100	100	98.37	86
3-4	100	100	100	100	100	100	100	100	99.71	98
4-2	100	100	98.20	70	87.59	58	100	100	98.20	70
4-3	100	100	99.38	90	98.85	96	100	100	99.02	90
4-4	100	100	100	100	99.74	98	100	100	99.96	98
5-2	100	100	99.40	82	93.95	78	100	100	99.58	92
5-3	100	100	99.84	98	99.49	98	100	100	99.91	98
5-4	100	100	100	100	100	100	100	100	100	100
6-2	100	100	100	100	98.18	88	100	100	99.88	98
6-3	100	100	100	100	99.42	98	100	100	100	100
6-4	100	100	100	100	100	100	100	100	100	100
Avg	99.99	99.33	99.11	90.17	95.25	86	99.84	98.67	99.08	89.67

Table 2 Results of the Initial Runs

- The results of the initial runs show that, in most cases, the algorithm achieves substantial reductions in the sizes of the relations. On average, approximately 98.6% of all tuples not required for the final result are eliminated from the relations involved in the query. Also, these results show that, on average, the algorithm fully reduces the relations in 92.7% of all queries.
- The results of varying the selectivity of the joining attributes show that both the amount of reduction, and the percentage of queries that achieve full reduction, decrease as the selectivities approach 1.0 (in other words,

as the selectivity decreases). The difference in the best average reduction (selectivity of 0.02-0.4) and the worst average reduction (selectivity of 0.7-0.95) is approximately 5%, which is not substantial. However, the difference in the best percentage of fully reduced queries (selectivity of 0.02-0.4) and the worst (selectivity of 0.7-0.95) is approximately 13%, which is substantial.

- The results of varying the query connectivity show that both the amount of reduction, and the percentage of queries that achieve full reduction, increase as the connectivity increases. The difference in the best and worst average reduction of relation sizes is less than 1%, which is not substantial. However, a substantial difference between the best and worst percentages of fully reduced queries is found at 9%.
- Approximately 90% of the queries, especially those with high selectivity and high connectivity, produce null results. However, approximately 99% of these null queries are fully reduced by the algorithm. Therefore, null queries can be detected cheaply by the algorithm. This is very important since the shipment of large volumes of useless data can be avoided.
- Queries of types 3-2 and 4-2, in many cases, have substantial lower amounts of data reduction than queries of other types. Also, queries of types 3-2 and 4-2 are the least likely to be fully reduced.

5.2 Results of Collision Runs

The initial results show that, on average, the algorithm achieves significant reduction of unneeded data in query relations, and achieves an acceptable percentage of fully reduced queries. However, the initial runs did not take into consideration the effect of collisions on the performance of the algorithm. Therefore, the main question to be answered in this section is: *How does the existence of collisions affect the performance of the algorithm?*

The results of the collision runs are divided into two groups. The first group consists of the runs that evaluate the algorithm at collision rates from 0% to 10%. The second group consists of the runs that evaluate the algorithm at 10%, 20%, 30%, 40%, 50%, and 60% collisions.

%Coll	0%	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%
3-2	83.54	83.48	84.60	84.93	84.69	83.87	84.19	81.94	81.59	83.36	85.26
3-3	95.34	97.38	93.95	92.48	95.15	93.95	95.56	93.07	96.15	92.95	95.75
3-4	99.26	100	100	99.81	99.60	100	100	100	99.51	100	99.55
4-2	91.51	92.37	94.92	95.69	94.88	94.97	93.03	90.57	91.85	91.68	93.54
4-3	99.05	99.09	98.66	97.21	98.29	98.89	97.96	99.36	98.52	99.21	97.45
4-4	100	100	100	99.87	100	100	99.72	99.77	100	100	100
5-2	99.55	96.72	97.59	98.55	96.91	97.87	96.40	96.75	97.68	96.86	96.01
5-3	100	100	99.85	99.82	99.96	99.99	99.84	98.70	98.65	99.98	99.64
5-4	100	100	100	100	100	100	100	100	100	100	100
6-2	98.55	98.92	98.74	98.98	99.03	99.67	99.11	99.12	98.30	98.11	99.28
6-3	100	100	99.97	99.99	100	100	99.76	99.74	100	99.93	100
6-4	100	100	100	100	100	100	100	100	100	100	100
Avg	97.23	97.33	97.36	97.27	97.38	97.43	97.13	96.59	96.85	96.84	97.21

Table 3 Average Percentage Reduction at 0%-10% Collisions

- On average, 97.2% of all unneeded tuples are eliminated by the algorithm when the collision rate is between 0% and 10%. The range of the average percentage reductions from the first group of runs is less than 1%. Therefore, on average, the algorithm consistently gives substantial reductions in relation sizes, given the existence of 0% to 10% collisions.
- For query type 3-2, the algorithm gives average percentages of reduction which are substantially lower than those of the remaining query types. The algorithm still achieves 90+% elimination of unneeded data for the remaining query types.

%Coll	0%	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%
3-2	34	28	26	36	32	16	26	18	16	16	24
3-3	90	84	78	76	84	78	82	72	84	70	76
3-4	96	100	100	98	98	100	100	100	98	100	98
4-2	44	52	58	68	54	54	56	48	42	50	54
4-3	94	92	96	92	94	96	90	94	92	96	84
4-4	100	100	100	98	100	100	98	98	100	100	100
5-2	88	76	68	78	64	72	60	64	74	64	70
5-3	100	100	98	96	96	98	98	94	92	96	96
5-4	100	100	100	100	100	100	100	100	100	100	100
6-2	84	82	90	72	76	86	84	78	78	72	90
6-3	98	100	98	98	100	100	96	98	98	98	100
6-4	100	100	100	100	100	100	100	100	100	100	100
Avg	85.67	84.50	84.33	84.33	83.17	83.33	82.50	80.33	81.20	80.33	82.67

Table 4 Average Percentage of Fully Reduced Queries at 0%-10% Collisions

- On average, 82.9% of all queries are fully reduced by the algorithm when the collision rate is between 0% and 10%. The range of the average percentage of fully reduced queries is approximately 5%. Although this average is less significant than the average percentage reduction of data sizes, it appears to still be consistent, given the existence of collisions between 0% and 10%.
- Query types 3-2 and 4-2 have the worst percentages of fully reduced queries — less than 60% of these queries are fully reduced by the algorithm. Query types 3-3, 5-2 and 6-2 also have lower percentages of queries achieving full reduction, but not as substantially low as 3-2 and 4-2. The remaining

query types fully reduce 90+% of queries, given the existence of 0% to 10% collisions.

%Coll	10%	20%	30%	40%	50%	60%
3-2	85.26	80.11	75.21	73.31	72.74	69.56
3-3	95.75	93.15	87.27	90.53	86.10	88.58
3-4	99.55	99.02	98.95	97.92	98.58	97.11
4-2	93.54	92.13	89.29	86.25	87.43	81.23
4-3	97.45	97.83	96.41	98.34	94.53	96.62
4-4	100	100	100	100	99.63	98.31
5-2	96.01	96.39	95.39	92.22	91.21	89.37
5-3	99.64	98.38	98.98	96.56	98.72	98.10
5-4	100	100	100	100	100	99.77
6-2	99.28	97.88	96.53	98.55	97.41	95.81
6-3	100	100	99.58	99.13	99.59	99.44
6-4	100	100	100	100	100	100
Avg	97.21	96.24	94.80	94.40	93.83	92.82

Table 5 Average Percentage Reduction at 10%-60% Collisions

- As the percentage of collisions increases after 10%, the amount of reduction begins to decline slightly. The average percentage reduction is 94.9%, with the average at 10% collisions being 97.2% and the average at 60% collisions being 92.9%. Therefore, after 60% collisions, the average amount of reduction is still substantially high.
- As with the 0% to 10% collision range, query type 3-2 achieves the worst average percentage reduction of the relations. Query types 3-3 and 4-2 also

achieves lower percentages of reduction, although not as substantially low as query type 3–2. For the remaining query types, the our algorithm still achieves 90+% elimination of unneeded data in almost all cases.

- On average, 96% of the null queries are being fully reduced by our algorithm. In the presence of 0% to 10% collisions, this is very substantial — many null queries can still be detected cheaply by the algorithm.

%Coll	10%	20%	30%	40%	50%	60%
3-2	24	4	2	4	0	0
3-3	76	70	58	62	54	58
3-4	98	96	96	94	92	86
4-2	54	32	32	16	30	14
4-3	84	84	84	90	80	82
4-4	100	100	100	100	98	94
5-2	70	60	66	46	44	34
5-3	96	90	94	88	94	86
5-4	100	100	100	100	100	98
6-2	90	76	66	78	76	66
6-3	100	100	98	92	98	96
6-4	100	100	100	100	100	100
Avg	82.67	76.00	74.67	72.50	72.17	67.83

Table 6 Average Percentage of Fully Reduced Queries at 10%-60% Collisions

- As the percentage of collisions increases after 10%, the percentage of fully reduced queries declines substantially. Between 10% and 20% collisions, a decline of almost 7% occurs. Between 20% and 50% collisions, the

percentage of fully reduced queries ranges between 72% and 76%. At 60%, the percentage of fully reduced queries makes another significant decline of approximately 4.5%. Therefore, after 10% collisions, the amount of collisions substantially affects the number of queries achieving full reduction.

- Query types 3–2 and 4–2 have percentages of fully reduced queries that are significantly lower than all other query types. In fact, after 40% collisions, none of the queries of type 3–2 are being fully reduced! Query types 3–3, 4–3, 5–2 and 6–2 have the next lowest percentages of fully reduced queries. For the remaining query types, the algorithms still fully reduces 90+% of queries in most cases, even in the presence of 60% collisions.
- Even in the presence of 10% to 60% collisions, the algorithm still fully reduces 88% of the null queries. Therefore, many of the null queries can still be detected by our algorithm.

5.3 Discussion

The performance evaluation shows that, on average, the algorithm gives substantial reductions of relation sizes, even when collisions are a problem. Also, for lower percentages of collisions, the algorithm fully reduces an acceptable percentage of queries. However, other trends have been determined from the experimental results. One trend is that queries containing a lower number of joining attributes and fewer relations almost always achieve both the worst

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

Figure 19 Example Relations

reduction in relation sizes and the lowest number of full reductions. It was also found that both the selectivity of the joining attributes and the query connectivity affect both the amount of reduction and the number of fully reduced queries. A further analysis of the results and the algorithm has revealed the following. When a query contains few joining attributes with low selectivities, the filtering effect of the reduction filters is hindered. This is illustrated with a simple example.

We have two relations, R_1 and R_2 , both containing joining attributes A and B, given in Figure 19. The join of these two relations will result in the following relation:

A	B
2	1

The fully reduced R_1 and R_2 are:

R1	A	B	R2	A	B
	2	1		2	1

Suppose the algorithm processes R_1 first. R_1 is scanned to create the following filters for A and B: A:1,2,3,5,6 and B:1,2,4,5,6. If we assume a domain size of six

distinct attribute values for both A and B, then the selectivities of A and B will be 0.83. When the filters for A and B are applied to R_2 , we have the following unexpected result:

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

No reduction has occurred in R_2 . When the filters for A and B are recreated, we have the same filters as before. Therefore, we have no additions to the queue and the algorithm terminates.

The reason why no reduction occurs in R_2 is the following. For each tuple in R_2 , the attribute values for A and B are being hashed and set in each filter by two different tuples in R_1 . For example, the first tuple in R_2 has A equal to 1 and B equal to 4. Although no matching tuple exists in R_1 , the first tuple in R_1 contains A equal to 1 while the second tuple in R_1 contains B equal to 4. Both of these values will be hashed and set to 1 in their respective filters. When the first tuple of R_2 is tested, it passes the filters because both of its attribute values hash to a 1 bit in their respective filters!

Although this hinderance in reduction affects queries with few joining attributes and relations, it does not appear to be a problem with the queries con-

taining a higher number of joining attributes and relations. One possible reason for this is that, even though each joining attribute may have a high selectivity, the products of the various selectivities of a common joining attribute can lower the overall selectivity of the common joining attribute. Therefore, the higher number of joining attributes decrease the chance of a tuple falsely passing all necessary filters.

5.4 Conclusion

In this chapter, the results of the performance evaluation have been presented. The initial results show that the algorithm achieves both significant reductions in relation sizes and an acceptable percentage of fully reduced queries. When determining what effect the existence of collisions has on the performance of the algorithm, it is found that the algorithm still achieves substantial relation reductions. The algorithm also achieves an acceptable percentage of fully reduced queries for low percentages of collisions. Some other interesting results were found and discussed.

Chapter 6 CONCLUSIONS

In this thesis, a new filter-based algorithm is proposed that uses filters to accomplish the same reduction effects as semijoins, but at a lower cost. The primary goal of our algorithm is to reduce relation sizes while incurring minimum data transmission costs. The secondary goal is to incur minimum processing costs by processing each relation as little as possible. This algorithm can process general queries consisting of an arbitrary number of relations and joining attributes, and it does not assume the use of a perfect hash function.

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. Using the results of the evaluation, we now answer the following questions:

- On average, how much reduction, with respect to the full reducer, is achieved?
On average, our algorithm achieves substantial reductions in the sizes of query relations. Approximately 97–99% of all tuples not required for the final result are eliminated from the relations involved in the query.
- Full reduction of relations is achieved in what percentage of queries?
The initial results show that the relations of approximately 93% of all query are fully reduced by our algorithm. The 0% collision run shows that approx-

imately 86% of all queries are fully reduced. The reason for the difference in averages is that the selectivity range for the 0% collision run is larger (0.5–0.95) than the selectivity ranges of all the initial runs. However, a substantial number of queries achieve full reduction, although the percentage is not as substantial as the average percentage reduction of query relations.

- What effect do collisions have on the amount of reduction?

Results show that, no matter how high the percentage of collisions, the average percentage of reduction is still substantial. The average for 0% to 10% collisions is approximately 97%, while between 10% and 60% it is approximately 95%. The worst average percentage reduction, at 60% collisions, was found to be 93% — still substantial given that 60% collisions are occurring.

- What effect do collisions have on the percentage of queries achieving full reduction?

Results show the percentage of queries that achieve full reduction is 82% when the percentage of collisions is between 0% and 10%. However, this percentage substantially declines when the percentage of collisions increases after 10%. At 60%, only 68% of all queries achieve full reduction. Although this decrease is substantial, the average between 0% and 10% is still acceptable.

In conclusion, if a reasonably uniform hashing function is used in our algorithm, then our algorithm performs significantly well, with respect to both the average

percentage reduction of query relations and the percentage of queries that achieve full reduction. This is a reasonable conclusion since it is desirable to use a uniform hash function which results in few collisions.

6.1 Future Research Directions

Results show that, for higher percentages of collisions, the percentage of queries that achieve full reduction decreases. One direction of research is to use multiple reduction filters for each common joining attribute to attempt to minimize the collision problem.

Results also show the filtering effect of the algorithm is hindered when queries contain few relations and few joining attributes. Many tuples that are not required for the final result are accidentally passing the filter tests. This is also a problem when the selectivity of the joining attributes is low. Therefore, another direction of research is to find a solution to this problem of false accepts.

SELECTED BIBLIOGRAPHY

- [AHY83] P.M.G. Apers, A.R. Henver, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering*, 9(1):57–68, 1983.
- [Bea95] W.T. Bealor. Semijoin strategies for total cost minimization in distributed query optimization. Master's thesis. University of Windsor, 1995.
- [BGW⁺81] P.A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J.B. Rothnie. Query processing in a systems for distributed databases (sdd-1). *ACM Transactions on Database Systems*, 6(4):602–625, 1981.
- [Blo70] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [BR88] P. Bodorik and J.S. Riordon. A threshold mechanism for distributed query processing. In *Proceedings of the ACM Computer Science Conference*, pages 616–621, 1988.
- [Bra84] Kjell Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the Tenth International Conference on VLBD*, pages 323–333, 1984.

- [BRJ89] P. Bodorik, J.S. Riordon, and C. Jacob. Dynamic distributed query processing techniques. In *Proceedings of the ACM Computer Science Conference*, pages 348–357, 1989.
- [BRP92] P. Bodorik, J.S. Riordon, and J.S. Pyra. Deciding to correct distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, pages 253–264, 1992.
- [CCY92] T.-S. Chen, A.L.P. Chen, and W.-P. Yang. Hash-semijoin: A new technique for minimizing distributed query time. In *Proceedings of the 3rd Workshop on Future Trends of Distributed Computing Systems*, pages 325–330, 1992.
- [CL84] A.L.P. Chen and V.O.K. Li. Improvement algorithms for semijoin query processing programs in distributed database systems. *IEEE Transactions on Computers*, 33(11):959–967, 1984.
- [CL90] J.S.J. Chen and V.O.K. Li. Domain-specific semijoin: a new operation for distributed query processing. *Information Sciences*, 52:165–183, 1990.
- [CP84] S. Ceri and G. Pelagetti. *Distributed Databases: Principles and Systems*. McGraw-Hill, 1984.
- [CY90a] M.-S. Chen and P.S. Yu. Using combination of join and semijoin operations for distributed query processing. In *Proceedings of the*

10th International Conference on Distributed Computing Systems, pages 328–335, 1990.

- [CY90b] M.-S. Chen and P.S. Yu. Using join operations as reducers in distributed query processing. In *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems*, pages 116–123, 1990.
- [CY91] M.-S. Chen and P.S. Yu. Determining beneficial semijoins for a join sequence in distributed query processing. In *Proceedings of the 7th International Conference on Data Engineering*, pages 50–58, 1991.
- [CY92] M.-S. Chen and P.S. Yu. Interleaving a join sequence with semijoins in distributed query processing. *IEEE Transactions on Parallel and Distributed Systems*, pages 611–620, 1992.
- [CY93] M.-S. Chen and P.S. Yu. Combining join and semijoin operations for distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, pages 534–542, 1993.
- [CY94] M.-S. Chen and P.S. Yu. A graph theoretical approach to determine a join reducer sequence in distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, pages 152–164, 1994.
- [Hen80] A.R. Henver. *The optimization of query processing in distributed*

database systems. PhD thesis, Purdue University, 1980.

- [KR87] H. Kang and N. Roussopoulos. Using 2-way semijoins in distributed query processing. In *Proceedings of the 3rd International Conference on Data Engineering*, pages 644–651, 1987.
- [LMH⁺85] G.M. Lohman, C. Mohan, L.M. Haas, D. Daniels, B.G. Lindsay, P.G. Selinger, and P.F. Wilms. Query processing in R*. In *Query Processing in Database Systems*, pages 31–47. Springer, New York, 1985.
- [LPP91] P. Legato, G. Paletta, and L. Palopoli. Optimization of join strategies in distributed database. *Information Systems*, 16(4):363–374, 1991.
- [Ma97] X. Ma. The use of bloom filters to minimize response time in distributed query optimization. Master’s thesis, University of Windsor, 1997.
- [MB97] J.M. Morrissey and W.T. Bealor. Minimizing data transfers in distributed query optimization: A comparative study and evaluation. *Computer Journal*, 39(8):675–687, 1997.
- [MBB95a] 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.

- [MBB95b] J.M. Morrissey, S. Bandyopadhyay, and W.T. Bealor. A heuristic for minimizing total cost in distributed query processing. In *Proceedings of the 7th International Conference on Computing and Information — ICCI'95*, pages 736–758, 1995.
- [MBBK95] J.M. Morrissey, S. Bandyopadhyay, W.T. Bealor, and S. Kamat. Dynamic strategies and bloom filters for minimizing data transfers in distributed query optimization. University of Windsor, working paper, 1995.
- [MBK96] J.M. Morrissey, W.T. Bealor, and S. Kamat. A comparative evaluation of dynamic heuristics for cost minimization. In *Proceedings of the Eighth International Conference on Computing and Information (ICCI '96)*, pages 700–716, 1996.
- [ML86] L.F. Mackert and G.M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the Twelfth International Conference on VLDB*, pages 149–159, 1986.
- [MM98] J.M. Morrissey and X. Ma. Investigating response time minimization in distributed query optimization. In *Proceedings of the International Conference on Computing and Information (ICCI' 98)*, pages 157–164, 1998.
- [MO97] J.M. Morrissey and W.K. Osborn. Experiments with the use of

- reduction filters in distributed query optimization. In *Proceedings of the Ninth IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 327–330, 1997.
- [MO98] J.M. Morrissey and W.K. Osborn. The use of reduction filters for distributed query optimization. In *Proceedings of the 1998 Canadian Conference on Electrical and Computer Engineering*, pages 707–710, 1998.
- [Mor96] J.M. Morrissey. Reduction filters for minimizing data transfers in distributed query optimization. In *Proceedings of the 1996 Canadian Conference on Electrical and Computer Engineering*, pages 198–201, 1996.
- [Mul83] J.K. Mullin. A second look at bloom filters. *Communications of the ACM*, 26(8):570–571, 1983.
- [Mul90] J.K. Mullin. Optimal semijoins for distributed databases systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.
- [Mul93] J.K. Mullin. Estimating the size of a relational join. *Information systems*, 18(3):189–196, 1993.
- [Osb96] W.K. Osborn. Distributed query optimization using bloom filters. Report, 60–491, University of Windsor, 1996.
- [OV91] M.T. Ozsü and P. Valduriez. *Principles of distributed database*

systems. Prentice Hall International, 1991.

- [PC90] W. Perrizo and C.-S. Chen. Composite semijoins in distributed query processing. *Information Sciences*, 50:197–218, 1990.
- [RK91] N. Roussopoulos and H. Kang. A pipelined n-way join algorithm based on the 2-way semijoin program. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):486–495, 1991.
- [TC92] J.C.R. Tseng and A.L.P. Chen. Improving distributed query processing by hash-semijoins. *Journal of Information Science and Engineering*, 8:525–540, 1992.
- [VG84] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems*, 9(1):133–161, 1984.
- [WC96] C. Wang and M.-S. Chen. On the complexity of distributed query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):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*, 3(3):325–332, 1992.
- [WLC91] C. Wang, V.O.K. Li, and A.L.P. Chen. Distributed query optimization by one-shot fixed-precision semi-join execution. In *Proceedings of the*

7th International Conference on Data Engineering, pages 756–763, 1991.

[YC84] C.T. Yu and C.C. Chang. Distributed query processing. *ACM Computing Surveys*, 16:399–433, 1984.

[YL89] H. Yoo and S. Lafortune. An intelligent search method for query optimization by semijoins. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):226–237, 1989.

[YLG⁺86] C. Yu, L. Lilien, K. Guh, M. Templeton, D. Brill, and A. Chen. Adaptive techniques for distributed query optimization. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 86–93, 1986.

VITA AUCTORIS

Wendy Osborn was born in 1973 in **Windsor, Ontario, Canada**. She graduated from **Tilbury District High School** in 1992. From there she went on to the **University of Windsor** where she obtained a B.C.S. in Computer Science in 1996. She is currently a candidate for the Master's degree in Computer Science at the University of Windsor and will graduate in the Fall of 1998.