1996

# Dynamic strategy and Bloom filters in distributed query optimization.

Sandeep. Kamat
*University of Windsor*

# Dynamic Strategy and Bloom filters in Distributed Query Optimization

**by**

**Sandeep Kamat**

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
1996

0-612-30910-X

ADK1802

# Abstract

Distributed query optimization is an important issue in distributed database management systems, since it can greatly affect the performance of the system. Many query optimization strategies have been proposed to minimize either the total cost or the response time. Most strategies are static in nature in the sense that their construction is based on database statistics which are obtained prior to query execution. In this thesis we investigate the use of dynamic strategies and better estimation techniques in query optimization. Bloom filters are used to obtain better estimates for query processing. Based on the above concept three algorithms are proposed, first using a pure dynamic strategy, the second using Bloom filters and the third using a combination of both. The performance of these algorithms with respect to total cost is compared against the AHY algorithm. The algorithms are executed against a large number of synthetically generated databases and queries. The experiments show a significant improvement over the AHY algorithm. The dynamic strategy shows an improvement over the static strategy and the combination heuristic shows a marginal improvement over the dynamic strategy.

*To my parents*

# Acknowledgments

I would like to express my sincere gratitude and appreciation to Dr. Joan Morrissey, who has been a constant source of inspiration and encouragement. Her support, guidance were invaluable in the successful completion of this thesis. I am thankful to Dr. Subir Bandyopadhyay for his insightful comments and constant support. I would also like to thank Dr. Samson for his comments and suggestions. I would also like to thank my colleague Mr. Bealor. I would like to thank my friends Mohan and Lynn for all the help and support during the completion of this thesis. Also thanks must go to Nandini and Ramesh whose friendship and company was a constant source of strength, happiness and encouragement during the final and critical phase of this thesis. Last, but not least, my family has been a constant source of strength and encouragement which kept my hope and confidence alive throughout the course of this work.

# TABLE OF CONTENTS

# List of Figures

# Chapter 1
# Introduction

Distributed database systems (DDBMs) allow data to be stored at geographically distinct sites [Des92]. These sites are connected to each other via communication networks. Each site is able to process local transactions and may also participate in the execution of global transactions. Global transactions are transactions which access data from several sites. This kind of distribution of data has many advantages. DDBMs allow for data sharing and distributed control, increased reliability in terms of site failures and speed up in query processing. Several commercial databases are now available which offer full-fledged distributed architectures. It is also predicted that in the near future there will be more and more use of distributed systems rather than centralized systems.

One important issue, on which the performance of distributed system depends, is that of query processing. Since a query in a distributed system requires data from several locations to be transferred over slow communication networks, the cost of transferring data is a significant factor. An important issue is to choose a strategy so as to process the query requiring data from several locations in an optimal manner. The central problem is to obtain a sequence of database operations such that the required cost function is minimized. Some of the most important cost functions considered are the response time and the total cost of processing the query in terms of the amount of data transferred over the network.

## 1.1 Problem to be Investigated

Many distributed database query optimizing algorithms depend on static heuristics to derive optimal query processing strategies. Furthermore the statistics used to perform the cost/benefit analysis are estimates of actual values present in the database. This introduces a margin of error in the calculation of intermediate results that can be compounded and propagated throughout the heuristic, often resulting in a suboptimal solution. Our hypothesis is that better optimization strategies can be obtained by

1. Using a dynamic strategy where more up to date information is available
2. Using better estimation techniques

This thesis tests this hypothesis.

## 1.2 The Thesis Objectives

A major objective is to investigate the effect of dynamic strategies and better estimation techniques on the performance of distributed query processing. Four different algorithms are to be investigated. We use Algorithm AHY [AHY83] as the benchmark algorithm against which our algorithms are compared. DW is a dynamic algorithm and Bloom+W uses Bloom filters [Blo70] to obtain better estimates. Bloom+DW is the combination of the above two techniques. Our experimental work attempts to answer the following questions:

1. Which approach is best ? The use of a dynamic strategy or the use of better estimation techniques ?
2. Will a combination of better estimation techniques and a dynamic strategy lead to further improvements ?

The performance of AHY, W, DW and Bloom+W are compared to answer the first question. To answer the second question we compare the performance of DW, Bloom+W and Bloom+DW.

In order to do a systematic comparison of the above algorithms, a synthetic database is generated using random and uniform distributions A large number of queries are generated and executed. The queries are varied in the number of relations and the number of join attributes at different levels of connectivity to cover a wide range of queries. The total cost of executing each query, using each algorithm, is recorded and compared.

## 1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 reviews related work in the area of query optimization using semijoins. The properties of a semijoin operations are discussed. Next several semijoin query processing algorithms are presented. A detailed discussion of using Bloom filters in query optimization and use of Bloom filters is also presented.

Chapter 3 presents the heuristic algorithms to be investigated. These include Algorithm W, the dynamic version of W (DW), W using Bloom filters (Bloom+W ) and finally a dynamic version of W using Bloom filters (Bloom+DW).

In Chapter 4 the frame work for carrying out the experiments is presented. The design of the benchmark queries is explained in detail. The results obtained from the experiments are presented and discussed.

Finally in Chapter 5 we conclude and make suggestions for future work.

# Chapter 2
# Background Review

## 2.1 Introduction

A distributed database system consists of a collection of data in which separate parts of the collection are under the control of a separate DBMS running on independent systems [Des92]. The major advantages of distributed databases are increased data accessibility at local sites, increased reliability and the ability to execute operations in parallel to achieve a lower response time. A user specifies a query in a high level language describing what he/she wants. Various modules of the database system, like the access planner and query optimizer, are used to retrieve the required data. The objective of query optimization is to determine a sequence of operations such that the cost of executing the query is minimized. Here we consider a particular subclass of queries which involve only the select, project, and join operations, hereafter referred to as SPJ queries. Most often the join operation is the costliest operation and tremendous research effort has gone into determining the optimum strategy for executing join operations. In a centralized database the dominant cost is the I/O for performing the join.

In a distributed database system the dominant cost when performing a multi-attribute multi-relation join is that of the data transmission cost involved in transferring relations from one site to another. The local processing cost is relatively insignificant and is often ignored. One of the most popular methods of performing multi-relation joins is to use a sequence of semijoin operations. The semijoin operation eliminates tuples which can't be part of the final result and so they do not have to be transmitted. The problem of obtaining a optimal sequence of semijoins so as to optimize the query is shown to be NP-hard [WC93]. Hence various heuristic algorithms have been developed to obtain a near optimal sequence of semijoins. Most of the research in this area falls under the following four categories: using the join operation [AM91], using semijoin operations [WLC91, BGW$^+$81, AHY83, PC90, CL90, ?], using a combination of both [CY93] and finally those that use improvements techniques [CL84, BPR90]. In this report we present a brief survey of related work concerning semijoin algorithms. In particular we concentrate on dynamic semijoin algorithms [YLG$^+$86, BPR90, BRP89] and algorithms based on Bloom filters [Blo70].

### 2.1.1 Cost Model

Frequently, in distributed query processing the objective is to minimize one of two cost functions: the total cost or the response time. The total cost is the sum of all the costs involved in the transmission of all data. The response time is the elapsed time between the initiation of the query and the time the final results are obtained. As the speed of the networks over which the data are transmitted is relatively low, the data transmission cost is the most important factor. It is assumed that the cost involved in transmitting data from one site to another is linear and can be represented as

$$\text{Cost} = C_0 + C_1 * X$$

where $C_0$ is the start-up cost of establishing a transmission and $C_1$ is a cost coefficient associated with the amount of data transmitted (X). The local processing cost is considered to be negligible and is ignored in most cases.

## 2.1.2 Semijoin Query Optimization

One common operation in distributed query optimization is the semijoin operation. A semijoin between two relations $R_i$ and $R_k$ on attribute j is defined as follows. The relation $R_k$ is projected over attribute j. The resulting distinct values are shipped to the site of relation $R_i$ and relation $R_i$ is reduced using attribute j. The cost of this semijoin is the cost of shipping the projected attribute j on relation $R_k$. The benefit is the amount of reduction caused by attribute j on relation $R_i$. A semijoin is said to be profitable if the benefit is more than the cost of the semijoin. This can be summarized as follows.

$$\text{Cost}\left(d_{ij} \rtimes d_{kj}\right) = \text{Size}(d_{ij})$$

$$\text{Benefit}\left(d_{ij} \rtimes d_{kj}\right) = \text{Size}(R_k) - (S(R_k) \text{ X } \rho(d_{ij}))$$

where $\rho$ is selectivity of the attribute and is defined as the number of distinct values in the attribute divided by the domain size of the attribute.

$$\text{Profit } \left(d_{ij} \rtimes d_{kj}\right) = \text{Benefit}\left(d_{ij} \rtimes d_{kj}\right) - \text{Cost}\left(d_{ij} \rtimes d_{kj}\right)$$

A typical query has many relations and each relation has many joining attributes. Query optimization algorithms perform a cost/benefit analysis on the semijoins involved and select those semijoins which are beneficial for execution. The sequence in which these operations are carried out also affects the overall performance of the algorithms.

One of the earliest systems to use a semijoin algorithm was the SDD-I system [BGW$^{+}$81]. In SDD-I, after the local processing has been performed, all possible semijoins are identified. A cost/benefit analysis is performed for each semijoin and the semijoin with the least cost is selected and added to the schedule. The effect of executing this semijoin is reflected in the remaining relations and their statistics are updated. The cost/benefit analysis is again carried out for the remaining semijoins and the process is repeated until no more profitable semijoins are left. Once the semijoins are carried out, the reduced relations are sent to one site where the final join is performed. This site is chosen so that the cost of transferring all the other reduced relations is minimized. In the final stage, semijoins which were wrongly considered to be beneficial are eliminated. This heuristic is a greedy heuristic and as no backtracking mechanism is provided, it may not always produce the optimal sequence of semijoins.

In [AHY83] two algorithms to minimize either the response time or the total cost for simple queries are proposed. A simple query is one in which, after the initial local processing, each relation has only one common join attribute. Algorithm SERIAL and algorithm PARALLEL are used to reduce the *total cost* and *response time* respectively. Algorithm parallel is described below:

1. Order relation $R_i$ so that $s_1 \leq s_2 \leq \ldots \leq s_m$.
2. Consider each relation $R_i$ in ascending order of size.
3. For each relation $R_j$ ($j < i$), construct a schedule to $R_i$ that consists of the parallel transmission of the relation $R_j$ and all schedules of relations $R_k$ ($k < j$). Select the schedule with minimum response time.

Algorithm serial is as follows :

1. Order relations $R_i$ such that $s_1 \leq s_2 \leq \ldots \leq s_m$.
2. If no relations are at the result node, then select strategy:

   $R_1 \rightarrow R_2 \rightarrow \ldots \rightarrow R_n \rightarrow$ result node.

   Or else if $R_r$ is a relation at the result node, then there are two strategies:

   $R_1 \rightarrow R_2 \rightarrow \ldots \rightarrow R_r \rightarrow \ldots \rightarrow R_n \rightarrow R_r$ or

   $R_1 \rightarrow R_2 \rightarrow \ldots \rightarrow R_{r-1} \rightarrow R_{r+1} \rightarrow R_n \rightarrow R_r$

In [AHY83] an improved strategy for general queries is presented. General queries are characterized by multiple relations and each relation can have multiple join attributes. Algorithm general is as follows:

1. Do all initial local processing.
2. Generate candidate relation schedule. Consider each join attribute to define a simple query.

   a. To minimize response time, apply algorithm PARALLEL to each simple query.

   b. To minimize total time, apply Algorithm SERIAL to each simple query.

3. Integrate the candidate schedules. The integration is done by procedure RESPONSE for response time minimization and by procedure TOTAL or procedure COLLECTIVE for total time minimization.

4. Remove schedule redundancies. Eliminate relation schedules for relations which have been transmitted in the schedule of another relation.

All the schedules generated by the application of algorithm PARALLEL are integrated using procedure RESPONSE for response time minimization.

Procedure RESPONSE is given below:

1. Candidate schedule ordering. For each relation $R_i$, sort the candidate schedules on the joining attribute in ascending order of arrival time.

2. Schedule Integration. For each candidate schedule construct an integrated schedule that consists of the parallel transmission of candidate schedules having a smaller arrival time. Select the integrated schedule with the minimum response time.

Procedure RESPONSE takes maximum advantage of the parallel transmission of attributes. This can sometimes lead to traffic congestion on computer networks and cause performance degradation. Hence algorithm TOTAL is proposed particularly for multiprocessing environments. The query processing strategy derived by algorithm GENERAL (total time) is not optimal. This is because procedure TOTAL does not consider the existence of redundant data transmissions in separate relation schedules. An alternative version of algorithm GENERAL (total time) called algorithm GENERAL (collective) which is based on redundant transmission is as follows:

1. Select candidate schedule. For each relation $R_i$ and for each joining attribute $d_{ij}$, select the minimum cost candidate schedule that contains the transmission of all components of attribute j with selectivities < 1.
2. Build processing strategy. For each relation $R_i$, define the schedule to be the parallel transmission of all $d_{ij}$ candidate schedules to $R_i$.
3. Test variations of strategy. Using a removal heuristic, derive new strategies by removing the most costly data transmissions. Compare the total time cost of the new and old strategies. Maintain the less costly strategy and continue testing until no cost benefit can be obtained.

## 2.2 Dynamic Strategies in Query Optimization

Static strategies rely on accurate estimates of intermediate results to derive optimal solutions. If there is an error in the estimation, the results may be far from optimal. To solve this problem two approaches can be taken. The first one is to seek more accurate estimates. This often proves to be expensive in terms of the size and maintenance of the required statistical data. The second strategy is to use adaptive query execution techniques. In [YLG$^+$86] some simple adaptive techniques for the estimation of

1. the size of a relation resulting from a join or a semijoin
2. the data transfer rate between the sites and
3. the local processing costs.

are presented.

Only the first is relevant to our work and it is described below. The estimation of the size of a relation due to a join or a semijoin operation is usually based on the assumption that the attribute values are uniformly and independently distributed. The authors propose a cost estimation technique where the initial cost estimation formula is static and based on the assumptions mentioned above. The formula is dynamically adapted to a changing environment by comparing the actual number of tuples returned with the estimated value. If the two numbers are drastically different and one number is consistently below or consistently above the other, then the two numbers will be recorded and used to adjust the cost estimation formula. In [YLG$^+$86] it is suggested that in order to avoid the propagation of errors committed in the strategy formulation phase, only a semijoin or a small number of semijoins at a time should be planned and then executed. Costs and the size of the results can be obtained accurately and further calculations can be carried out with the newly updated values. Another strategy [Ngu81] is to use a Threshold mechanism. Initially a static strategy is generated together with a dynamically updated threshold value. The threshold value is the average size of the partial results. A "purely" dynamic query execution is triggered when the average size of the intermediate results is greater than the threshold value. In a purely dynamic execution a complete strategy is not formulated; instead, an optimizing algorithm is used to plan the next relational operation which is then immediately executed. Information about the result is obtained and used in deciding on the next operation to be executed. In [BPR90] the use of threshold values for dynamic query execution is also proposed. In comparison to the method in [YLG$^+$86] they propose a method to determine a execution value for each partial result formed in the strategy execution instead of one value per query as suggested by [YLG$^+$86].

In [BRP89] a three phase approach for dynamic query processing is proposed.

1. *Monitoring Phase* : Processors exchange information about the progress of the strategy execution and collect relevant information. Monitoring continues until the completion of processing or until it is decided to correct the strategy. Monitoring can either be distributed or individual or at a master site. In distributed monitoring the processors exchange information about the progress of the strategy execution using broadcasting. Any time a local processor forms a partial result by performing a operation, it broadcasts its cost and size. Any information processor may decide to correct the strategy. This strategy incurs high overhead due to the broadcasted messages. In master monitoring, a master site keeps track of the query execution and all nodes report to this site. In individual monitoring, information processors do not co-operate in

the monitoring process at all. Any time a processor completes a relational operation, it decides whether or not the strategy should be corrected using only its local information.

2. *Decision making Phase*: A decision is made whether or not to correct the current strategy because its cost or the size of its partial results differ "too much" from their estimates. This decision can be made through the *Reformulation* or *Thresholds* approach. In the reformulation approach, any time a new partial result is formed, the unprocessed portion of the strategy is reformulated using available up-to-date information. If the new strategy leads to a lower execution cost, a correction is made. In the thresholds strategy some information is provided in support of the decision making process regarding corrections. For each partial result two threshold values are prepared. The strategy is corrected whenever the actual value crosses the threshold values. A variation of this method utilizes only one threshold value for the query, the average size of partial results. In addition, this value is dynamically updated throughout the strategy's execution.

3. *Corrective Phase*: The current strategy is aborted and a new, corrective strategy is initiated. This correction can be carried out either through *Centralized Correction*, *Distributed Correction* or *Individual Correction*. In centralized correction only one information processor, the master, can formulate the new corrective strategy. In distributed correction all processors supply each other with information about their partial results. In individual correction the pro-

cessor which decides to abort formulates the corrective strategy and broadcasts it to the other processors.

## 2.3 Bloom Filters in Query Optimization

The idea of a Bloom filter was introduced by Bloom in [Blo70]. This paper introduced two new hash coding methods for applications where an error margin due to collisions was tolerable. This tolerance of a margin of error allows the size of the filters to be reduced, so that they can be kept in main memory for rapid access. For example, consider an application where a set of words is to be hyphenated. Also assume that in most of the cases (90 %) the rules for performing these hyphenations are simple. In the remaining 10% it might be necessary to perform an dictionary lookup. These words can be represented using a Bloom filter. Whenever a new word is to be hypenated it is hashed and checked to see if it is present in the Bloom filter. If it is present then a directory lookup is performed. If the word is not present in the Bloom filter then the hyphenation is carried out in the normal manner without a disk access. In some cases a word might be wrongly hashed into the Bloom filter. In such cases an extra disk access is performed. By choosing the size and the hashing transformations carefully it is possible to make this error insignificantly small. The paper also analyses the time/space requirements for the two different hash algorithms. The local processing cost of creating a Bloom filter is less than the local processing cost of performing a semijoin. Also the cost of transmitting a Bloom filter is much less than that of transmitting a semijoin attribute.

A Bloom filter is basically a bit vector generated by using a hashing function on an attribute. The filter can be viewed as a compressed representation of the attribute values on which it has been generated. A Bloom filter provides a probabilistic way to determine if an element is a member of a given set. A Bloom filter consists of a bit vector of fixed size. The attributes values are hashed using a hashing function and the corresponding bit values are set in the bit vector. Some values might hash onto the same bit location. In this case the bit is set only once. Consider the following example. The relation **Crew** has three attributes, *Name*, *Age*, and *Salary*. The Bloom filter for attribute *Age* is represented by bit vector $\mathbf{B_T}$. The Bloom filter for Age is as follows. The filter size is taken to be

| Name | Age | Salary |
|---|---|---|
| Tuvoke | 45 | 62 |
| William | 13 | 45 |
| Catherine | 37 | 67 |
| Balana | 29 | 89 |
| Kess | 4 | 50 |

Figure 2.1 Sample relation table: CREW

10 and the hash function is assumed to be Ceil(Age /10). Generally a hashing function is chosen so as to minimize the collisions. Hashing on AGE gives the following bit vector:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Figure 2.2 Bloom Filter Vector on Attribute Age

To reduce errors due to collisions two approachs can be taken. Multiple bit vectors are created on the same attribute using different hashing functions or a single bit vector is chosen of a larger size and the same value is hashed using multiple hash functions. To test the membership of a element in the Bloom filter the element is hashed on each of the hashing functions involved. If all address bits produced by the hashing functions are set then that element is accepted to be present in the filter. If even one of the bits is not set, then that element does not belong to the set of values and is rejected. There is a chance that an element might be accepted if all the bit values set were due to a collision. In [Mul90] it is shown that this error, *Pe*, is equal to:

$$Pe = [\ 1 - (1 - 1/N)^{tk}\ ]^{t}$$

where

*Pe* : Probability of error.

*N* : The Size of the Bit vector.

*t* : The number of transformation or hashing functions.

*k* : number of element validly in the set.

In [Gre82] the design of Bloom filters for differential file access is discussed. In a transaction processing system, huge numbers of transactions might be performed every second. If the main database tables are updated for every transaction the performance of the database system comes down as the database files are locked during every update. A solution to this problem is to use a differential file which will record all the transactions until they can be batch processed periodically. The disadvantage of this method is that other users trying to access the information from the main database in between the file updates might not see the most up-to-date copy of the database. To avoid this, all the requests for information have to first look up the differential file and if the information is not present there then go to the main database. The use of Bloom filters to represent the differential file reduces the time required to access the most current information and also the information retrieved is always the most current. In [Gre82] a simulation study of designing the Bloom filters so that the error rates are minimized is presented. The parameters studied are the size of the Bloom filter and the number of transformations applied on the filter. This paper presents interesting insights into the use of Bloom filters and their performance. In [Mul83] a further discussion on the above topic is presented. Instead of the simulation approach used in [Gre82] this paper presents an analytical formula to study the performance of Bloom filters. In [Mul93] the use of Bloom filters as better estimators of intermediate results due to a join operation is discussed. Bloom filters are used to estimate the size of a join operation between two relations. Their estimation method is based on partial Bloom filters. The use of Bloom filters in optimal semijoins for distributed database systems is discussed in [Mul90]. The use of Bloom filters in the design of parallel database hardware is discussed in [Coo90]. The filters give improved estimates of the number of false drops. False drops are due to the occurrence of collisions while constructing a Bloom filter. Here the case is explored where the relations are stored as bit vectors on each of the individual columns and all the operations are carried out on the bit vectors instead of on the original relations. The operations are carried out by specialized hardware and since the attributes are represented as individual bit vectors the operations can be carried out in parallel. In [VG84] various algorithms for relational and set operations based on hashing are presented. Specifically they explore the hashing methods in three areas: multiple key comparisons, problem partitioning and filter techniques. The application of hash filters in two-operand operations is discussed. An example of the use of hash filters in performing a join is presented. Though this paper does not mention the use of hash filters for a semijoin operation, they have presented a comparison of multiple hash filters and single filters with multiple hash functions for various filter densities. They conclude that for filter densities of 0.5 or higher one filter with minimum density is best. For low densities several serial filters are better. In [OV91] algorithms for computing joins and semijoins of relations in a multiprocessor database machine are presented and analyzed. They explore algorithms for both joins and semijoins using hashed bit arrays. They have given formulas for computing the execution time for different algorithms and conclude that different algorithms perform better depending upon a number of parameters like the relation sizes, selectivity and soon. Hence the best algorithm should be chosen on-the-fly while performing the operation after comparing the results obtained by applying the given formulas.

Bloom filters are used in semijoin processing in the following manner. A Bloom filter is constructed at one site for the attribute on which the semijoin is to be performed. This Bloom filter is transmitted to the site of the other relation which is to be reduced. The attribute values are hashed and checked in the Bloom filter for membership as explained above. Those which fail the membership test cannot possible join with a value at the original site and are ignored. Those passing the membership test are transmitted to the query site. The algorithm proposed by [Mul90] sends a sequence of small but optimally information dense Bloom filters from the master site to the apprentice site. For the sake of simplicity the algorithm is explained for a case where only two sites are involved. The algorithm is as follows:

1. Site A prepares a one transform filter and sends it to site B. A one transform filter is built by using only a single hash function on the attribute involved.
2. Site B computes the fraction of bits set and then:

   — Builds the reduced Relation Rb.

   — Calculates the fraction of tuples accepted *faccept*.

   — Calculates the Reduction which is :

   $(1-\text{Faccept}) \times \text{Nb} \times \text{size(Rb)}$

   where (1–Faccept) gives the fraction of the tuples rejected, Nb is the total number of tuples in relation Rb and size(Rb) gives the size of each tuple in relation Rb.
3. If the reduction is greater than the filter size, then Site A is alerted and the algorithm iterates from the beginning. Otherwise it sends the remaining

tuples of Rb to Site A.

The Bloom filter helps to identify ineffective semijoins quickly. Instead of sending a big multitransform filter it sends a number of small one transform filters. The space requirements of one multitransform Bloom filter and multiple single transform Bloom filters is the same for the same error rate. Hence it is more useful to use multiple single transform filters. The optimal size of a Bloom filter is shown in [Blo70] to be

$$N = tk/ \ln(2)$$

where t is the number of filters and k is the number of distinct values in the attribute. It is possible to estimate the number of tuples that will be rejected by using the filter sent. Using this information it can be determined whether or not the semijoin should proceed or should be rejected. The Bloom filters can be built while performing the initial local processing of single operand operations and hence do not pose a very high local processing cost.

Mullin [Mul90] presents a series of methods to estimate the size of a relational join operation. Here a Bloom filter is represented as set of bits along with the size of the filter and the number of bits set. In some cases, to reduce the size of the filter only a partial filter is stored. In such cases the size of the partial filter, along with the size of the original filter, is stored.

In [Mul93] the authors discuss the use of Bloom filters for performing joins and evaluate their performance in an actual distributed database environment. Bloom filters are used to perform the Bloomjoins to filter out tuples that have no matching tuples in a join. The Bloomjoin algorithm presented works as follows

Let S and T be the relations to be joined. S.a and T.b represent the attribute over which these two relation will be joined. T′ represents the relation produced by reducing T.

1. Generate a Bloom filter, Bfs, from relation S. The Bloom filter is generated by scanning S and hashing each value of column S.a to a particular bit in the bit vector and setting that bit to "1".
2. Send Bfs to site 2.
3. Scan relation T at site 2, hashing the values of T.b using the same hash function as in step (1). If the bit hashed to is "1", then stored that tuple into T′.
4. Send T′ to relation site 1. At site 1, join T′ to S and return the result to the user.

The main disadvantage of Bloom filters is that they cannot be updated after a deletion has occurred in the original relation from which the Bloom filters were created. In this case either the inaccuracy of Bloom filters has to be accepted or the whole filter has to be recreated.

# Chapter 3
# The Algorithms

## 3.2 Introduction

In this chapter we present the algorithms in detail. Much of the material is transcribed from [MBK96, MBB95]. First we state the assumptions we have made about the query processing environment and then we define the common terms used in the algorithms. Finally the algorithms W, Dynamic W, W using Bloom filters and Dynamic W using Bloom filters are presented. Algorithm W is a static strategy [MBB95]. It uses the concepts of marginal profit and gainful semijoins to construct cost effective reducers. A comparison of this algorithm with AHY is done in [MBB95]. Algorithm DW is the purely dynamic version of Algorithm W. Dynamic algorithms have the advantage that they can base their calculations on the most recent values and avoid error propagation due to inaccurate estimation. Algorithm Bloom+W is a static heuristic which uses Bloom filters to get better estimates of the costs and benefits when constructing the schedules. Algorithm Bloom+DW is a dynamic heuristic which also uses Bloom filters. Bloom+DW has the added advantage that it builds filters dynamically and hence has a more accurate representation of the attribute information.

## 3.3 Assumptions and Definitions

A distributed database query processing environment is assumed. The relations are stored at different sites and there is no fragmentation or data replication. These issues are assumed to be dealt with before the query processing stage. An SPJ query model is considered where all initial local processing has already been carried out. The attribute values are considered to be uniformly distributed and are assumed to be independent of each other. The Bloom filters are single function filters where only one hashing function is used to generate the filters. The objective of the query processing is to reduce the total cost which can be expressed in terms of the amount of data transferred over the network. The local processing costs are considered to be insignificant and are ignored.

We consider m relations with n join attributes. The following values are defined.

1. $\mathbf{R_i}$: Relation number i.
2. $\mathbf{S(R_i)}$ : Size of relation $R_i$ in tuples.
3. $\mathbf{d_{ij}}$ : The projection of $R_i$ over attribute j.
4. $\rho(\mathbf{d_{ij}})$ : The selectivity of attribute $d_{ij}$.

5. **D($d_{ij}$)** : The domain of attribute $d_{ij}$ ( the set of all possible values in attribute $d_{ij}$.)

6. | **$d_{ij}$**| : Cardinality of attribute $d_{ij}$ (the number of distinct values in attribute $d_{ij}$).

7. $d_{ij} \ltimes d_{kj}$ : The semijoin between relation *i* and relation *k* on the jth attribute.

The selectivity of an attribute is an estimate of the reduction power of that attribute when applied to another attribute. For different algorithms selectivity is calculated in a different manner [MBB95].

1. For Static heuristics :

$$\rho(d_{ij}) = \frac{|d_{ij}|}{|D(d_{ij})|}$$

2. For Dynamic heuristics :

   The selectivity is calculated in the same manner as above. The only difference is that it is calculated at run time from the actual values.

3. For Bloom filter heuristics :

$$\rho(d_{xj}) = \frac{|h(d_{xj}) \bigcap h(d_{yj})|}{|h(d_{yj})|}$$

   where h($d_{ij}$) is the Bloom filter for $d_{ij}$ and ∩ represents the logical *and* operation on the two filters. That is, the selectivity is estimated by first calculating the cardinality of the bit intersection of the two filters involved and then dividing by the cardinality of the original filter.

A semijoin between the attributes of two relations is represented as

$$d_{ij} \rtimes d_{kj}$$

For the semijoin following terms are defined

1. **Cost** : The cost of semijoin is the cost of sending attribute $d_{ij}$ to the site of $R_k$. That is,

   $C(d_{ij} \rtimes d_{kj}) = C_0 + S(d_{ij}) \times C_1$

   where $C_0$ is the start-up cost and $C_1$ is some positive constant. The formula basically assumes that the cost of sending data over the communications line is linearly dependent on the amount of data sent. In our work, for simplicity we assume that the cost is same as the amount of data to be sent.

2. **Benefit**: The benefit of the semijoin is the amount of reduction in relation k, due to the semijoin. That is,

   $B(d_{ij} \rtimes d_{kj}) = S(R_k) - (S(R_k) \times \rho(d_{ij}))$

3. **Profit**: The profit of the semijoin is its benefit minus the cost.

   $P(d_{ij} \rtimes d_{kj}) = B(d_{ij} \rtimes d_{kj}) - C(d_{ij} \rtimes d_{kj})$

4. **Reducer**: A reducer for a relation is a sequence of semijoin operations which has a very high selectivity. The reducers are constructed for each join attribute after performing a cost/benefit analysis on the sequence of semijoins

involved. A reducer is denoted by $d_{mj}^*$. The reducer can be used to reduce a relation and the reduction effect depends on the selectivity of the reducer. The selectivity of the reducer for different cases is defined below

a. Static heuristics:

- 1 in case of $R_m$ since $d_{mj}^*$ cannot be used to reduce the relation $R_m$.
- the product of the selectivities of all attributes which come after $d_{ij}$ in the sequence otherwise, since $R_i$ has already been reduced by all other attributes before it in the sequence.
- The selectivity of the reducer with respect to a relation not in the sequence, but which has a common-join attribute is simply the product of all the selectivities of the attributes in the sequence.

The selectivities of attribute $d_{mj}^*$ for different cases is given below

$$\rho_{R_i}(d_{mj}^*) = \begin{cases} 1 & i = m \\ \prod_{x=i+1}^{m} \rho(d_{xj}). & i < m \\ \prod_{x=a}^{m} \rho(d_{xj}). & otherwise \end{cases}$$

b. Dynamic heuristics : In the case of a **dynamic** heuristic, the selectivity of $d_{mj}^*$ with respect to any $R_i$ in the construction sequence is defined as $|d_{mj}^*| / |d_{xj}|$. The selectivity of $d_{mj}^*$ w.r.t any relation not in the sequence is $|d_{mj}^*| / |D(d_{mj}^*)|$.

In summary, given a sequence of semijoins which dynamically constructs the reducer $d_{mj}^*$ the selectivity of the reducer w.r.t $R_i$ is defined as follows.

$$\rho_{R_i}\left(d_{mj}^*\right) = \begin{cases} 1 & i = m \\ \frac{|d_{mj}^*|}{|d_{ij}|} & i \in a..m, i! = m \\ \frac{|d_{mj}^*|}{|D(d_{ij})|} & otherwise \end{cases}$$

c. Bloom Filter heuristics. For this case the selectivity of the reducer $R_i$ is defined as follows:

$$\rho_{R_i}\left(d_{mj}^*\right) = \begin{cases} 1 & i = m \\ \frac{|h(d_{mj}^*)|}{|h(d_{ij})|} & i \in a..m, i! = m \\ \frac{|h(d_{mj}^*)\cap h(d_{ij})|}{|h(d_{ij})|} & otherwise \end{cases}$$

Once the reducers are constructed they can be applied to get further reduction in the relations. The decision whether or not to use the reducer is made after doing a cost benefit analysis of the semijoins involved. This is carried out as follows:

The cost of using the reducer is the size of the reducer itself.

$$c\left(d_{mj}^* \rtimes R_i\right) = S\left(d_{mj}^*\right)$$

The benefit is the reduction in the size of relation $R_i$. This is estimated as

$$B\left(d_{mj}^* \rtimes R_i\right) = S'(R_i) \times \left(1 - \rho_{R_i}\left(d_{mj}^*\right)\right)$$

5. **Marginal Profit**: Marginal profit tries to look at the overall profit of a semijoin. Some semijoins might not be immediately profitable but might

increase the profit of subsequent semijoins. Consider the semijoin $d_{xj}^* \ltimes d_{yj}$. The marginal profit is defined as the difference in profit obtained by using $d_{yj}^*$, rather than $d_{xj}^*$ as the reducer. It is summed over all relations with a common join attribute. The marginal profit is used to see the effects of the current semijoin down the line [MBB95]. That is, it provides a look ahead for the current semijoin and looks at the global picture of the effects of the semijoin before a decision can be made as to whether to accept or reject the semijoin. The marginal profit of a semijoin $d_{xj}^* \ltimes d_{yj}$ with respect to a relation is calculated according to the following rules. The marginal profit of $d_{yj}^* \ltimes R_y$ is zero as there is no marginal profit; an attribute can not be used to reduce the relation to which it belongs. Also the marginal profit is zero if the cost of the semijoin is greater than any potential benefit. In the case where there is no profit in using $d_{xj}^*$ but there is profit in using $d_{yj}^*$ then the marginal profit is the same as the profit of $d_{yj}^* \ltimes R_i$. Therefore the marginal profit with respect to $R_i$ is

$$MP_{R_i}\left(d_{xj}^* \ltimes d_{yj}\right) = \begin{cases} 0, \ if\ i = y \\ 0, \ if\ P\left(d_{xj}^* \ltimes R_i\right) \leq 0 \ and \quad P\left(d_{yj}^* \ltimes R_i\right) \leq 0 \\ P\left(d_{yj}^* \ltimes R_i\right), \ if\ P\left(d_{xj}^* \ltimes R_i\right) \leq 0 \\ P\left(d_{yj}^* \ltimes R_i\right), \ if\ i = x \\ P\left(d_{yj}^* \ltimes R_i\right) - P\left(d_{xj}^* \ltimes R_i\right), \quad otherwise \end{cases}$$

The marginal profit is than summed over all positive marginal profit to get the total:

$$MP\left(d_{xj}^{*} \rtimes d_{yj}\right) = \sum_{i=1}^{m} MP_{R_i}\left(d_{xj}^{*} \rtimes d_{yj}\right) \quad s.t \quad MP_{R_i} > 0$$

6. **Gain** : Gain is defined as the sum of the profit and marginal profit of a semijoin:

   $$G\left(d_{xj}^{*} \rtimes d_{yj}\right) = P\left(d_{xj}^{*} \rtimes d_{yj}\right) + MP\left(d_{xj}^{*} \rtimes d_{yj}\right)$$

   A semijoin is said to be cost-effective if the gain is positive.

7. **Bloom Filter**: A Bloom filter is a hashed representation of the attribute values on which it is constructed. It's an array of bits and is constructed as follows:

   a. Set all the bits in the bit vector to Zero.

   b. Read the value of the attribute and hash the value using a hashing function to produce a bit address.

   c. Set the addressed bit to 1.

   The use of Bloom filters in the cost/benefit analysis of semijoins is carried out as follows. Consider the semijoin $d_{aj} \rtimes d_{bj}$. The join attribute here is j. This semijoin should be executed only if the cost of transmitting $R_a[j]$ to the site of $R_b$ is less than the benefit.

   a. Using some hash function, construct filters for $R_a[j]$ and $R_b[j]$.

   b. Perform a bit-wise "and" of the two filters at some site. The number of bits set is an estimate of the number of attribute values held in common. Suppose this number is X.

c. Let $|R_b(j)|$ denote the number of attribute values in the projection of $R_b$ over the common-attribute.

d. The reduction in relation is estimated as

$$R_b' = |R_b| * \frac{X}{|R_b[j]|}$$

## 3.4 Algorithm W

Algorithm W [MBB95] uses the concept of Marginal profit and gainful semijoins to derive cost effective reducers. The algorithm follows a three phased approach. In the first phase the schedules for the construction of the reducers are established. Each attribute is considered separately. The outline of the heuristic is as follows.

1. **Phase 1: Establish the schedules for the construction of the reducers.** For each join-attribute establish a reducer construction schedule in parallel. The semijoins are considered in the order

$$((d_{aj} \rtimes d_{bj}) \rtimes d_{cj}) \ldots \rtimes d_{mj}$$

such that

$$S(d_{aj}) \leq S(d_{bj}) \leq S(d_{cj}) \ldots \leq S(d_{mj})$$

A semijoin, $d_{xj}^* \rtimes d_{yj}$, is appended to the schedule for $d_{mj}^*$ if

a. $P\left(d_{xj}^{*} \rtimes d_{yj}\right) > 0 \;\; and \;\; MP\left(d_{xj}^{*} \rtimes d_{yj}\right) > 0$ or

b. $P\left(d_{xj}^{*} \rtimes d_{yj}\right) < 0 \;\; but \;\; G\left(d_{xj}^{*} \rtimes d_{yj}\right) > 0$

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

   In this phase the reduction effects of the reducers, from smallest to largest, on all applicable relations are considered. Profitable semijoins are appended to the final schedules. Some reducers may not be applied to any relation as the cost of shipping them may be greater than any potential benefits. In this case the semijoin sequence for this reducer is reviewed and any profitable semijoins are appended to the final schedule. This ensures that profitable semijoins will be applied to the schedule but the unnecessary ones will not be executed.

3. **Phase 3: Execute the schedule.**

   During this phase the reducers are first constructed in parallel. Then the reducers are shipped to the designated sites in parallel. Finally the reduced relations are transferred to the final assembly site, in parallel.

### 3.4.1 The W heuristic in detail

Consider m relations and n join-attributes. Let J be the set of all join attributes.

1. Step1: In this step a cost/benefit analysis is carried out for every semijoin. Each attribute is considered independently.

   a. Sort the attributes according to the attribute size, such that

   $S(d_{aj}) \leq S(d_{bj}) \leq \ldots \leq S(d_{mj})$

b. Consider each semijoin in the sequence. Perform an cost/benefit analysis on it and append it to the schedule if

   - $P\left(d_{xj}^{*} \rtimes d_{yj}\right) > 0 \;\; and \;\; MP\left(d_{xj}^{*} \rtimes d_{yj}\right) > 0$ or
   - $P\left(d_{xj}^{*} \rtimes d_{yj}\right) < 0 \;\; but \;\; G\left(d_{xj}^{*} \rtimes d_{yj}\right) > 0$

   If $d_{aj} \rtimes d_{bj}$ is appended to the schedule then $d_{bj}^{*} \rtimes d_{cj}$ is evaluated next otherwise $d_{aj} \rtimes d_{cj}$ is evaluated next. This process is repeated until all the semijoins in the sequence have been evaluated. The attribute reduced in the final semijoin appended is the reducer.

2. Step 2: In this step each reducer is considered for application to the relations present in order to see if further reductions are possible. The reducers are sorted according to their size in ascending order. A cost benefit analysis is done to check if the reducer is profitable. Profitable semijoins are appended to the schedule.
3. Step 3: In this step those reducers not found to be profitable with respect to all the relations in the above step are reviewed. Any profitable semijoins in the sequence are appended to the final schedule. The marginal profit is not considered in this step.
4. Step 4: In this final step the schedule is executed and the reduced relations are shipped to the query site for joining. The reducers are constructed in parallel, they are applied in parallel and the final shipping is done in parallel, ensuring that the response time is reasonable even if it is not minimal.

## 3.4.2 Algorithm W Example.

Consider the following example having three relations and up to two join attributes each.

| Relation | $S(R_i)$ | $S(d_{i1})$ | $\rho(d_{i1})$ | $S(d_{i2})$ | $\rho(d_{i2})$ |
|---|---|---|---|---|---|
| R1 | 1000 | 400 | 0.4 | 100 | 0.2 |
| R2 | 2000 | 400 | 0.4 | 450 | 0.9 |
| R3 | 3000 | 900 | 0.9 | - | - |

Figure 3.1 Database Query Statistics

As there are two attributes each attribute is considered independently during the construction of the schedule. These schedules can be constructed in parallel.

**Attribute $d_{i1}$:** The first step is to sort the attributes in ascending order of size. This gives us the sequence

$$d_{11} \ltimes d_{21} \ltimes d_{31}$$

Consider the semijoin $d_{11} \ltimes d_{21}$

The cost is the size of the attribute, 400 and the benefit is 1200. The marginal profit for $R_3$ is calculated as

$$MP_{R3} = 3000(0.4 - 0.4 \text{ X } 0.4) + 400 - 160$$

In this case both the profit and marginal profit are greater than zero; hence the semijoin is added to the schedule. Next $d_{21}^{*} \ltimes d_{31}$ is examined: the cost is 160 units, the benefit is 2520 units. The marginal profit of the semijoin with respect to $R_1$ is

$$MP_{Ri} = 1000\ (\ 0.4\ -\ 0.4\ X\ 0.9\ )\ +160\ -144$$

Again both the profit and the marginal profit are positive so the semijoin is added to the schedule. The reducer is called $d_{31}^*$. It is constructed by the following schedule:

$$d_{11} \xrightarrow{400} d_{21} \xrightarrow{160} d_{31}$$

The schedule indicates that attribute $d_{11}$ is sent to the site of relation $R_2$ at a cost of 400 units. The attribute $d_{21}$ of relation $R_2$ is reduced and sent to the site of relation $R_3$ at the cost of 160 units

**Attribute $d_{i2}$** : Only one semijoin needs to be considered, $d_{12} \rtimes d_{22}$. The cost of the semijoin is 100 units and the benefit is 1600. The marginal profit with respect to $R_1$ is

$$MP_{Ri}\ =\ 1000\ (1\ -\ 0.9)\ -\ 90.$$

Hence the schedule for constructing $d_{22}^*$ is

$$d_{12} \xrightarrow{100} d_{22}$$

The next phase is where the reduction effects of each reducers are considered. The reducers are first sorted according to there sizes. The reduction $d_{22}^* \rightarrow R_1$ is considered first. The cost is 90 units and the benefit is 100 and therefore this semijoin is appended to the schedule. The reducer $d_{31}^*$ is considered next. For the purpose of calculations the updated sizes of relations are considered and these relation sizes will be affected by the application of previous reducers. These updated sizes are as follows: $S(R_1) = 900$; $S(R_2) = 160$ and $S(R_3) = 480$.
We next consider the semijoin $d_{31}^* \rightarrow R_1$. The cost is 144, the benefit is 576. The semijoin is profitable and the semijoin is appended to the schedule. The final schedule for execution is as follows.

$$\begin{tikzcd}
d_{11} \arrow[r, "400"] & d_{21} \arrow[r, "160"] & d_{31} & d_{31}^* \arrow[r, "144"] & R_1 & R_1 \arrow[r, "324"] & QS \\
d_{12} \arrow[r, "100"] & d_{22} & & d_{22}^* \arrow[r, "90"] & R_1 & R_2 \arrow[r, "160"] & QS \\
& & & & & R_3 \arrow[r, "480"] & QS
\end{tikzcd}$$

The total cost for this schedule is 1858 units: the response time is 1184 units.

## 3.5 Algorithm DW

Algorithm DW is a purely dynamic version of Algorithm W. The basic strategy to build the reducers remains the same. In DW the reducers are built and a decision is made to construct the smallest profitable reducer first. In the next phase the semijoins to build the reducer are actually carried out dynamically. Each semijoin is checked to see if it is gainful. If not it is dropped from the sequence. Once a reducer has been constructed the whole process is repeated until all the reducers are constructed and used. This algorithm is described in detail below

1. **Estimate the size of the reducers**

   For each join attribute in the query estimate the size of the reducer. The attributes are sorted in ascending order of attribute size. The cardinality of the reducer $d_{xj}^*$ is estimated as follows

   $$\left|d_{xj}^*\right| = \rho(d_{aj}) * \rho(d_{bj}) * \ldots * \rho(d_{nj}) * S(d_{xj})$$

2. **Select a reducer to construct and use**

   Choose the smallest profitable reducer.

3. **Construct the reducer**

   In this phase the reducer is built by executing one semijoin at a time. As soon as the semijoin is executed the database statistics are updated. The cost/benefit analysis is carried out for the remaining semjoins using the updated statistics.

4. **Use the reducer**

   In this step the reducer is checked to see if it is profitable for application with any of the relations involved in the query. Profitable semijoins are executed.

5. **Repeat the process**

   The current reducer is marked as done and the whole process is repeated for the remaining join attributes.

## 3.6 Algorithm Bloom+W

This is also a static heuristic but it uses the Bloom filters to obtain better estimates of the cost and benefit of a semijoin. A Bloom filter is constructed for every joining attribute in the query. The length of the Bloom filter is made proportional to the domain size of the attribute. In this algorithm the basic strategy remains the same. The manner in which the selectivities are calculated is different. The algorithm is outlined below:

1. Build the reducer for each join attribute

   a. Sort the attributes in ascending order of attribute size.

   b. Estimate the cost and benefit for every semijoin in this order. For the semijoin $d_{aj} \rtimes d_{bj}$, the cost is $S(d_{aj})$ and the benefit is estimated as $S(R_b) * \left(1 - \frac{|h(d_{aj}) \cap h(d_{bj})|}{|h(d_{bj})|}\right)$

   c. The marginal profit is calculated as before. The semijoin is added to the schedule if it is gainful.

   d. If the semijoin is added then $d_{bj}^{*} \rtimes d_{cj}$ is considered next using $h(d_{aj}) \cap h(d_{bj})$ as an approximation for $h(d_{bj}^{*})$ and $|h(d_{aj}) \cap h(d_{bj})|$ as an estimate of $|d_{bj}|$.

e. Otherwise the semijoin $d_{aj} \ltimes d_{cj}$ is considered next.

f. Repeat the process until all the semijoins in the sequence have been considered.

2. Apply the profitable reducers to reduce the relations. Once the reducers are built they are sorted in ascending order of size. A cost benefit analysis is done for every relation. For the semijoin $d_{mj}^{*} \ltimes R_i$ the cost is $S(d_{mj}^{*})$ and the benefit is

$$S(R_i) * \left(1 - \frac{|h(d_{mj}^{*})|}{|h(d_{ij}^{*})|}\right)$$

if $d_{ij}$ is used in the construction of the reducer. Else it is calculated as

$$S(R_i) * \left(1 - \frac{|h(d_{mj}^{*})|}{|D(d_{ij})|}\right)$$

3. Certain reducers will not be profitable for application to the relations. The semijoin sequence in such reducers is re-examined and all the profitable semijoins are added to the schedule.

4. Execute the Schedule: the schedules are executed and the reduced relations are shipped to the query site where the answer is assembled.

## 3.7 Algorithm Bloom+DW

This algorithm combines the previous two algorithms to obtain the benefit of both the algorithms. Bloom filters are used to get accurate estimates of the projection sizes. As the filters are constructed and updated dynamically the most current information is used and the filters always represent the current attribute values. Here Bloom filters are used as accurate estimators and not as low cost reducers. The heuristic is as follows:

1. Sort the attributes in ascending order of size. Select the attribute whose reducer has the smallest size and is profitable. The size of the reducer $d_{xj}^*$ is estimated as follows
   $\left|d_{xj}^*\right| = \rho(d_{aj}) * \rho\left(d_{bj}\right) * \ldots * \rho(d_{nj}) * S(d_{xj})$
2. Construct the reducer for join attribute j:
   a. Consider semijoin $d_{aj} \rtimes d_{bj}$. The cost for this semijoin is calculated as $S(d_{aj})$. The benefit is
      $S(R_b) * \left(1 - \frac{|h(d_{aj}) \cap h(d_{bj})|}{|h(d_{bj})|}\right)$
      The marginal profit is calculated in the usual manner.
   b. The semijoin is added to the schedule if it is gainful.
   c. Execute each semijoin in the reducer dynamically. After the execution of the semijoin $d_{aj} \rtimes d_{bj}$, $h(d_{bj}^*)$ is reconstructed dynamically using the reduced attribute $d_{bj}$. All the other attributes of relation b are rehashed to construct their Bloom filters.
   d. If the semijoin is executed then consider the semijoin $d_{bj}^* \rtimes d_{cj}$ else consider the semijoin $d_{aj} \rtimes d_{cj}$.

3. Apply the reducer by executing all profitable semijoins $d_{mj}^* \rtimes R_i$. The cost for the semijoin is $S(d_{mj}^*)$. The benefit is
$$S(R_i) * \left(1 - \frac{|d_{mj}^*|}{|h(d_{ij})|}\right)$$
if $R_i$ was used in the construction of the reducer; otherwise the benefit is estimated as
$$S(R_i) * \left(1 - \frac{|h(d_{ij}) \cap h(d_{mj}^*)|}{|h(d_{ij})|}\right)$$
In this chapter we have described five algorithms.

1. **Algorithm W**, which is a static heuristic, uses the concept of marginal profit and gainful semijoins to construct profitable reducers.
2. **Algorithm AHY** is used as a bench mark algorithm. The total cost version of this algorithm is used to perform the bench marking.
3. **DW** is a dynamic algorithm which monitors the execution of semijoins and minimizes the error due to propagation of wrong estimates.
4. **Bloom+W** is based on the use of Bloom filters. Bloom filters are used to obtain a better estimates for calculation.
5. **Bloom+DW** combines the dynamic strategy with better estimation techniques.

In the next chapter we describe how the heuristics were evaluated and present the results of our experiments.

# Chapter 4
# Evaluation

## 4.1 Introduction

To study the performance of the proposed algorithms a large number of experiments were carried out. Each algorithm was executed on a large number of synthetically generated test databases and the total cost in each case was calculated and compared. The objective of these experiments was to test the total cost of the proposed algorithms against the total cost of the AHY algorithm. The performance study was carried out for a wide range of SPJ queries and different data distributions. In this chapter, we will present the methodology for conducting the experiments, present the experimental results and discuss the results.

## 4.2 Methodology

The performance evaluation of the algorithms is done by executing the algorithms using a large number of synthetically generated test queries and databases. It is assumed that the queries have undergone the initial local processing. As a result, all the unary operations that could be executed locally are assumed to have been done. The remaining query can be represented using parameters like the size of each relation, the number of attributes in each relation and for each attribute the number of distinct values, the size of each attribute, the domain size of each attribute and its selectivity.

| Rel- | Attribute 1 | | Attribute 2 | | Attribute 3 | |
|---|---|---|---|---|---|---|
| Size | Size | Selectivity | Size | Selectivity | Size | Selectivity |
| 3000 | 600 | 0.6 | 250 | 0.5 | 200 | 0.2 |
| 4000 | 800 | 0.8 | 500 | 1.0 | 400 | 0.4 |

Figure 4.1 Format of a typical query

In the above table the query has two relations and each relation has three attributes. The first column indicates the size of a relation in terms of number of tuples. The first relation has 3000 tuples and the second has 4000 tuples. The remaining three columns have information regarding the relation attribute. For each attribute two values are indicated. The first sub—column indicates the number of distinct values in the attribute and the second sub—column indicates the selectivity of the attribute. The selectivity is calculated by dividing the size of a column by its domain. The domain of each attribute is indicated in a separate file called the domain table. By varying these parameters a wide range of queries can be generated. In the performance evaluation the query parameters were varied as follows:

1. Each query has between 3 and 6 relations and each relation has between 2 and 4 attributes. Overall, this gives 12 different types of test queries.
2. The domain size of each attribute is varied between 500 to 1500. For the sake of simplicity, all the domains are integer based.
3. The selectivity of each attribute is selected randomly to be between 0.5 and 1.0. The lower bound on selectivity was fixed at 0.5 so that the number of queries producing the NULL result is small.
4. The size of each relation was randomly selected to be from between 800 and 6000 tuples.
5. The number of join attributes is varied to get three different levels of connectivity: 50%, 75% and 100%. The attributes are chosen in a manner that the queries remain connected.

The test queries are generated in the following manner. First, the number of relations and the maximum number of attributes is chosen. Then the query generator randomly selects the domain size for each attribute from the range specified. Next the attributes for each relation are chosen so as to maintain connectivity. The connectivity [MBB95] can be specified to be 50%, 75% or 100%. Connectivity is defined as the fraction of the total number of attributes present in the query and is calculated as follows:

$$\frac{\sum_{i=1}^{n} (number\ of\ join\ attributes\ in\ R_i)}{n \times m} \times 100\%$$

where m is the total number of relations and n is the number of attributes in each relation. A query is said to be connected when each relation shares at least one common join attribute with another relation and the query graph is not disconnected. That is, all the relations can be joined. The cardinality for each join attribute is selected randomly. The cardinality is selected so that the selectivity of the attribute is in the range 0.5 to 1.0. Lastly the cardinality of each relation is selected in such a way that it exceeds the cardinality of all its attributes. Once the statistical information is generated for the query, the relations are constructed and populated with data according to the selected distribution. When a uniform distribution is desired the actual values for each attribute are uniformly chosen from the set of all values in the domain. The number of attribute values is selected so as to satisfy the desired selectivity. For example, if a selectivity of 0.5 is chosen and the domain size is 1000, then 500 unique values are randomly selected from the set of 1000 values. These values are repeated an equal number of times to fill the relation. In the case of a random distribution the initial selection of values is the same. When the values are repeated each value is repeated a random number of times.

## 4.3 Experimental Results

The results were obtained by carrying out a number of test runs. The test runs can be divided into two distinct groups: those where the data are distributed uniformly and those where the data are distributed randomly. The test runs in both cases are identical except for the data distribution. In each case there are twelve different types of queries. These are obtained by varying the number of relations and the number of attributes. The connectivity for each query is set at 50%, 75%, and 100%. For each connectivity the query statistics are generated and populated with synthetic data. Each of the five algorithms is run using this query. For each algorithm the total cost is calculated and stored. This procedure is repeated 100 times for each type of query. Hence, in each distribution we have three levels of connectivity, for each connectivity level we have twelve different types of queries and for each query type we have 100 different queries. The results obtained for each query type are averaged over the 100 queries. The results obtained for the different algorithms are then compared with the AHY algorithm.

The following tables show the percentage improvement of various algorithms over the AHY algorithm. The actual values of total cost for each algorithm is presented in Appendix A. The three tables are for 100%, 75% and 50 % connectivity. In each table the rows indicate the 12 different query types obtained by varying the number of relations and attributes. The columns indicate the percentage improvement of each of the four algorithms over the AHY algorithm. For each algorithm the performance for Uniform and Random distributions is presented. The total cost for each case is calculated as follows:

1. **AHY**: The total cost is the cost of performing all the semijoins and the cost of sending the reduced relations to the query site.
2. **W**: The total cost is the cost of carrying out the semijoins in each schedule

and the cost of sending the reduced relations to the query site.

3. **DW**: The total cost is the cost of performing the semijoins plus the cost of sending the final reduced relations. We do not consider the cost of extra messages sent as this was found to be very small [Bea95]
4. **Bloom+W**: The total cost is the cost of performing the semijoins plus the cost of transmitting the Bloom filters plus the cost of sending the reduced relations to the query site.
5. **Bloom+DW**: The total cost is the cost of performing the semijoins plus the cost of sending the initial Bloom filters plus the cost of sending updated Bloom filters plus the cost of sending the final reduced relations.

Once the total cost for each algorithm is calculated the percentage improvement is calculated as follows. Let cost(unopti) represent the cost of sending the original relations to the query site. This is the unoptimized cost of sending the relations without any reduction. The improvement of the benchmark algorithm AHY over the unoptimized cost is calculated as follows:

$$Value1 = \frac{cost(unopti) - cost(AHY)}{cost(unopti)} \times 100\%$$

Let the cost of any algorithm be indicated by cost(Algo). Then the improvement of the algorithm over unoptimized cost is calculated as follows:

$$Value2 = \frac{cost(unopti) - cost(Algo)}{cost(unopti)} \times 100\%$$

The improvement of an algorithm over AHY is calculated as follows:

$$Value1 - Value2$$

### 4.3.1 Results for 100 % Connectivity :

Below are the results for the 100% connectivity experiments. Each entry in the table indicates the average improvement for 100 different queries of same type. For example, in the table below the entry 12.88 indicates that the improvement of algorithm W over algorithm AHY for a uniform distribution and query type 3–2 is 12.88%. Note that the entry is averaged over 100 queries in this category. The improvement for each query category is then averaged and is indicated in the bottom most row.

The table below shows results for 100 % connectivity.

| Query Type | W | | DW | | Bloom+W | | Bloom+DW | |
|---|---|---|---|---|---|---|---|---|
| | Uniform | Random | Uniform | Random | Uniform | Random | Uniform | Random |
| 3-2 | 12.88 | 4.97 | 17.03 | 3.34 | 14.74 | 6.48 | 16.64 | 5.05 |
| 3-3 | 17.99 | 11.47 | 22.76 | 11.83 | 17.72 | 11.43 | 22.71 | 9.71 |
| 3-4 | 16.24 | 21.09 | 18.47 | 28.95 | 16.09 | 21.23 | 18.58 | 28.01 |
| 4-2 | 17.71 | 10.68 | 19.83 | 9.88 | 17.93 | 14.32 | 20.50 | 7.09 |
| 4-3 | 18.12 | 30.39 | 20.45 | 40.98 | 18.17 | 31.04 | 20.44 | 39.63 |
| 4-4 | 16.51 | 43.97 | 20.34 | 55.68 | 16.08 | 47.49 | 16.92 | 55.47 |
| 5-2 | 17.47 | 25.47 | 21.34 | 24.85 | 17.36 | 26.54 | 19.88 | 23.08 |
| 5-3 | 19.48 | 48.14 | 22.67 | 61.97 | 19.47 | 49.62 | 19.86 | 61.39 |
| 5-4 | 13.95 | 61.05 | 16.14 | 65.44 | 18.91 | 60.93 | 20.15 | 65.64 |
| 6-2 | 26.93 | 39.70 | 25.37 | 36.24 | 26.93 | 39.23 | 27.91 | 37.40 |
| 6-3 | 14.98 | 59.20 | 15.42 | 64.91 | 14.89 | 60.74 | 15.42 | 67.56 |
| 6-4 | 12.26 | 64.22 | 15.53 | 66.66 | 12.27 | 63.81 | 12.54 | 66.39 |
| Average | 17.04 | 35.03 | 19.61 | 39.23 | 17.55 | 36.07 | 19.24 | 38.82 |

Figure 4.2 Results for 100% Connectivity

### 4.3.2 Results for 75 % Connectivity :

The table below shows the results for 75 % connectivity

| Query type | W |  | DW |  | Bloom+W |  | Bloom+DW |  |
|---|---|---|---|---|---|---|---|---|
|  | Uniform | Random | Uniform | Random | Uniform | Random | Uniform | Random |
| 3-2 | 7.08 | 5.06 | 9.06 | 16.72 | 6.87 | 5.08 | 8.65 | 16.40 |
| 3-3 | 12.34 | 8.02 | 12.87 | 3.38 | 12.37 | 8.18 | 12.97 | 2.65 |
| 3-4 | 16.26 | 14.08 | 24.88 | 15.85 | 18.34 | 15.97 | 24.92 | 15.94 |
| 4-2 | 11.16 | 12.72 | 11.57 | 5.95 | 11.18 | 13.77 | 11.18 | 5.29 |
| 4-3 | 18.04 | 18.72 | 16.46 | 15.28 | 18.04 | 19.59 | 21.82 | 15.09 |
| 4-4 | 18.65 | 23.95 | 21.24 | 31.67 | 18.80 | 26.55 | 21.03 | 33.18 |
| 5-2 | 18.80 | 19.53 | 15.71 | 15.34 | 19.10 | 19.66 | 18.75 | 14.65 |
| 5-3 | 20.60 | 24.70 | 24.08 | 26.21 | 21.30 | 26.05 | 22.50 | 24.77 |
| 5-4 | 19.00 | 44.01 | 22.07 | 48.41 | 19.83 | 46.12 | 22.18 | 48.22 |
| 6-2 | 25.67 | 24.36 | 26.37 | 20.22 | 25.97 | 24.97 | 26.42 | 20.73 |
| 6-3 | 19.59 | 35.23 | 20.97 | 41.47 | 19.58 | 35.98 | 21.73 | 41.86 |
| 6-4 | 26.34 | 49.02 | 28.48 | 51.21 | 26.43 | 48.79 | 28.30 | 54.05 |
| Average | 17.79 | 22.44 | 19.48 | 21.52 | 18.15 | 23.38 | 20.04 | 21.67 |

Figure 4.3 Results for 75 % Connectivity

### 4.3.3 Results for 50% Connectivity :

The table below shows the results for 50 % connectivity.

| Query Type | W | | DW | | Bloom+W | | Bloom+DW | |
|---|---|---|---|---|---|---|---|---|
| | Uniform | Random | Uniform | Random | Uniform | Random | Unifomr | Random |
| 3-2 | 5.94 | 9.05 | 3.92 | 0.55 | 5.95 | 9.05 | 3.56 | 0.56 |
| 3-3 | 9.12 | 4.97 | 9.72 | 5.12 | 10.30 | 4.88 | 9.28 | 4.72 |
| 3-4 | 17.72 | 6.44 | 20.37 | 9.50 | 17.54 | 8.38 | 20.35 | 8.82 |
| 4-2 | 10.72 | 10.70 | 9.48 | 5.80 | 10.58 | 10.54 | 9.20 | 5.51 |
| 4-3 | 12.73 | 8.65 | 13.40 | 1.2 | 13.89 | 8.61 | 13.40 | 3.38 |
| 4-4 | 16.90 | 8.42 | 20.23 | 2.75 | 16.87 | 10.79 | 20.16 | 2.75 |
| 5-2 | 15.57 | 10.62 | 12.84 | 4.11 | 15.57 | 10.49 | 12.85 | 2.33 |
| 5-3 | 21.69 | 10.34 | 25.83 | 2.32 | 22.97 | 13.94 | 26.27 | 1.97 |
| 5-4 | 19.26 | 12.69 | 25.71 | 10.45 | 21.96 | 17.22 | 25.30 | 9.24 |
| 6-2 | 25.60 | 17.14 | 21.19 | 9.60 | 27.05 | 15.58 | 21.80 | 5.72 |
| 6-3 | 18.80 | 15.11 | 17.55 | 12.99 | 19.04 | 21.57 | 17.76 | 11.9 |
| 6-4 | 24.94 | 20.06 | 28.00 | 20.08 | 24.08 | 11.65 | 28.22 | 21.5 |
| Average | 16.58 | 11.18 | 16.70 | 6.09 | 17.15 | 11.65 | 16.72 | 5.02 |

Figure 4.4 Results for 50% Connectivity

## 4.4 Discussion of Results

All strategies showed some improvement over the AHY algorithm in terms of reducing the total cost of processing a query. The following observations can be made based on the results:

- A general observation can be made that the improvements were higher for the random distribution than for the uniform distribution. This is because the attribute dependency is more in case of random distribution than in case of uniform distribution. When an attribute is reduced the effect of reduction of this attribute on the other attributes is more in case of random distribution. This is especially true in cases where the attribute size and the size of the relations are comparable as is the case in our experiments. At lower connectivity, the uniform distribution showed better performance than the random distribution. Also the higher the number of attributes, the greater the improvement when the distribution is random. The above two observations is also due to the same reasons. When the connectivity is lower the effect of reduction of one attribute is not that much as when the connectivity is higher. Also as the number of attributes increases the chances of this effect also increases. Hence random distribution shows better improvements over the uniform distribution.

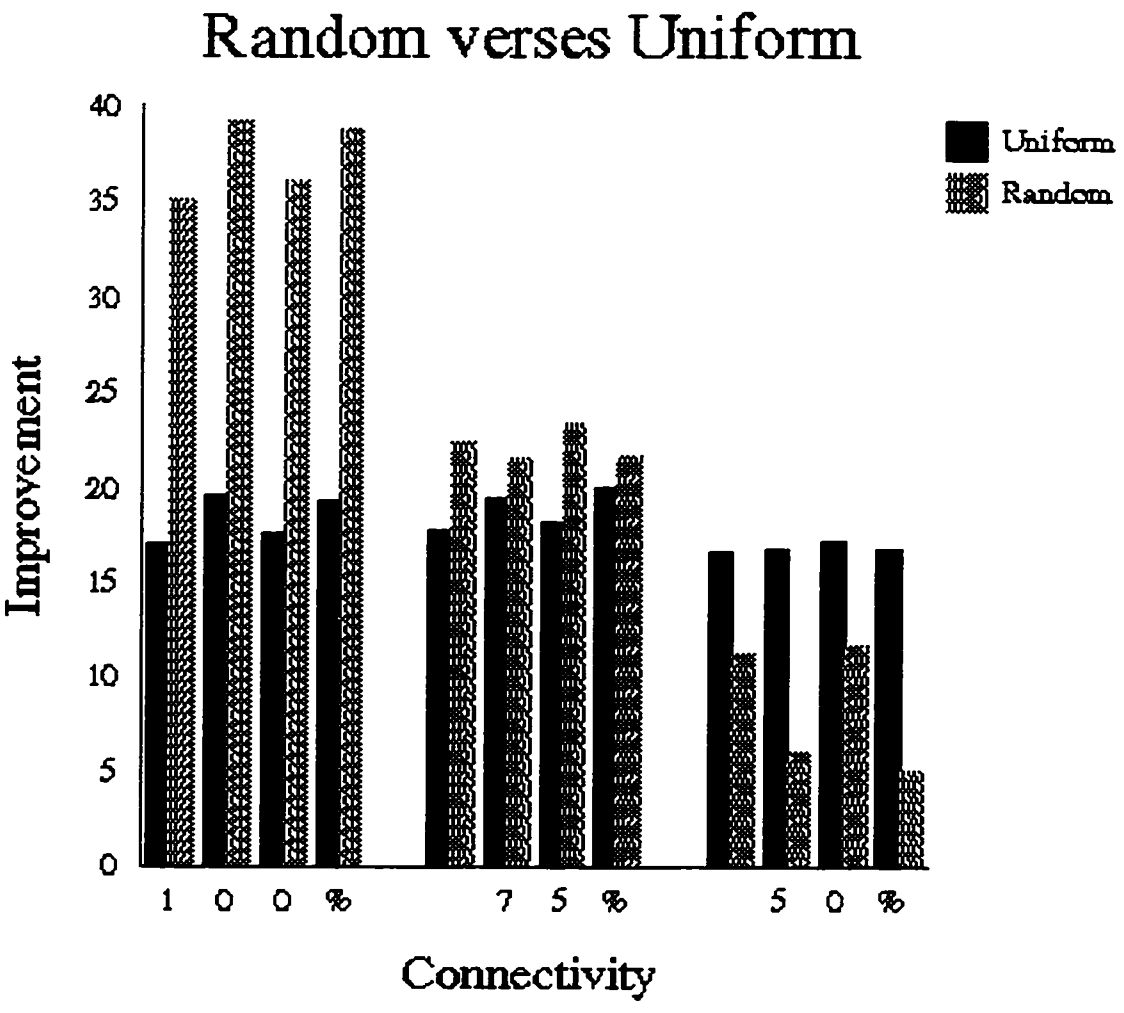


Figure 4.5 Improvement of random and uniform distribution for different connectivity

- The improvement increased as the number of attributes in the query increased and the number of relations in the query increased. When the number of attributes is small, the random distribution gives a poorer performance than the uniform distribution.

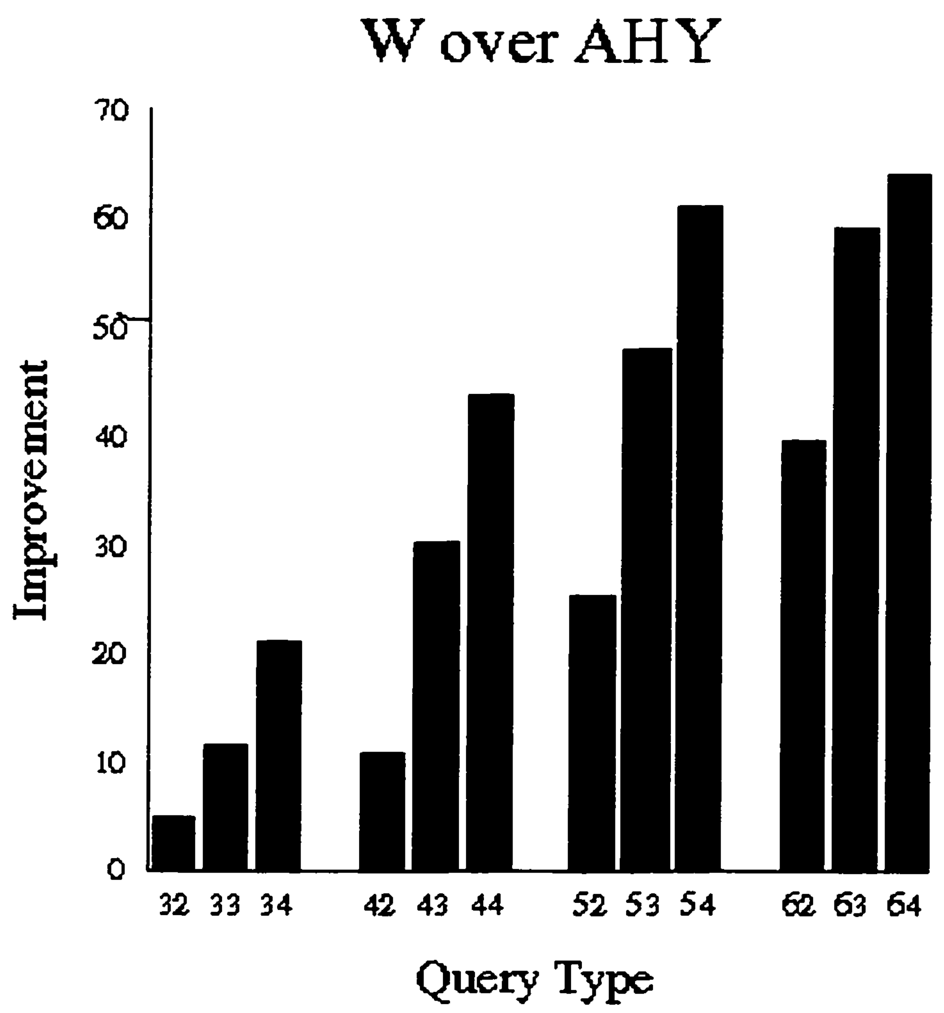


Figure 4.6 Improvement of W over AHY for Random distribution for different query types

- In case of the Dynamic algorithm the improvement was not very significant over the improvement achieved by algorithm W. A small percentage of improvement was shown in almost all the cases over algorithm W.
- There is virtually no difference in the performance of W and Bloom+W. This because the basic strategy used for W and Bloom+W is the same. The semijoins are considered in the same sequence in both the cases. The only

difference is that Bloom+W uses the bloom filters to calculate profitability of a semijoin. It was observed that most of the semijoins which were profitable in case of W were also profitable in case of Bloom+W. Only those semijoins where the profit was very near to zero showed a difference in performance. This is because the effectiveness of Bloom filters is lost after the application of first reducer. The effect of one reduction of attributes is not reflected in the other bloom filters of other attributes as the these filters are not updated.

The average improvement showed by each algorithm is about 3 - 5% over algorithm W. This small improvement is due to a number of factors. Algorithm Bloom W uses static Bloom filters. Once the attributes are reduced the effect of reduction is not reflected in the filters. Hence a margin of error is introduced in this algorithm. Dynamic W considers the semijoins in the same sequence as algorithm W. The difference being dynamic W will execute semijoins only if they are absolutely profitable. Algorithm W might execute some semijoins which are not actually profitable. This happens in those cases where the margin of profit is small. Hence the only semijoins that are not executed by dynamic W are those where the profit is near to zero. Hence the over all gain is small. The dynamic version of W might give better performance if after each semijoin operation it is possible to identify the semijoin with maximum profit and execute it. This is especially true in the case of the random distribution where it is difficult to predict the effect of one semijoin on the other attributes of a relation.

Also we have used Bloom filters as single attribute reducers. The reduction in a relation could be much greater if multiple Bloom filters were used to reduce the relation simultaneously. Another way to use Bloom filters might be to get a frequency distribution of the attribute values. With this kind of information it would be much more easier to accurately predict the semijoin benefit.

# Chapter 5
# Conclusions and Future work

## 5.1 Future Work

In this section we describe another algorithm which uses Bloom filters. Bloom filters can be used as reducers instead of just for the purpose of estimating the selectivities. Also as all the filters are built simultaneously multiple attribute Bloom filters can be used to reduce the relations. This should lead to a significant cost improvement. A detailed algorithm is given below. This algorithm modifies algorithm Bloom+DW to use Bloom filters as reducers.

1. Sort the attributes in ascending order of their sizes. Select the attribute whose reducer has the smallest size and is profitable. The size of the reducer $d_{xj}^*$ is estimated as follows

   $$\left|d_{xj}^*\right| = \rho(d_{aj}) * \rho\left(d_{bj}\right) * \ldots * \rho(d_{nj}) * S(d_{xj})$$

2. Construct the reducer for join attribute j:

   a. Consider semijoin $d_{aj} \rtimes d_{bj}$. The cost for this semijoin is calculated as $S(d_{aj})$. The benefit is

$S(R_b) * \left(1 - \frac{|h(d_{aj}) \cap h(d_{bj})|}{|h(d_{bj})|}\right)$

The marginal profit is calculated in the usual manner.

b. The semijoin is added to the schedule if it is gainful.

c. Execute each semijoin in the reducer dynamically. After the execution of the semijoin $d_{aj} \rtimes d_{bj}$, h($d_{bj}^*$) is reconstructed dynamically using the reduced attribute $d_{bj}$. Also use the Bloom filters for other attributes to see if the current tuple can be selected. The current tuple is selected only when the semijoin attributes value is present and for each of the other attributes the corresponding bit in their filter is set to 1. As the relation is being reduced reconstruct the Bloom filters for all the attributes.

d. If the semijoin is executed then consider the semijoin $d_{bj}^* \rtimes d_{cj}$ else consider the semijoin $d_{aj} \rtimes d_{cj}$.

3. Apply the reducer by executing all profitable semijoins$d_{mj}^* \rtimes R_i$. The cost for the semijoin is S($d_{mj}^*$). The benefit is

$S(R_i) * \left(1 - \frac{|d_{mj}^*|}{|h(d_{ij})|}\right)$

if $R_i$ was used in the construction of the reducer; otherwise the benefit is estimated as

$S(R_i) * \left(1 - \frac{|h(d_{ij}) \cap h(d_{mj}^*)|}{|h(d_{ij})|}\right)$

Our hypothesis is that this algorithm will give significant improvements over the algorithms proposed and tested in this thesis.

## 5.2 Conclusions

In this thesis, we have studied the performance of algorithms based on dynamic query execution and Bloom filters. The performance study of these algorithms was carried out against synthetically generated databases. The performance of the algorithms was compared against a well known heuristic for query processing, the AHY algorithm. Three different algorithms were compared. The first algorithm was based on the dynamic execution of a query. This algorithm used the most current information for calculation and reduced the errors due to the propagation of inaccuracies. The second algorithm used Bloom filters to obtain more accurate estimates of selectivities. The third algorithm is a combined strategy which is by dynamic and uses Bloom filters.

The performance study was done using a large number of synthetically generated test queries and databases. In total 12 different query types were used. For each query type 100 different queries were generated. The database was generated using both random and uniform distributions. The experimental results indicated that in each case slight improvements were achieved over the previous algorithm W. In all cases the improvement was significant compared to algorithm AHY. The experiments have shown that with methods for better estimation of selectivities and using dynamic execution the performance of algorithms can be improved.

# Selected Bibliography

[AHY83] Peter M. G. Apers, Alan R. Hevner, and S. Bing Yao. Optimization algorithms for distributed queries. *IEEE Trans. on Software Engineering*, pages 57–68, January 1983.

[AM91] A. Ahn and S. Moon. Optimizing joins between two fragmented relations on a broadcast local network. *Info. Syst.*, 16(2), 1991.

[Bab79] E. Babb. Implementing a relational database by means of specialized hardware. *ACM Trans. on Database systems*, 4.1:1–29, 1979.

[BGW+81] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. Rothnie. Query processing in a system for distributed database (SDD-1). *ACM Transactions on Database Systems*, 6(4):105–128, 1981.

[Blo70] B. H. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BPR90] P. Bodorik, J. Pyra, and J. S. Riordon. Correcting execution of distributed queries. *Proc. Second Int. Symp. on Databases in parallel and Distributed Systems*, pages 192–201, July 1990.

[BR88] P. Bodorik and J. S. Riordon. Distributed query processing optimization objectives. *Proc. Fourth Int. Conf. on Data Engineering*, pages 320–329, 1988.

[Bra84] K. Bratbergsengen. Hashing methods and relational algebra operations. *Procs. of the 10th VLDB Conf*, pages 323–333, 1984.

[BRP89] P. Boderick, J.S. Riordan, and J. Pyra. Dynamic distributed query processing techniques. *In 17th Annual ACM Computer Science conference*, 1989.

[BRP92] P. Boderick, J.S. Riordan, and J. Pyra. Deciding to correct distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 4(3), 1992.

[CL84] L. Chen and V. Li. Improvement algorithms for semijoin query processing programs in distributed database systems. *IEEE on Computers*, 33(11), 1984.

[CL90] L. Chen and V. Li. Domain-specific semijoin: A new operation for distributed query processing. *Info. Sci. 52*, 1990.

[Coo90] Martin C. Cooper. Estimating optimal parameters for parallel database hardware. *Int. Journal of Systems Sci*, 23(1):119–125, 1990.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw Hill, 1984.

[CY93] M. Chen and P. Yu. Combining join and semijoin operations for distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 5(3), 1993.

[Dat90] C. Date. *An Introduction to database Systems.* Addison Wesley, 1990.

[Des92] Bipin C. Desai. *An Introduction to Database Systems.* McGraw Hill, 1992.

[Gre82] L. L. Gremillion. Designing a bloom filter for differential file access. *Communications of the ACM*, 25(9):600–604, 1982.

[HY79] Alan R. Hevner and S. Bing Yao. Query processing in distributed database systems. *IEEE Transactions On Software Engineering*, 5(3):177–187, 1979.

[KR87] H. Kang and N. Roussopoulos. Using 2–way semijoins in distributed query processing. *In Proc. 3rd Int. Conf. on Data Engineering*, 1987.

[Loh89] G.M. Lohman. Is query optimization a "solved" problem? In *Proceedings of the ODBF Workshop*, number Tech Report CS/E 89–005, pages 13–18, May 1989.

[LS88] F. Li and L.V. Saxton. Two-way join optimization in partitioned database systems. In *Proc. 2nd Int. Conf. on Database Theory*, pages 191–204, 1988.

[MBB95] J. M. Morrissey, S. Bandyopadhyay, and W. T. Bealor. A heuristic for minimizing total cost in distributed query processing. *Proceedings of 7th International Conference on Computing and Information. Trent University*, pages 736–758, 1995.

[MBK96] J. M. Morrissey, W. T. Bealor, and S. Kamat. A comparative evaluation of dynamic heuristics for cost minimization. *Proceedings of 8th International Conference on Computing and Information. University of Waterloo*, 1996.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. *Proc of the 12th VLDB conference*, August 1986.

[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 database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.

[Mul93] James K. Mullin. Estimating the size of a relational join. *Information Systems*, 18(3):189–196, 1993.

[Ngu81] N. G. Nguyen. Distributed query management for a local area distributed database system. *Proc. 2nd Int. Conf. on Distributed Computing Systems, Paris, France*, pages 188–196, April 1981.

[OV91] M.T. Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice Hall International, 1991.

[PC90] W. Perrizo and C. Chen. Composite semijoins in distributed query processing. *Information Sciences*, 1990.

[RK91] N. Roussopoulos and H. Kang. A pipeline n-way join algorithm based on the 2–way semijoin program. *IEEE Trans. on Knowledge and Data Engineering*, 3(4), 1991.

[Seg91] A. Segev. Strategies for distributed query optimization. *Information Sciences*, 54:67–89, 1991.

[VG84] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems*, 9,1, March 1984.

[WC93] C. Wang and M. Chen. On the complexity of distributed query optimization. *IBM Technical Report RC 18671*, 1993.

[WLC91] C. P. Wang, V. O. K. Li, and A. L. P. Chen. One-shot semijoin execution strategies for processing distributed queries. *Proc. 7th IEEE Data Eng. Conf.*, April 1991.

[YC83] C. T. Yu and C. C. Chang. On the design of query processing strategies in a distributed database environment. *Proceedings of the 1983 ACM-SIGMOD International Conference on Management of Data*, 1983.

[YLG$^+$86] C. Yu, L. Lilien, K. Guh, M. Templeton, D. Brill, and A. Chen. Adaptive techniques for distributed query optimization. *The second International Conference on Data Engineering*, pages 86–93, February 1986.

# Appendix A Experimental Data

A summary of experimental results for total cost is given in the following tables. Each row of the table indicates a query type. The second column gives the total cost for shipping the unoptimized relations to the query site. The subsequent columns gives the total cost for each of the five algorithms used in the experiments. The entries in each row represents the average over 100 different queries. The last row gives the averages over all the different query types.

| Query Type | NO_OP | AHY | W | DW | Bloom+W | Bloom+DW |
|---|---|---|---|---|---|---|
| 32 | 24462.80 | 14019.90 | 10870.20 | 9854.10 | 10415.30 | 9950.10 |
| 33 | 34702.00 | 15040.90 | 8797.90 | 7144.20 | 8891.50 | 7161.80 |
| 34 | 49733.00 | 12626.50 | 4548.80 | 3439.70 | 4625.80 | 3387.30 |
| 42 | 38067.60 | 13637.50 | 6897.40 | 6087.50 | 6811.60 | 5831.80 |
| 43 | 52416.40 | 14767.50 | 5268.80 | 4050.40 | 5242.80 | 4054.50 |
| 44 | 64650.00 | 13756.40 | 3085.20 | 2769.60 | 3361.20 | 2814.70 |
| 52 | 48359.10 | 13544.30 | 5093.80 | 5690.30 | 5149.90 | 3930.10 |
| 53 | 55520.00 | 13192.80 | 2375.90 | 2369.00 | 2384.70 | 2169.30 |
| 54 | 80700.00 | 13576.50 | 2321.10 | 2166.30 | 2351.20 | 2154.30 |
| 62 | 49740.00 | 17551.10 | 4157.50 | 4933.60 | 4157.30 | 3666.60 |
| 63 | 79650.00 | 14386.50 | 2457.80 | 2107.00 | 2527.00 | 2104.30 |
| 64 | 98620.00 | 14377.90 | 2282.80 | 2023.30 | 2274.90 | 2013.60 |
| Average | 56385.07 | 14206.48 | 4846.43 | 4386.25 | 4849.43 | 4103.20 |

Figure A.1 Results of Uniform Distribution at 100% connectivity

| Query Type | NO_OP | AHY | W | DW | Bloom+W | Bloom+DW |
|---|---|---|---|---|---|---|
| 32 | 19694.20 | 11283.70 | 9889.90 | 9499.90 | 9931.20 | 9581.00 |
| 33 | 29914.20 | 14168.30 | 10478.30 | 10318.30 | 10469.40 | 10287.20 |
| 34 | 30784.90 | 13194.30 | 8188.40 | 5535.60 | 7549.20 | 5523.30 |
| 42 | 26160.00 | 14684.00 | 11765.10 | 11657.60 | 11758.80 | 11759.30 |
| 43 | 42085.00 | 18952.70 | 11361.10 | 12024.90 | 11361.10 | 9769.00 |
| 44 | 45287.10 | 15451.50 | 7005.80 | 5833.90 | 6936.20 | 5926.70 |
| 52 | 39718.60 | 17973.30 | 10505.50 | 11734.70 | 10385.50 | 10525.10 |
| 53 | 46090.00 | 16899.90 | 7404.70 | 5801.70 | 7081.90 | 6529.00 |
| 54 | 66800.00 | 18303.70 | 5609.90 | 3563.10 | 5054.10 | 3486.20 |
| 62 | 36724.80 | 18242.80 | 8814.50 | 8556.90 | 8705.70 | 8540.10 |
| 63 | 54860.00 | 18705.70 | 7957.80 | 7202.30 | 7963.60 | 6783.70 |
| 64 | 59430.00 | 20138.80 | 4483.10 | 3212.00 | 4433.60 | 3319.10 |
| Average | 41462.40 | 16499.89 | 8622.01 | 7911.74 | 8469.19 | 7669.14 |

Figure A.2 Results for Uniform Distribution with 75 % connectivity

| Query Type | NO_OP | AHY | W | DW | Bloom+W | Bloom+DW |
|---|---|---|---|---|---|---|
| 32 | 21443.40 | 15555.10 | 14280.30 | 16394.80 | 14280.30 | 16396.10 |
| 33 | 23895.80 | 15781.30 | 13602.20 | 13458.20 | 13321.00 | 13563.90 |
| 34 | 26894.40 | 13176.50 | 8411.30 | 7698.40 | 8460.50 | 7704.70 |
| 42 | 25919.70 | 19518.00 | 16740.00 | 17060.10 | 16775.90 | 17134.30 |
| 43 | 33057.30 | 19081.50 | 14873.70 | 14653.40 | 14488.40 | 14650.50 |
| 45 | 32622.10 | 16623.80 | 11109.80 | 10024.50 | 11120.60 | 10048.40 |
| 52 | 31120.60 | 20424.80 | 15580.60 | 16430.40 | 15580.60 | 16426.30 |
| 53 | 35810.80 | 19506.30 | 11737.30 | 10257.10 | 11278.80 | 10098.30 |
| 54 | 38757.40 | 21154.10 | 13689.20 | 11189.20 | 12642.00 | 11348.60 |
| 62 | 38622.20 | 22318.20 | 12431.70 | 14135.80 | 11870.60 | 13899.90 |
| 63 | 50648.10 | 25420.80 | 15899.60 | 16533.60 | 15779.50 | 16427.60 |
| 64 | 63070.00 | 23027.50 | 7298.40 | 5365.70 | 7838.50 | 5229.70 |
| Average | 35155.15 | 19298.99 | 12971.18 | 12766.77 | 12786.39 | 12744.03 |

Figure A.3 Results of Uniform Distribution at 50 % connectivity

| Query Type | NO_OP | AHY | W | DW | Bloom+W | Bloom+DW |
|---|---|---|---|---|---|---|
| 32 | 19694.20 | 11283.70 | 9889.90 | 9499.90 | 9931.20 | 9581.00 |
| 33 | 29914.20 | 14168.30 | 10478.30 | 10318.30 | 10469.40 | 10287.20 |
| 34 | 30784.90 | 13194.30 | 8188.40 | 5535.60 | 7549.20 | 5523.30 |
| 42 | 26160.00 | 14684.00 | 11765.10 | 11657.60 | 11758.80 | 11759.30 |
| 43 | 42085.00 | 18952.70 | 11361.10 | 12024.90 | 11361.10 | 9769.00 |
| 44 | 45287.10 | 15451.50 | 7005.80 | 5833.90 | 6936.20 | 5926.70 |
| 52 | 39718.60 | 17973.30 | 10505.50 | 11734.70 | 10385.50 | 10525.10 |
| 53 | 46090.00 | 16899.90 | 7404.70 | 5801.70 | 7081.90 | 6529.00 |
| 54 | 66800.00 | 18303.70 | 5609.90 | 3563.10 | 5054.10 | 3486.20 |
| 62 | 36724.80 | 18242.80 | 8814.50 | 8556.90 | 8705.70 | 8540.10 |
| 63 | 54860.00 | 18705.70 | 7957.80 | 7202.30 | 7963.60 | 6783.70 |
| 64 | 59430.00 | 20138.80 | 4483.10 | 3212.00 | 4433.60 | 3319.10 |
| Average | 41462.40 | 16499.89 | 8622.01 | 7911.74 | 8469.19 | 7669.14 |

Figure A.4 Results for Random Distribution at 100 % connectivity

| Query Type | NO_OP | AHY | W | DW | Bloom+W | Bloom+DW |
|---|---|---|---|---|---|---|
| 32 | 23374.90 | 18852.60 | 20035.70 | 30845.92 | 20039.80 | 22686.80 |
| 33 | 32620.00 | 29609.00 | 26993.10 | 28505.70 | 26940.90 | 28745.90 |
| 34 | 38484.30 | 34214.20 | 28794.80 | 28115.20 | 28070.10 | 28078.90 |
| 42 | 34344.20 | 31695.30 | 27327.50 | 29652.50 | 26967.40 | 29878.80 |
| 43 | 47218.00 | 42631.50 | 33791.00 | 35416.50 | 33383.30 | 35504.30 |
| 44 | 59241.90 | 51177.70 | 36989.50 | 32413.80 | 35446.40 | 31520.50 |
| 52 | 45395.10 | 41988.90 | 33123.90 | 35026.90 | 33065.20 | 35336.50 |
| 53 | 59404.60 | 52687.80 | 38015.50 | 37117.90 | 37210.40 | 37973.90 |
| 54 | 64799.50 | 53814.30 | 25297.70 | 22445.50 | 23927.80 | 22568.10 |
| 62 | 48150.60 | 46189.90 | 34460.80 | 36456.20 | 34165.60 | 36206.50 |
| 63 | 61509.40 | 57590.40 | 35920.00 | 32080.80 | 35458.10 | 31843.30 |
| 64 | 78795.90 | 68026.70 | 29401.60 | 27678.90 | 29579.60 | 25437.70 |
| Average | 49444.87 | 44039.86 | 30845.92 | 30639.21 | 30354.55 | 30481.77 |

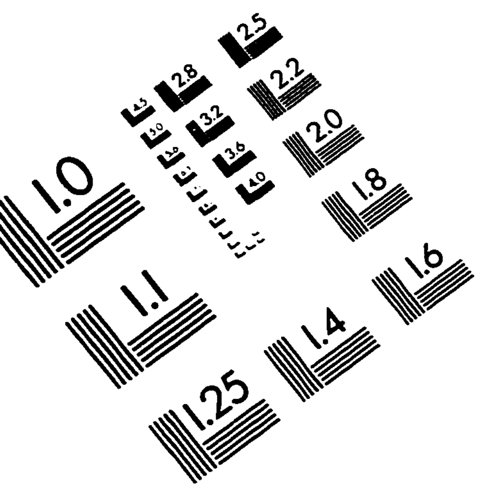
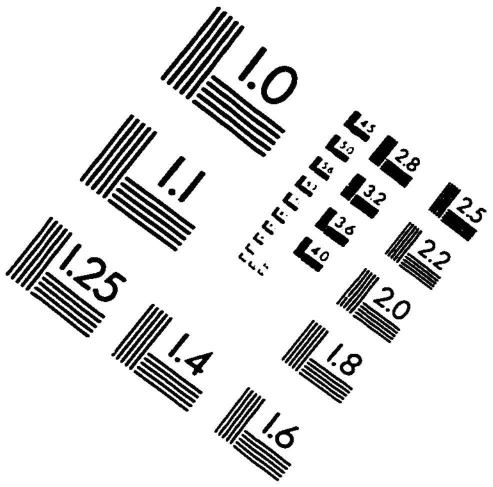
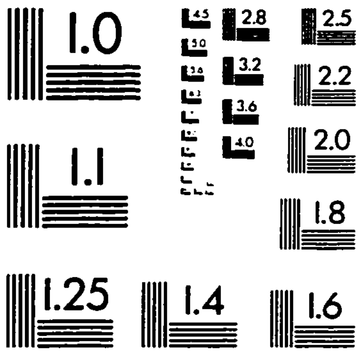
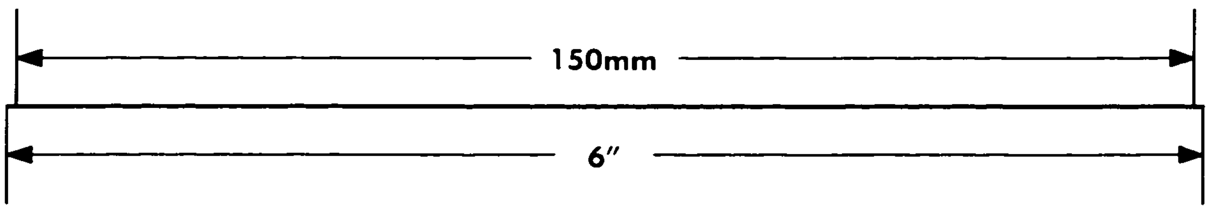
Figure A.5 Results of Random Distribution at 75 % connectivity

| Query Type | NO_OP | AHY | W | DW | Bloom+W | Bloom+DW |
|---|---|---|---|---|---|---|
| 32 | 21406.50 | 20957.00 | 19018.80 | 21075.70 | 19018.80 | 21076.20 |
| 33 | 26710.00 | 24232.70 | 22904.40 | 25600.60 | 22928.50 | 25493.10 |
| 34 | 29185.50 | 27287.50 | 25408.10 | 24514.00 | 24840.70 | 24713.30 |
| 42 | 28649.20 | 26606.10 | 23541.20 | 24944.60 | 23585.70 | 25028.90 |
| 43 | 35515.00 | 32779.00 | 29706.00 | 32351.40 | 29721.20 | 33979.40 |
| 44 | 43202.80 | 39027.30 | 35388.40 | 37838.20 | 35236.80 | 37839.40 |
| 52 | 32880.20 | 31046.00 | 27554.30 | 29694.20 | 27498.20 | 30278.80 |
| 53 | 45809.90 | 42308.60 | 37570.60 | 41247.90 | 37502.10 | 41406.90 |
| 54 | 52391.00 | 49012.80 | 42365.50 | 43537.00 | 41710.70 | 44173.50 |
| 62 | 38127.70 | 35887.90 | 29351.60 | 32227.50 | 29322.80 | 33705.90 |
| 63 | 47103.20 | 44804.80 | 37687.00 | 38686.40 | 37464.50 | 39586.50 |
| 64 | 55084.10 | 52485.90 | 41437.90 | 41423.50 | 40601.80 | 40629.90 |
| Average | 38005.43 | 35536.30 | 30994.48 | 32761.75 | 30785.98 | 33159.32 |

Figure A.6 Results of Random Distribution at 50 % connectivity

# Vita Auctoris

**Sandeep Kamat** was born in Bombay, India. He graduated from University of Poona obtaining a Bachelor's Degree in Computer Science in 1991. From there he joined the Indian Institute of Technology, Bombay as Software Engineer. He is currently a candidate for a Master's degree in Computer Science at the University of Windsor and will complete all degree requirements in the Fall of 1996.





150mm
6"