This is to certify that the

dissertation entitled

OPTIMIZING THE COST OF RELATIONAL JOIN QUERIES

presented by

Farshad Fotouhi

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

B. Ramamik

Major professor

Date 4-27-88

OPTIMIZING THE COST OF RELATIONAL JOIN QUERIES

By

Farshad Fotouhi

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1988

ABSTRACT

OPTIMIZING THE COST OF RELATIONAL JOIN QUERIES

By

Farshad Fotouhi

The join operation is one of the most time consuming operations in a relational database system because it requires a large amount of cross referencing between tuples of different relations. Therefore, the efficiency of the join operation has a deterministic effect on the system performance. The best method for performing join depends, in general, on the access methods available, the parameters of the relations involved, and the context in which the query is presented. The objective of this thesis is to determine *optimal strategies* for performing join for a given set of constraints and assumptions. Here, the theoretical lower bound on the number of disk I/Os is achieved. Both join-only queries and queries involving restrictions, projections and join are considered.

Here, the existing join algorithms are classified into *Relation-Scan, Index-Scan,* and *Hybrid* classes. This classification is based on the availability and use of indices on the join attribute values. This research will show that each class of algorithms performs best for a range of parameter values. It is shown that the relation-scan class of algorithms performs best when all or most of the tuples of the joining relations participate in the join. For the index-scan class of algorithms, several graph models are proposed in order to show that the optimization problem for these algorithms is NP-hard. Therefore, a heuristic algorithm with linear time complexity is given. For the hybrid class, several algorithms which are based on preprocessing a new auxiliary data structure, called the *Partial-Relations*, are proposed. It is shown that for a wide range of parameter values the proposed algorithms perform better than the best available algorithms of the other two classes.

To my parents, Mahmmoud Fotouhi and Azarmidokht Shazad

ACKNOWLEDGEMENTS

I wish to express my appreciation to Professor Sakti Pramanik for his valuable assistance and continual support in every phase of the preparation of this work. I also wish to thank the committee members, Professor Forsyth, Professor Ni, Professor Shanblatt and Professor Erickson for their helpful comments on the contents of this work.

Professor Esfahanian provided many useful suggestions and participated in many of the discussions which helped to develop some of the theoretical results in Chapter 3 of this document.

Finally, I want to thank my family, without whose love and inspiration this would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND PROBLEM STATEMENT

1.1. Relational Database Systems

As mass storage prices fall and computers are used extensively throughout businesses and other organizations, vast amounts of valuable information are piling up in electronic repositories. Finding ways to organize this information, and to allow convenient access to it, is steadily becoming more important. The emergence of advanced concepts based on artificial intelligence will make databases and their management even more critical.

Software developers have come up with a variety of database management systems, which accept, organize, store, and retrieve information quickly and efficiently. A *database management system* (DBMS) consists of a collection of interrelated *data* and a set of *programs* to access that data. The collection of related data which contains information about an enterprise is usually referred to as the *database*.

In order to describe the structure of a database, a *data model* is created and is administered by *database administrator*. A data model is a collection of conceptual tools for describing data, data relationships, data semantics, and data constraints [Kort86]. Database designers can choose among several data models, of which three currently reign as the most popular [Date75]. These models are the *network data model*, the *hierarchical data model* and the *relational data model*. In this research, only the relational data model [Codd70], which is the data model for relational database systems, is considered. A *relational database* consists of a collection of *relations*, each of which is assigned a unique name. In this model, data are represented as a table, with each horizontal row representing a *tuple* or a *record* and each vertical column representing one of the *attributes*, or *fields* of the tuple. In other words, a relation of arity n (i.e., a

relation with n attributes) is a subset of tuples from the *cartesian products* of the domains $D_1 \times D_2 \times \cdots \times D_n$. The cartesian product of two relations R and S, written $R \times S$, of arity k_1 and k_2, respectively is the set of (k_1+k_2)– tuples whose first k_1 components are from a tuple in R and last k_2 components are from a tuple in S.

Figure 1.1 shows a relational model of a simplified inventory database. This example was first popularized by Codd [Codd70]. Individual rows in the PART relation correspond to individual types of parts in the inventory. Because rows are required to be nonredundant within a relation, each row can be identified uniquely by a subset of its attributes. For example, in Figure 1.1a S# can be used to identify a row uniquely, while S# and P# must be used together to identify a row of the SUPPLY relation. Such a set of attributes is called the *key* of the relation. If a subset of the attributes in one relation is the key of the second relation, then such a key is called a *foreign key*. For example, in Figure 1.1, S# and P# are the keys of the SUPPLIER and PART relations, respectively. Thus, S# and P# in SUPPLY relation are called foreign keys.

S#	SN	SL
S1	ABC Co.	MI
S2	XY Co.	NY
S3	JK Co.	NY

(a) SUPPLIER Relation

P#	PD
P1	CAM
P2	GEAR

(b) PART Relation

S#	P#	Q
S1	P1	5
S1	P2	6
S3	P1	6

(c) SUPPLY Relation

Figure 1.1. A Simplified Relational Database for Keeping Inventories.

1.2. Operations on a Database

Modeling data in a relational structure lends itself to the development of high-level *query languages*. Several styles of relational query languages have been developed. One of these is *relational algebra*. Relational algebra consists of a set of operators which can be applied to relations to create a new relation. These operators can be composed to form expressions. A typical set of relational operators is *projection, selection, join, union, intersection* and *difference*. Here, both join-only *queries*†, and queries involving selections, projections and join are considered.

A projection operation is the elimination of certain fields of all tuples. In many cases, a number of the fields contain information which is irrelevant for the query and these fields are deleted. This may have the effect, however, of producing multiple, identical tuples, since many tuples may be identical in a number of attributes. The relational database model assumes that all the tuples are unique. Thus, in theory, duplicates must be eliminated. Figure 1.2 shows the effect of projecting on P# of the relation SUPPLY. Note that the result is a relation (duplicates on *P* 1 have been removed) with only one attribute.

P#
P1
P2

Figure 1.2. Effect of Projecting on P# of Relation SUPPLY.

The selection operation, sometimes referred to as *restriction*, is a specification which eliminates some of the tuples of the relation. The specification is a Boolean expression defined over the columns of the relation. The expression may consist of arithmetic comparison operations (i.e., $=,\neq,<,>,\leq,\geq$) on the value of one or more columns. The result of selection is a new relation with only the rows from the original

† A *query* is a statement requesting the retrieval of information.

relation which satisfy the selection condition. Figure 1.3 gives an example of the selection operation on the SUPPLIER relation where the SL="NY".

S#	SN	SL
S2	XY Co.	NY
S3	JK Co.	NY

Figure 1.3. An Example of a Selection Operation.

The join operation is one of the most important operations in query processing. Join is a binary operation that allows us to combine the selection and cartesian products into one operation. The θ–*join* of two relations R and S on columns i and j is denoted by $R \underset{i\theta j}{\bowtie} S$ which is the shorthand for $\sigma_{i\theta(r+j)}(R \times S)$. Here σ represents the selection and θ is an arithmetic comparison operator. If both R and S have columns that are named the same then the equijoin of R and S on their common attributes is called *natural join* and it is denoted by $R \bowtie S$. Figure 1.4 gives an example of a natural join between two relations SUPPLY and SUPPLIER, of Figure 1.1. As shown in the example, this operation allows a user to navigate through the database. In this research the natural join of two relations on a single domain is considered, and from now on wherever the term join is used we mean natural join.

S#	P#	Q	SN	SL
S1	P1	5	ABC Co.	MI
S1	P2	6	ABC Co.	MI
S3	P1	6	JK Co.	NY

Figure 1.4. An Example of a Natural Join.

1.3. Access Methods

Many queries require only a small number of the tuples to be retrieved from a relation. Therefore, the system is not efficient if it must access all the tuples of the relation to check for the tuples which satisfy the user's request. Ideally, the system should be

able to locate those tuples directly. In order to allow this form of access, an additional data structure associated with relations was designed. Here two such structures, namely *indexing* and *hashing* are considered.

1.3.1. Indexing

An *index* on attribute *A* of relation *R* permits rapid access to a single tuple that has the desired value in that attribute. An index consists of pairs whose first component is a value from attribute A and whose second component is the *Tuple Identifier* (TID) of a tuple having that value. A TID is assumed to give direct access to the tuple, so that at most one page is accessed if a tuple is referenced using a TID. An index is stored in a special way to provide rapid access to it. The model which is widely used in most database systems is B-Tree [Baye72] or a variant of it [Knut73, Come79].

A B-tree index takes the form of a balanced tree in which the path from the root to any leaf of the tree is the same length. Each node in the tree has between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree. Leaf pages contain (key,TID) pairs in sorted order, and the higher level pages contain pairs that consist of the largest key of the lower-level page with a pointer to that page. These pairs are also sorted. The cost of search operation is proportional to $\log N$, where N is the number of tuples in the file [Baye72].

The efficiency of an index in query processing depends on whether the relation is *clustered* or *unclustered* with respect to the index. Suppose an index *I* is used to extract the tuples of relation *R*. If each data page of *R* is accessed at most once, then *R* is clustered with respect to the index *I*. The index *I* may be termed a *clustering* index with respect to *R*. On the other hand, if the data pages of *R* are referenced in a random, approximately uniformly distributed manner, then *R* is *unclustered* with respect to *I*.

1.3.2. Hashing

One disadvantage of indexing schemes is that an index structure must be traversed in order to locate a datum. Hashing avoids the traversing of an index structure and it gives the best average access time to a single tuple. However, indexing is more efficient for range queries. The basic idea in hashing is to divide the tuples of a relation among *buckets*, each consisting of one or more blocks of storage. The address of a tuple is then determined by computing a function, called the *hash function*, on the search-key value of the desired tuple.

Conventional hashing functions such as those discussed by Lum [Lum 71, Lum 73], Knuth [Knut73], Severance [Seve74] and Knott [Knot75] are useful for static files. However, when the file size is dynamic, either a large amount of secondary storage must be pre-allocated or long overflow chains and frequent rehashing must be tolerated. Starting in mid 70's several new hashing techniques such as *linear hashing* [Litw80, Lars85], *virtual hashing*[Litw76], *extendible hashing*[Fagi79,Lome83], and *dynamic hashing*[Lars78] were proposed. In these techniques rehashing is avoided by allowing the storage space to dynamically adjust to the number of tuples actually stored. Most of these techniques, such as extendible hashing, do not allow overflow to occur, thus, the expected number of accesses in these techniques is minimized. Here, wherever the term hashing is used, I mean a conventional hashing function.

1.4. Motivation

The rapid technological advances of the recent decade have made possible the use of database management systems over a wide range of applications. Among the most significant problems in computer science today is the issue of efficient implementation of database management systems to support an array of applications. Certain types of database operations are not widely performed today because of the high cost. One such operation is the join.

The join operation is one of the most important and time-consuming operations in a relational database system. Because it allows the user to navigate through the database, it is an important operation. In addition, many of the techniques used for performing join can be used for other relational operators such as a cross product and aggregate functions. Join is a time-consuming operation because it requires a large amount of cross referencing between tuples of different relations. Therefore, the efficiency of the join operation has a deterministic effect on the system performance. Because of its importance, the join operation has been a subject of intensive study in the development of relational database systems, and different approaches have been proposed.

The objective of this thesis is not to give a join algorithm but to develop principles by which a join can be implemented. The best method depends on the access methods available, the parameters of the relations involved, and the context in which the query is presented.

1.5. Problem Statement

The primary goal of this work is to determine *optimal strategies* for performing join for a given set of constraints and assumptions. Both join-only queries and queries involving restrictions, projections and join are considered. Optimal strategy requires the least cost among all other strategies. The cost is measured based on the number of disk I/Os when a small size main memory buffer is assumed. When large size main memory buffer is available, the cost is measured based on the number of disk I/Os as well as the amount of CPU usage.

In this thesis, the existing join algorithms are classified into *Relation-Scan, Index-Scan*, and *Hybrid* classes. This classification is based on the availability and use of indices on the join attribute values. This research will show that each class of algorithms is best for a range of parameter values. There has been intensive study of the relation-scan class of algorithms. Therefore, this research concentrates on the last two

classes of algorithms. Using graph models for the index-scan class of algorithms I will show that the optimization problem for these algorithms is NP-hard. For the hybrid class, several algorithms which are based on preprocessing a new auxiliary data structure, called the *Partial-Relations*, are proposed here. The performance of these algorithms will be compared with that of the existing ones. This research will show that for a wide range of parameter values the proposed algorithms perform better than the relation-scan class of algorithms.

The organization of this thesis follows. Chapter 2 presents a classification of various join algorithms proposed in the literature. In chapter 3 several graph models are presented to analyze the performance of the join operation in a paging environment. The graph models are based on the block or page connectivity of the joining relations. In chapter 4 several join algorithms based on the *Partial-Relation Scheme* are presented. The performances of these algorithms are compared with some of the existing algorithms under various conditions. Chapter 5 contains the conclusions and suggestions for further work.

CHAPTER 2

A CLASSIFICATION SCHEME FOR JOIN ALGORITHMS

2.1. Introduction

The availability of inexpensive, large main memories coupled with the demand for faster response time is bringing a new perspective to database technology. Designers of database systems have assumed, until recently, that databases reside on disk during transaction processing. However, substantial performance gains can be achieved by letting a large portion of, or the entire database, reside in main memory. So far two approaches have been proposed for using large amounts of main memory in database systems. The first approach uses very large buffers to improve conventional disk access methods by storing part of the database or part of the indices. DeWitt *et al.*[DeWi84], Shapiro[Shap86], and Elhardt *et al.*[Elha84] have taken this approach in their work. The second approach, the *memory-resident database* approach, is to store the entire database in main memory. Ammann *et al.*[Amma], Lehman and Carey[Lehm86a, Lehm86b], and Leland and Roome[Lela85] take this approach.

Here, a classification of join algorithms under the assumption that the database is disk resident and a small or large main memory buffer is available (first approach) is considered. The second approach (i.e., memory-resident database) is not considered because it requires a redesign of the database management systems. That is, the algorithms and the data structures for query processing, concurrency control and recovery must all be restructured to stress the efficient use of CPU cycles and memory rather than disk accesses and disk storage.

The existing join algorithms may be categorized into three classes. The classification is based on the availability and use of the indices on the join attribute values. As shown in later chapters each class of algorithms is best for a range of

parameter values. The first class contains those algorithms which perform join by accessing all the tuples of the joining relations. This class of algorithms is referred to as *relation-scan* algorithms. This thesis show that the relation-scan class of algorithms performs better than the algorithms of the other two classes when most of the tuples of the joining relations participate in the join. The second class of algorithms uses indices to access only those tuples of the joining relations which participate in the join. Therefore, as shown in this thesis, the algorithms of this class perform better than the algorithms of the other two classes when a small number of tuples of the joining relations participate in the join. These algorithms are referred to as *index-scan* algorithms. The third class of algorithms uses the *semijoin* technique to perform join. The semijoin technique has been widely used in *distributed databases* and in *multiprocessor database machines* in order to reduce the communication costs. This class of algorithms is referred to as the *hybrid* class of algorithms. The next sections describe some of the algorithms proposed for each class.

2.2. Relation-Scan Class of Algorithms

Relation-scan algorithms perform join on all the tuples of the joining relations without any attempt to reduce the size of the relations. Therefore, when only a few tuples of the joining relations participate in the join, we show that the algorithms of this class are more costly than the algorithms of the index-scan and the hybrid classes. This situation occurs if a restriction or another join operation is applied to at least one of the joining relations before the join is performed. In this section some of the proposed algorithms in this class are considered.

2.2.1. Nested-Loop Join Algorithm

The simplest of all join algorithms in terms of implementation is the nested-loop algorithm [Gotl75, Blas77, Kort86]. The nested-loop join algorithm for two relations R and S is as follows:

a) Read R or as much as possible of R into the buffer.

b) For each record in S check if it has the same join values as any record of R stored in the buffer. If they match, join the two records and output the result.

c) Repeat this process for all parts of R.

We see that the complexity of this algorithm is in the order of $O(|R|\times|S|)$; where $|R|$ and $|S|$ represent the number of data pages in relations R and S, respectively. However, the nested-loop can be an efficient join algorithm if at least one of the joining relations is small enough to fit in the main memory buffer. This algorithm also takes advantage of different relation sizes in contrast to the sort-merge join algorithm which is described later.

Since this algorithm has a high degree of parallelism, it has been implemented in multiprocessor database machines [Bitt83, Vald84]. A *database machine* is a device which implements the operations of the database by means of hardware. The nested-loop algorithm when implemented on a multiprocessor machine with P processors proceeds in the following manner: Distribute the smaller relation (called *external*) among P processors. Next, the other relation (called *internal*) is broadcast page by page to these processors. Each processor joins each page in the memory of the external relation with the entire internal relation. If the external relation could not fit into the P processor's local memories, the same operation must be performed for each pages of the external relation.

2.2.2. Sort-Merge Join Algorithm

Another widely used join algorithm is the sort-merge technique[Blas77, DeWi84]. This algorithm requires initial sorting and multiway merging of both joining relations on their join attributes, and then merging the two sorted relations. The sort process itself can be optimized (see, for instance, Knuth [Knut73]). Initial sorting takes several passes over the relations and requires a considerable amount of CPU time. Sorting does not take advantage of different relation sizes. A detailed description of this algorithm follows.

Assume that both joining relations R and S are sorted in ascending order with respect to their join attributes A. Let us also assume that attribute A has no duplicate values in relation S. In order to join the two relations, two pointers r and s are used over relations R and S, respectively. Each pointer identifies a tuple in a relation. Both pointers are initially positioned on the first tuple of their relations. The merging of the two sorted relations starts by reading a page from each relation and comparing the join attribute values. If $r.A = s.A$, then r concatenate s is output, and r is advanced to the next tuple. $r.A$ represents the value of attribute A in the tuple pointed at by the pointer r. If $r.A > s.A$, s is advanced; otherwise, r is advanced. The process is iterated until the end of one of the relations.

In the case discussed above there is no looping, and the join I/O cost is exactly $|R|+|S|$. However, the I/O cost of m-way merge-sorting the two relations prior to merging them is $O(|R| \times (\log_{m-1}|R|) + |S| \times (\log_{m-1}|S|))$. Therefore, the complexity of this algorithm is in the order of $O(n \times \log n)$, where n is the number of the data pages to be sorted.

If attribute A has duplicate values in relation S, then the algorithm can be modified by adding another pointer s_1 on S positioned on the first tuple of the current run of tuples having the same values for attribute A. Suppose that r is to be advanced; let t represent the new tuple identified by the pointer. It $t.A = s.A$ the pointer s is set to the

position of the s_1 pointer. In this way a loop occurs on the run of equal values of attribute A in S. Note that if the join attribute of relation S can have replicated values, then the cost of merging scans is not guaranteed to be linear.

2.2.3. Hash-Based Join Algorithms

The main idea for hash-based algorithms is to partition the joining relations into smaller subrelations where the nested-loop algorithm or any other algorithm can be employed. This is an example of a divide and conquer technique. Bratbergsengen[Brat84] presented join algorithms based on *hashing*. The performances of the proposed algorithms are compared against the nested-loop and the sort-merge. He has shown that for small relations the nested-loop method is preferable to sort-merge and hashing algorithms. However, for large buffer partitioning the relations based on hashing are always better than the sort-merge join algorithm. DeWitt, *et al.*[DeWi84], and Shapiro[Shap86] have also proposed three hash-based join algorithms, namely the *simple-hash, GRACE-hash* and the *hybrid-hash*. They analyzed and compared the performances of these algorithms with the sort-merge algorithm. Their results were the same as Bratbergsengen[Brat84]. They have also shown that the hybrid-hash join algorithm performs better than the simple-hash and GRACE-hash over a wide range of parameter values. In the next three sections each of these algorithms and a recent hashing method proposed by Sacco [Sacc86a] will be described briefly.

2.2.3.1. Simple-Hash Join Algorithm

Let us assume that the hash table for the smaller of the two relations R and S fits in the main memory. Then the simple-hash join algorithm proceeds as follows:

1) Use a hashing function h to build a hash table for R in memory. Assume that R is smaller than S.

Scan S and for each tuple t of S, repeat the following steps.

2) Compute the hash value of the tuple t using the hashing function h.

3) Use the hashed value for t obtained in step 2 to search the hash table of R for a match. In case of a match output the result.

If the hash table for R will not fit in memory, the simple- hash join algorithm proceeds as follows: Fill the memory with a hash table for part of R. Scan S against that hash table and if there is a match, output the resulting tuple. Build a hash table for another part of R in the memory and scan the *remainder* of S against it. Repeat this process until a hash table for all parts of relation R has been created.

2.2.3.2. GRACE-Hash Join Algorithm

The GRACE-hash join algorithm has been implemented on the GRACE relational database machine [Kits83,Moto83,Kits84]. The GRACE architecture is one of the candidate relational database machines for the Japanese fifth-generation computer project sponsored by ICOT†. The GRACE database machine is organized for join-intensive applications. It utilizes the concepts of associative disks with filtering, hash-based data partitioning, and pipeline sort-merge in its processing modules. The basic premise of the architecture relies on the fact that the tuples of relations can be distributed into hash buckets determined by the range of values of the attribute to be hashed, which is the join attribute. Accordingly, the join complexity would be simplified from the $O(|R|\times|S|)$ complexity of the brute force nested-loop join algorithm, if we hashed the join attributes of these relations so that

$$|R| = \sum_{i=1}^{s} n_i \quad \text{and} \quad |S| = \sum_{i=1}^{s} m_i \quad ,$$

† Institute for New Generation Computer Technology

where n_i and m_i are the sizes of the ith bucket of respective relations and s is the number of buckets in each relation, the join complexity would be simplified. This is because we would compare only those tuples of relations which occupy compatible buckets (i.e., for $i \neq j$, we do not process n_i with m_i). Accordingly, the overall complexity would be $O(\sum_{i=1}^{s} n_i \times m_i)$.

The GRACE-hash join algorithm is executed in two phases as outlined in DeWitt *et al.*[DeWi84] and Shapiro [Shap86]. In the first phase the relations R and S are partitioned into $|M|$ sets; where $|M|$ is the number of pages in the main memory. The algorithm uses one page of memory as an output buffer for each of the |M| sets in the partition of R and S. The partitioning of the relations is done by using a hashing function. In the second phase the join is performed using a hardware sorter to execute a sort-merge algorithm on each pair of sets in the partition. To provide a fair comparison between different algorithms, DeWitt *et al.* [DeWi84] have used the hashing technique to perform join during the second phase.

2.2.3.3. Hybrid-Hash Join Algorithm

In this algorithm a large main memory buffer is used to minimize disk I/O. In contrast, the GRACE-hash join algorithm uses all the main memory as a buffer to partition the relations. The hybrid-hash algorithm [DeWi84, Shap86], however uses only as many pages (B) as are necessary to partition R into sets that can fit in the memory. The rest of the memory is used for a hash table that is processed at the same time that R and S are being partitioned. The algorithm proceeds as follows:

1) Assign the ith output buffer page to partition R_i, for $i=1,...,B$. Scan R and hash each tuple of it using a hashing function h. If the hashed tuple belongs to R_0, place it in memory in the hash table for R_0. Otherwise, place it in its appropriate output buffer page. When this step is finished, we have a hash table for R_0 in memory,

and $R_1, R_2, \ldots, R_B$ are on the disk. Note that the hash table for R_0 has $|M|-B$ pages and $R_1, \ldots, R_B$ are of equal size.

2) Assign the ith output buffer page to set S_i, for $i=1,...,B$. Scan S, and use the hashing function h to hash every tuple of S. If the hashed tuple belongs to S_0, search the hash table for R in memory for a match. If there is a match, output the result tuple; otherwise ignore the tuple. On the other hand, if the hashed tuple does not belong to S_0, then place the tuple in the appropriate output buffer page, S_i for some $i>0$. Now $R_1, \ldots, R_B$ and $S_1, \cdots, S_B$ are on disk.

Repeat steps (3) and (4) for $i=1,...,B$.

3) Read R_i and build a hash table for it in the memory.

4) Scan the partition S_i and hash each tuple of it. Scan the hash table of R_i, which is in memory, for a match. If there is a match, output the result. Otherwise, toss the S tuple.

As mentioned earlier, DeWitt *et al.*[DeWi84] or; and Shapiro [Shap86] have shown that the hybrid-hash and simple-hash outperform the sort-merge and the GRACE-hash join algorithms under the assumption that large main memory buffer is available. Their comparisons were based on the execution time of the algorithms (i.e., I/O time and the cpu usage).

2.2.3.4. Join Algorithms Based on Fragmentation Technique

Fragmentation[Sacc86a] joins two relations by recursively partitioning two unordered relations into fragments until each fragment of the smallest relation in no larger than $|M|-1$. The basic idea of fragmentation is to partition the joining relations R and S into n disjoint fragments in such a way that

1) for $1 \le i \le n$, fragments ith of both relations contain tuples whose join attribute values are drawn from a set T_i of values (which identifies a bucket) and

2) for any given i,j $(i \neq j)$, S_i and S_j are disjoint.

The partitioning phase logically partitions the join attribute range into $(|M|-1)^j$ disjoint buckets by using a hashing function. Each bucket identifies a pair of fragments, one for each relation. At each partitioning step a bucket i is partitioned into $|M|-1$ subbuckets. The problem of joining two relations then reduces to the problem of joining $(|M|-1)^j$ independent fragments of the two relations. Since each fragment of the smaller relation fits in the buffer, the final join phase has a linear cost. It has been shown in Sacco [Sacc86a] that the cost of fragmentation is $O((|R|+|S|)\times(\log_{|M|-1}|S|))$, with the assumption that S is a smaller relation than R.

Sacco [Sacc86] has compared the performances of the fragmentation technique against the sort-merge join algorithms for various relation sizes. He has shown that if at least one of the joining relations (i.e., the larger relation) is already sorted then the sort-merge is better than the fragmentation. However, if the larger relation or both relations need to be sorted then the fragmentation technique outperforms the sort-merge.

2.3. Index-Scan Class of Algorithms

The algorithms of this class attempt to reduce the size of the joining relations before performing the join. The reduction is done by using the indices on the join attributes. Therefore, these types of algorithms, require the existence of the indices on either one or both relations on their join attributes in order to be operational. If indices exist on both relations then the join is first performed on the indices, and a set of *pointer* pairs to the tuples (*tuple ids*) that will be joined is obtained. A pointer consists of a page number and an offset within the page. Using the pointer pairs, the selected pages of the relations are fetched and the appropriate tuples within these pages are then joined.

When only the index on one of the two relations, say S, exists, then the algorithm proceeds as follows:

Repeat steps below for all the tuples of the smaller relation *R*.

1) Scan relation *R* and for each tuple of *R* fetch join attribute value, *r*.

2) Scan the index on the join attribute of relation *S*, with *r* as the search key. If *r* exists then fetch the corresponding tuples in *S* and join it with the tuple of *R*.

For algorithms of this class Fotouhi and Pramanik [Foto88] have shown that if more than 50% of the tuples of the joining relations participate in the join, then the selected pages of the relations may be reaccessed several times. Reaccessing of the pages can be more costly than accessing the entire tuples of the joining relations as in the algorithms of the first class. Therefore, the algorithms of this class are good only when prior selection or join is performed on the joining relations.

Blasgen and Eswaran[Blas76, Blas77] have studied the properties of various join algorithms in considerable detail. In order to be operational some of their proposed algorithms require the existence of indices on the join attributes. They have considered queries which contain the join operation as well as selections and projections. They examined the cases where various indices were present, where the restriction had various effects, and for various sizes of the relations. The cost of the algorithms is computed in terms of disk operations and showed that many different algorithms are best under particular sets of conditions. Goodman[Good80] has also proposed a few modifications of these algorithms. It has been shown in [Blas77] that in the absence of indices, the sort-merge join algorithm performs better than the other join algorithms. However, if clustering indices on the join attributes exist, then it is more cost effective to use the indices to perform the join.

This research developed several *graph models* for some of the index-scan algorithms to determine an ordered page or block access sequence which requires the smallest size main memory buffer. The graph is formed for the page or block connectivity of the joining relations. In these models the join-participating pages of the relations are accessed only once. I will show that the problem of determining an ordered list of pages

or blocks which requiring a minimum size buffer is NP-hard. Therefore, a heuristic algorithm which determines an ordered page access sequence requiring a near optimal buffer size is presented. The graph models are discussed in the next chapter.

There have also been join algorithms which use a *precompiled join index* [Miss82, Vald85]. A precompiled join index is an index which has been constructed on the domain of the join attributes. Therefore, a join of two relations, R and S, in these algorithms is done only by scanning the index and finding which tuple of R is joined with a tuple or a set of tuples of S. This is a very fast method. However, its maintenance cost is very high and this type of index may be very large. Such join algorithms are not considered here.

2.4. Hybrid Class of Algorithms

For these types of algorithms the join of relations is being replaced by the join of their semijoins [Ullm82]. The *semijoin* of relations R and S, written $R \ltimes S$, is $\pi_R(R \bowtie S)$, where π denotes the projection operation. Note that, in general, $R \ltimes S \neq S \ltimes R$. The expression $R \ltimes S$ selects those tuples of R that participate in the join of R and S.

To perform join using semijoin, the algorithm proceeds as follow:

1) Scan relation R and save its join attribute values in a data structure A.
2) Scan relation S and compare the join value of each tuple with the ones in A. In case of a match save the tuple in a temporary file S' and store its join value in a data structure B.
3) Scan relation R again and compare the join value of each tuple with the ones in B. In case of a match save the tuple in a temporary file R'.
4) Join the two files R' and S' using any of the existing algorithms proposed for the first class of join algorithms.

In this thesis, several semijoin algorithms which are based on preprocessing the *Partial-Relation* scheme are being presented. A Partial-Relation scheme is a projection of the join and restriction attributes of the queries. Furthermore, the values of an attribute in a Partial-Relation scheme are obtained by applying a transformation function to the original values. The objective of this transformation function is to reduce the size of the attribute values. Partial-Relations are suitable for processing *join-only* queries as well as queries involving selections, projections and joins. The Partial-Relation scheme is described in chapter 4.

Babb [Babb79] has implemented semijoin operations using boolean arrays. The idea is to hash the join attribute and then use the result as an address into the Boolean array. The presence of a marked bit in the array means that matching tuples exist. The value of the boolean arrays is the elimination of most of the data not needed in the result. Specialized hardware has also been proposed by Pramanik [Pram86c] to perform the semijoin operation. In order to support join as well as semijoin operations, the method is improved and adapted for multiprocessor database machines [Vald84, Shul84, Rich87]. Many specialized architectures have been proposed for high performance relational database management. These architectures include logic-on-disk machines [Smit75, Schu79, Su79], VLSI-based special purpose processors [Kits83, Shib84], and loosely- and tightly-coupled multiprocessor architectures [DeWi79, Gard81, Hsia83, Tera83, DeWi86]. In this thesis, only join algorithms for von Neumann architecture are being considered. This is because most of the algorithms proposed in the past as well as those discussed here may be modified for efficient implementation on various database machine architectures [Good80, Bitt83, Shul84].

CHAPTER 3

OPTIMAL ACCESS SEQUENCE FOR THE INDEX-SCAN CLASS OF JOIN ALGORITHMS

3.1. Introduction

In order to optimize the performance of the index-scan class of algorithms in a paging environment this research developed two graph models, namely the *block connectivity* and the *page connectivity* models. The term paging environment means that the unit of buffer size is a page. These models will help to show that the index-scan class of algorithms performs better than the other two classes of algorithms when the *join factor*† is low. The performance measurement is based on the number of pages of main memory which are needed to guarantee one access per block or page. The block connectivity model assumes that a *block* is a unit of secondary storage access, and the page connectivity model assumes a *page* is a unit of secondary storage access.

Pramanik and Ittner [Pram85b] have considered a similar problem however, their model is based on saving only the tuples within pages that participate in the join. They have shown that the problem of determining the least upper bound on the size of the buffer, when only the joining tuples are saved, is NP-hard. The least upper bound on the buffer size implies that there is at least one page access sequence that needs this much buffer, but no page access sequence needs any more than this. Merrett, *et al.*[Merr81] have considered the page scheduling which requires the least page-swapping counts. They have shown that this problem can be represented as a special case of the Hamiltonian path problem. Thus, it is shown to be NP-complete. Two sufficient conditions for the existence of the optimal solutions were shown, which are based on the

† The join factor for a relation R is defined to be the ratio of the number of tuples of R participating in the join operation over the number of tuples in R [Blas77].

Hamiltonian path condition and the Euler path condition. Using these conditions, Merrett, *et al.* have presented heuristic procedures for near optimum solutions. Sacco and Schkolnick [Sacc86b] have developed a general model of buffer management, called *hot set model*, which can be used to minimize the number of page fault rates for different buffer sizes.

Here, a model in which the entire page is saved in the buffer is considered as in Merrett, *et al.* [Merr81] to avoid reaccesses. Thus, guaranteeing only one access per page. The objective here is to determine an ordered list of blocks (or pages) which requires the smallest size buffer among all possible ordered lists of blocks (or pages). We refer to an ordered list of blocks (or pages) as a *block (or page) access sequence*. The following assumptions are made in the analysis:

1) A join of only two relations is considered. Note that an m-way join operations ($m > 2$) can be decomposed into a series of binary joins.

2) The database is disk resident. Therefore, the selected pages of the relations need to be saved in the main memory buffer before being processed.

3) A data page may contain tuples of more than one relation.

4) A block may contain the data pages of more than one relation.

The remainder of this chapter is organized as follows: in Section 3.2 the graph models are presented. In Section 3.3 the problem of determining a block access sequence which requires the least amount of buffer will be shown to be NP-hard. In Section 3.4 the problem of computing the least upper bound on the main memory buffer size for a given set of joining pages will be considered and will be shown to be an NP-hard problem. In Section 3.5 a heuristic procedure is given. In Section 3.6 the performance of the heuristic is compared against the optimal page access sequence. Finally, in Section 3.7 the performances of the fragmentation technique, the sort-merge join algorithm and an algorithm based on the graph model are compared.

3.2. The Graph Models

The index-scan algorithms use the indices on the join-participating attributes of the relations to determine a list of pointer pairs to the tuples that are to be joined. Thus, from each pointer pair one can determine the corresponding block pairs or page pairs that need to be accessed. Here, three types of *graph models* are developed from these pairs. These graph models are the page connectivity, the block connectivity and the tuple connectivity [Pram85b]. These models differ from each other by the unit of secondary storage access and the unit of buffer storage. Table 3.1 summarizes the differences between the three models.

Table 3.1. The Difference Between the Graph Models.

The Graph Models	Unit of Disk Access	Unit of Buffer Storage
Page Connectivity Model	Page	Page
Block Connectivity Model	Block	Page
Tuple Connectivity Model	Page	Tuple

3.2.1. Page Connectivity Model

For this model the assumption is that a pointer consists of a page number and an offset within the page. Here, the connectivity of all the pages in a join operation is represented by the *page connectivity graph.* Two nodes a and e of the page connectivity graph have an edge (a,e) between them if a tuple in the page corresponding to a joins with a tuple in the page corresponding to e. Figure 3.1 gives an example of a page connectivity graph.

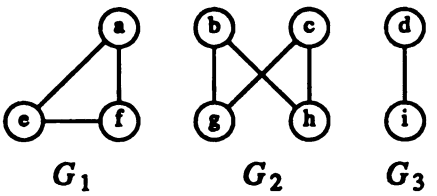


Figure 3.1. An Example of a Page Connectivity Graph.

A *component* of a graph G is defined to be the maximal connected subgraph of G. For example, in Figure 3.1 there are three components G_1, G_2 and G_3. Components are *disjoint*; that is, only the tuples within one component may need to be joined, independent of the tuples in other components. As a result, in order to avoid reaccessing the pages of the joining relations one needs to save, at most only the pages of a single component in the buffer. It has been shown by Pramanik and Ittner[Pram85b] that as the *selectivity factor* † increases, the number of components in the page connectivity graph approaches one very quickly (i.e., the graph becomes a connected graph). Note that this does not imply that as the selectivity factor increases the page connectivity graph becomes a *complete* graph. A graph is said to be *complete* only if every node has an edge with every other node.

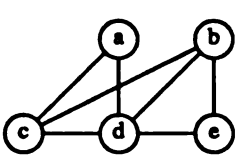


Figure 3.2. A Page Connectivity Graph.

Using the page connectivity graph concept, the problem of determining a page access sequence(PAS) which requires the least amount of buffer can be defined as

† Selectivity factor is defined to be the ratio of the number of tuples in the result of a join to the sum of the tuples in the joining relations.

follows:

Given: A page connectivity graph G(V,E), and a positive integer $K \leq |V|$, where V is a non-empty set of nodes, E is the set of edges of the graph G, and $|V|$ represents the number of elements in the set V.

Question: Does there exist a page access sequence which requires a buffer of size K or less?

The page access sequence which requires the least amount of buffer (i.e., smallest K value) is called the *Optimal Page Access Sequence* and is denoted by OPAS. Here, the problem of determining such an access sequence is referred to as the OPAS-problem. For example, for the page connectivity graph of Figure 3.2 there are 5! possible page access sequences. The idea here is to look for the page access sequence which requires the least amount of buffer. If the nodes of this graph are fetched in the order *abced* then the required buffer size is 5. On the other hand, if the pages are fetched in the order *acdbe* then the required buffer size is 3 as it is shown in Table 3.2.

Table 3.2. The OPAS for the Graph of Figure 3.2.

Content of the Buffer	Size of the Buffer
a	1
a,c	2
a,c,d	3
c,d,b	3
d,b,e	3

This is the smallest buffer size one can expect for this page connectivity graph. Thus, OPAS for this graph is *acdbe*. In Section 3.4 I show that the problem of computing the least upper bound for K is NP-hard. In Section 3.5 a heuristic procedure with $O(n^2)$ time complexity is given. The performance of this heuristic procedure is compared against the optimal page access sequence which has the complexity of $O(n!)$,

where n is the number of data pages in the page connectivity graph.

3.2.2. Block Connectivity Model

For this model the assumption is that a pointer consists of a block number and an offset within the block. Therefore, the unit of secondary storage access is a block and only the appropriate pages of a block are saved in the main memory. Two blocks are said to be connected if a data page in one is joined with a data page in another. The connectivity of all blocks in a join operation is represented by a *block connectivity graph.* Figure 3.3 gives an example of a block connectivity graph. The circles in the figure represent the data pages and the rectangles represent blocks. Note that if one page per block is assumed, then the block connectivity graph becomes a page connectivity graph.

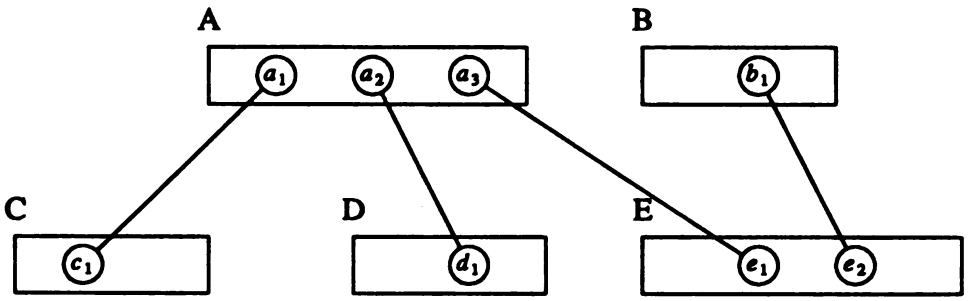


Figure 3.3. An Example of a Block Connectivity Graph $H(B,E)$.

The idea here is to look for a Block Access Sequence (BAS) which requires the least number of pages to be saved in the buffer. Such a sequence is referred to as *Optimal Block Access Sequence* (OBAS). The problem of determining such a block access sequence is referred to as the OBAS-problem. For example, in Figure 3.3 the block access sequence ABCDE requires 4 pages to be saved in the buffer while the sequence CADEB requires only 2 pages to be saved. An OBAS for this graph is *CADEB*. Table 3.3 shows the size of the buffer as blocks are fetched in the sequence CADEB.

Section 3.3 shows that the OBAS-problem is NP-hard. In Section 3.4 the problem of computing the least upper bound on the main memory buffer size for the block connectivity model is also shown to be NP-hard.

Table 3.3. The OBAS for the Graph of Figure 3.3.

Block Accessed	Content of the Buffer
C	c_1
A	a_2 , a_3
D	a_3
E	e_2
B	--

3.2.3. Tuple Connectivity Model

The tuple connectivity model was first presented in [Pram85b]. For this model, the assumption is that the unit of secondary storage access is a page as in the page connectivity model. However, the size of the buffer is determined in terms of the number of tuples. The problem associated with this model is how to determine for a given *tuple connectivity graph*, an ordered list of pages (PAST) which require minimum number of tuples to be saved in the buffer. Such a sequence is denoted as OPAST and the problem of determining such a sequence is referred to as the OPAST-problem. A *tuple connectivity graph* is an edge-weighted graph, where the weights represent the number of tuples of one page to be joined with the corresponding tuples in another page. This implies that the relations have no duplicate values on the joining domains. Figure 3.4 gives an example of a tuple connectivity graph. The OPAST for this graph is *ecdba* which requires a buffer storage of only 5 tuples as shown in Table 3.4.

When the join domains have duplicates, the tuple connectivity graph becomes a *directed* graph. The arc weights represent the number of tuples from the *tail* page to be

joined with tuples in the *head* page.

Table 3.4. The OPAST for the Graph of Figure 3.4.

Page Accessed	# of Tuples in the Buffer
e	5
c	5
d	3
b	4
a	0

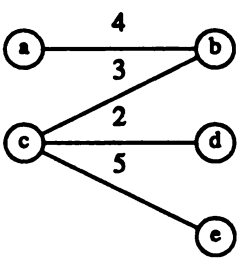


Figure 3.4. An Example of a Tuple Connectivity Graph.

As mentioned earlier, the problem of computing the least upper bound on the buffer size for this model (regardless of duplicate values on the joining domains) is NP-hard [Pram85b]. Next section shows that the OPAST-problem is also NP-hard.

3.3. The OBAS-problem and the OPAST-problem are NP-hard

In this section I show that a special case of the OBAS-problem is NP-hard. Therefore, the general OBAS-problem is NP-hard. This result is then used to show that the OPAST-problem is also NP-hard.

The special case of the OBAS-problem is defined as the problem of determining the OBAS for a block connectivity graph, such that, a page within a block of the graph may be joined with at most one other page of another block and no other page in any other block. Here, this problem is referred to as the OBAS1-problem. Figure 3.3 shows an example of such a block connectivity graph. Note that for this block connectivity graph, the degree of a node is the same as the number of pages in the block corresponding to that node. Next theorem shows that the OBAS1-problem is NP-hard.

Theorem 1. The OBAS1-problem is NP-hard.

Proof: I present a polynomial time reduction of a known NP-complete problem, *minimum cut linear arrangement*, to the OBAS1-problem. The minimum cut linear arrangement problem is defined as follows:

Given a graph $G(V,E)$ and a positive integer K, one has to decide if there exists a one-to-one function $f:V \rightarrow \{1,2,...,|V|\}$ such that for all i ,$1<i<|V|$,

$$|\{(u,v)\in E : f(u)\leq i < f(v)\}| \leq K \ .$$

This problem is known to be NP-complete[Gare79].

Given a graph $G(V,E)$, let $V=\{A_1,A_2,...,A_n\}$, and p_i be the degree of the node A_i, for $i=1,...,n$. We construct a block connectivity graph $H(B,E)$ from G where $B=\{A'_1,A'_2,\ldots,A'_n\}$ and each node $A'_i\in B$ is labeled with p_i. That is, $l(A'_i)=p_i$. In the block connectivity graph H, a node A'_i represents a block with p_i pages. Also an edge (A'_i,A'_j) in H for some $i\neq j$, represents a page in the block A'_i to be joined with a page in the block A'_j and no other page. This implies that each edge of the graph represents a join between two distinct pages of their corresponding blocks. Therefore, given a graph $G(V,E)$ one can construct a block connectivity graph $H(B,E)$ from G.

Now I show that for a minimum cut linear arrangement of nodes in G there exists a corresponding linear arrangement of blocks in the block connectivity graph H constructed from G, such that the value of K for a sequence in G is the same as the buffer

size for the corresponding sequence in H. Let $A_1A_2 \cdots A_n$ be the sequence in the graph $G(V,E)$,with $|V|=n$, such that for all i , $1<i<n$

$$\xi_i = |\{(u,v)\in E : f(u)\le i<f(v)\}| \le K \ .$$

ξ_i denotes the number of edges from nodes $A_1,A_2,\ldots,A_i$ to the nodes $A_{i+1},\ldots,A_n$ for $1<i<n$. Therefore, one can represent ξ_i as follows:

$$\xi_i = \sum_{j=1}^{i} p_j - \sum_{j=1}^{i}\sum_{\substack{k=1 \\ k\neq j}}^{i} X_{jk} \ , \tag{1}$$

where $X_{jk} = 1$ if $(A_j,A_k)\in E$, otherwise $X_{jk} = 0$. The first term in (1) represents the sum of the degree of the nodes $A_1,\ldots,A_i$, and the second term represents the total number of edges among the nodes $A_1,\ldots,A_i$. Now I show that the size of the buffer for the block access sequence $A'_1A'_2 \cdots A'_n$ is K.

Let γ_i, for $i>1$, be the number of pages need to be saved in the buffer when block A'_1 is in the buffer and the blocks $A'_2, A'_3,\ldots,A'_i$ are fetched. Note that this is the required buffer size prior to fetching block A'_{i+1}. Also note that when a block is accessed, all the pages of that block are brought into the buffer, and only those pages of the block that are needed for the future joins are kept in the buffer. Suppose two consecutive blocks A'_i and A'_j of a block access sequence are accessed, then the number of pages that need to be saved from these two blocks is

$$l(A'_i) + l(A'_j) - Y_{ij} - Y_{ji} \ ,$$

where $Y_{ij} = Y_{ji} = 1$ if $(A'_i,A'_j)\in E$, otherwise $Y_{ij} = Y_{ji} = 0$. The first two terms in the above formula represent the total number of pages in the blocks A'_i and A'_j, respectively, and the last two terms represent the total number of pages that need to be joined and removed from the buffer.

In general, when A'_1 is in the buffer and the next $i-1$, for $i>1$, blocks of the access sequence $A'_1A'_2 \cdots A'_n$ are fetched, the value of γ_i can be represented as follows:

$$\gamma_i = \sum_{j=1}^{i} l(A'_j) - \sum_{j=1}^{i} \sum_{\substack{k=1 \\ k \neq j}}^{i} Y_{jk} \qquad (2)$$

where $Y_{jk} = 1$ if $(A'_j, A'_k) \in E$, otherwise $Y_{jk} = 0$. Since the label on a node of H represents the degree of the node, one can replace $l(A'_j)$ in (2) with p_j. Also, since the edge set E is the same for both graphs G and H, and $B \equiv V$, then $Y_{jk} = X_{jk}$. Therefore, $\gamma_i = \xi_i \leq K$ for $i = 2,...,n$. This implies that the required buffer size for the block access sequence $A'_2 A'_3 \cdots A'_n$ is equal to K. Now I show that if the block access sequence $A'_1 A'_2 \cdots A'_n$ is an optimal sequence, then the required buffer size for this sequence is also equal to K.

If the buffer size for the sequence $A'_1 A'_2 \cdots A'_n$ is considered, one of the following three cases may occur:

Case 1) $l(A'_1) \geq 1$, $l(A'_2) \geq 1$, and $(A'_1, A'_2) \notin E$. For this case (refer to Figure 3.5a), when block A'_2 is accessed, the size of the buffer is $\xi_2 = \gamma_2 = l(A'_1) + l(A'_2)$. Therefore, $l(A'_1) < \xi_i = \gamma_i \leq K$, for $i > 1$. This implies that the required buffer size for the sequence $A'_1 A'_2 \cdots A'_n$ is K.

Case 2) $l(A'_1) \geq 1$, $l(A'_2) > 1$, and $(A'_1, A'_2) \in E$. For this case (refer to Figure 3.5b), when block A'_2 is accessed, the size of the buffer is $\xi_2 = \gamma_2 = l(A'_1) + l(A'_2) - 2$. Since $l(A'_2)$ should have at least a value of 2, then the size of the buffer for the sequence is equal to K, where $K \geq \xi_i = \gamma_i \geq l(A'_1)$.

Case 3) $l(A'_1) \geq 1$, $l(A'_2) = 1$, and $(A'_1, A'_2) \in E$. For this case (refer to Figure 3.5c), when block A'_2 is accessed, the size of the buffer is $\gamma_2 = l(A'_1) + l(A'_2) - 2 = l(A'_1) - 1$, which is less than $l(A'_1)$. If $l(A'_1) \leq K$ then K is the buffer size for the sequence. However, when $l(A'_1) > K$ then the buffer size for the sequence is $l(A'_1)$. But for this case I show that there exist another block access sequence which requires a buffer of size less than $l(A'_1)$. This implies that $A'_1 A'_2 \cdots A'_n$ is not an

optimal sequence. We obtain a block access sequence which requires less buffer than the sequence $A'_1A'_2 \cdots A'_n$ by exchanging only the ordering of the nodes A'_1 and A'_2 in the sequence. For this new sequence, $A'_2A'_1 \cdots A'_n$ the required buffer size is $l(A'_1)-1$ (i.e., $K = \gamma_2 = \xi_2 = l(A'_1)-1$).

Therefore, one can determine the sequence in $G(V,E)$ which gives minimum cut (i.e., minimum K value) by determining the block access sequence which requires the minimum size buffer in $H(B,E)$. □

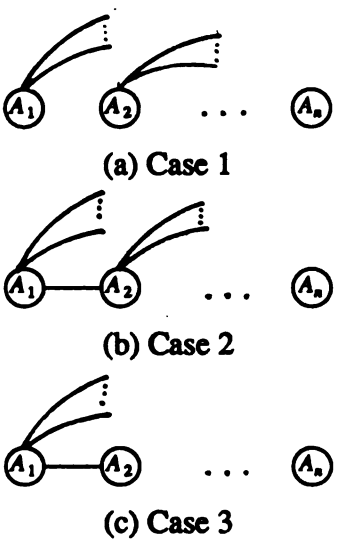


Figure 3.5. The Cases for when the Buffer Size for $A'_1A'_2 \cdots A'_n$ is Computed.

Following is an example of the minimum cut linear arrangement problem for a given graph and the construction of the block connectivity graph from it. Figure 3.3 shows the block connectivity graph, $H(B,E)$, constructed from the graph,$G(V,E)$, of Figure 3.6. In the graph of Figure 3.6, for K=4, the minimum cut linear arrangement can be defined by the function f as follows:

$$f(A')=1,\ f(B')=2,\ f(C')=3,\ ,f(D')=4,\ \text{and}\ f(E')=5$$

Table 3.5 shows the value of ξ_i for i=2,3,4.

Table 3.5. Value of ξ_i for the Graph of Figure 3.6.

i	ξ_i
2	$\|\{(A',C'),(A',D'),(A',E'),(B',E')\}\|=4$
3	$\|\{(A',D'),(A',E'),(B',E')\}\|=3$
4	$\|\{(A',E'),(B',E')\}\|=2$

Given any ordered sequence of nodes in graph G one can replace every node in the sequence with their corresponding nodes of graph H. For example, the sequence $A'B'C'D'E'$ of the graph G can be represented by the block access sequence $ABCDE$ in graph H. Note that K value for the sequence in G is the same as the buffer size for the corresponding sequence in H. For the sequence $A'C'D'E'B'$ in G the value of K is equal to 2. However, the corresponding block access sequence $ACDEB$ requires a buffer of size 3. Therefore, as mentioned in the proof of Theorem 1 (case 3), by exchanging the ordering of A and C in the above sequence we obtain the sequence $CADEB$ which requires a buffer of size K=2.

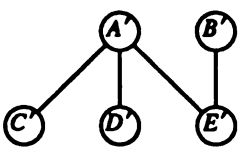


Figure 3.6. A Graph $G(V,E)$.

In the next theorem I use the result obtained in Theorem 1, to show that a special case of the OPAST-problem, call it OPAST1-problem, is also NP-hard. Therefore, the general OPAST-problem is NP-hard. The OPAST1-problem is defined as the problem of determining OPAST for a tuple connectivity graph, such that, there is only one tuple in each page of the tuple connectivity graph that needs to be joined with a tuple in another page. Therefore, the weights on the edges of the corresponding tuple connectivity graph are all 1's. For example, consider the graph of Figure 3.6. An edge (A',C') in this graph represents the need for joining a tuple in A' with a tuple in C'. Note that the weights on the edges are all 1s and not shown in the figure. Here, the case where the values of the joining attributes are unique is considered. Therefore, when page A', in Figure 3.6, joins with pages C',D' and E', there are three tuples with unique join values in A'. Now I show that the OPAST1-problem is NP-hard.

Theorem 2:. The OPAST1-problem is NP-hard.

Proof: I present a polynomial time reduction of the OBAS1-problem to the OPAST1-problem. Note that here when I refer to a block connectivity graph I mean the graph which was considered for the OBAS1-problem. Let a block with p join participating pages in a block connectivity model represents a data page with p join participating tuples in a tuple connectivity model. The join of two blocks in a block connectivity model is then represents the join of two tuples in their corresponding pages of the tuple connectivity model. Therefore, given a block connectivity graph, one can use the above transformation to construct a tuple connectivity graph with weights of all edges assumed to be 1. Now given a block access sequence in the block connectivity graph, it is easy to see that the required buffer size, in term of the number of tuples need to be saved, is the same for the corresponding page access sequence in the tuple connectivity graph. Therefore, one can determine the optimal block access sequence in a block connectivity graph by determining an optimal page access sequence in the corresponding tuple connectivity graph when the edges have weights of 1. □

3.4. Complexity of Computing the Least Upper Bound on the Buffer Size

In this section I show that the problem of computing the least upper bound for the block connectivity model, denoted by LUBB, is NP-hard. Let us assume that the size of a block in a block connectivity graph is one. Then the block connectivity graph is the same as a page connectivity graph. Therefore, the page connectivity model is a special case of the block connectivity model. To show that the LUBB problem is NP-hard, I will first show that the problem of determining the least upper bound for the page connectivity model, denoted by LUBP, is NP-hard. The LUBP problem is defined as follows:

Given: A page connectivity graph $G(V,E)$ and a positive integer $K \leq |V|$.

Question: Does there exist a page access sequence which requires a buffer of size $\geq K$?

Note that by a page access sequence I mean a sequence where a page is accessed only once. I will show that the LUBP problem is NP-hard. Before proving this the following definitions and propositions are needed.

A *cut* of the graph $G(V,E)$ is defined to be two disjoint subsets of nodes X and Y, represented by (X,Y), such that $X \cup Y = V$. Figure 3.7 shows a possible cut of the page connectivity graph of Figure 3.2; where $X=\{b,e\}$, and $Y=\{a,c,d\}$.

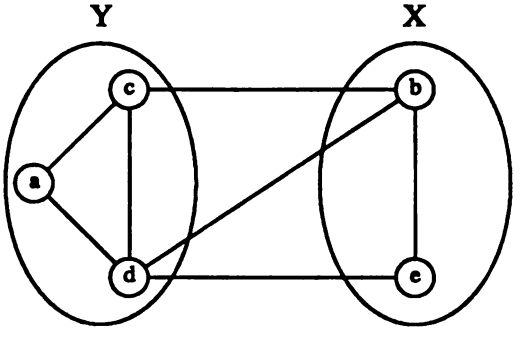


Figure 3.7. A Possible Cut of the Graph of Figure 3.2.

The *frontier* nodes for a subset of the cut, say X, are defined to be the set $\{x : (x,y) \in E;$ *where* $x \in X$ and $y \in Y\}$. I define the *non-frontier* nodes for a subset of the cut, X, to be the set $\{x : (x,y) \in E;$ *where* $x \in X$ and $y \in X\}$. Let F_X and NF_X represent the set of frontier and non-frontier nodes in the subset X of the cut, respectively. Note that $F_X \cup NF_X = X$. In Figure 3.7, $F_X = \{b,e\}$, $NF_X = \phi$, $F_Y = \{d,c\}$, and $NF_Y = \{a\}$. A *prefix* of a page access sequence is any number of leading nodes of the sequence. A possible page access sequence for the page connectivity graph of Figure 3.2 is *abcde*, and a prefix for this sequence is *abc* which has a *length* of 3.

Proposition 1. Let (X,Y) be a cut of the page connectivity graph G(V,E) and X,Y be a page access sequence which has the nodes in subset X of the cut as prefix (regardless of ordering) followed by the nodes in subset Y of the cut (regardless of ordering). Then X,Y page access sequence requires at least a buffer of size of $|F_X|+1$.

Proof: After fetching the pages of the subset X of the cut, regardless of the ordering, the only pages remaining in the buffer are the pages in the set F_X. In order to remove any one of these pages from the buffer one needs at least a node from the set F_Y. Therefore, the smallest size of buffer needed is $|F_X|+1$. □

For example, in Figure 3.7, let Y,X be the page access sequence *cadbe*. When all the nodes in the subset Y of the cut (i.e., $c,a,$ and d) are fetched, the only nodes remaining in the buffer would be the nodes in the set F_Y which are c and d. In order to remove any of the nodes c or d from the buffer one needs at least one node from the set F_X. Therefore, the buffer size is at least 3.

Proposition 2. Let $a_1 a_2 \cdots a_n$ be a page access sequence of the page connectivity graph $G(V,E)$ with n nodes. Also, let S_i, for $1 \le i \le n$, be the required buffer for a prefix of length i of the page access sequence. Then, for every S_i, for $1 \le i \le n$, there exists a cut (X,Y) of G.

Proof: I need to show that there exists a cut (X,Y) of G such that the required buffer size when all the nodes of a subset of the cut, X, are fetched is $|S_i|+1$. The cut can be

constructed as follows: place the nodes in a prefix of length i of the page access sequence in the subset X of the cut and the remaining nodes of the sequence in the subset Y of the cut. According to Proposition 1 the required buffer size when all the nodes of the subset X of the cut are fetched is $|S_i|+1$. □

Corollary: There is a one-to-one relationship between a cut of a page connectivity graph and the buffer size required for a prefix of a page access sequence.

Proposition 3. For a page connectivity graph $G(V,E)$ and a cut (X,Y) of G when the nodes in NF_X (or NF_Y) are placed into Y (or X) the size of the F_Y(or F_X) will increase by 1.

Proof: Here I show that by placing a node from the set NF_Y to X then the size of F_X will increase by 1. Let us assume the there is an edge $(a,b)\in E$ such that $a\in NF_Y$ and $b\in F_Y$. Then placing the node a into the set X causes a becomes a frontier node of X. Therefore, $|F_X|$ increase by 1. □

For example, in Figure 3.7, by placing node a of the subset Y into the subset X of the cut the size of the set F_X increases by 1. In the next two propositions I show that maximum $|F_X|$ gives the least upper bound on the size of the buffer. For example, in Figure 3.7, if the two nodes a and c from the subset Y are placed in subset X of the cut, then F_X has the maximum size. Therefore, the least upper bound on the size of the buffer for the page connectivity graph of Figure 3.2 is $|F_X|+1 = 4+1 = 5$.

Proposition 4. Let (X,Y) be a cut of the page connectivity graph G(V,E). Then $F_X \cup F_Y = |V|$ when $|F_X|$ is maximum.

Proof: Our assumption is that $|F_X|$ is maximum. I will show the following:

1) $NF_Y=\phi$. If $NF_Y\neq\phi$, then placement of some of the nodes in NF_Y into the subset X of the cut will increase $|F_X|$ by at least one, which is a contradiction.

2) $NF_X=\phi$. If $NF_X\neq\phi$, then placing all of the nodes in NF_X into the subset Y of the cut, may cause one of the following two events to occur:

a) $|F_Y|$ increases and $NF_Y=\phi$. For this case one might have $|F_Y|>|F_X|$ which requires switching the labeling of the two subsets of the cut.

b) $|F_Y|$ increases and $NF_Y\neq\phi$. In this case I will place the nodes in NF_Y into the cut subset X. According to proposition 3, this will increase the size of F_X by at least one which is a contradiction.

Note that by placing the pages in the set NF_X into the set Y, the size of the $|F_X|$ can not be decreased. □

Proposition 5. Let G(V,E) be a page connectivity graph G(V,E). Then the least upper bound on the size of the buffer is max($|F_X|$)+1; where max gives the maximum value of F_X.

Proof: This follows directly from Proposition 1. □

The following theorem shows that the *LUBB* problem is NP-hard.

Theorem 3. The problem *LUBP* is NP-hard.

Proof: I present a polynomial time reduction of a known NP-complete problem to the *LUBB* problem. The *dominating set* problem is formulated as follows:

Given a graph $G(V,E)$ and a positive integer $K \leq |V|$ one has to decide if there exists a dominating set of size K or less for G, i.e., a subset $V' \subseteq V$ with $|V'| \leq K$ such that for all $u \in V-V'$ there is a $v \in V'$ for which $(u,v) \in E$.

K is called the *domination number* of the graph G if K is minimum among all possible dominating sets for G. The dominating set problem is known to be NP-complete[Gare79]. Suppose that the nodes of the graph G were data pages of the page connectivity graph and the edges in graph G were the edges in the page connectivity graph. Then one can determine the upper bound on the buffer size for the page connectivity graph by determining the domination number of the graph. □

Theorem 4. The problem LUBB is NP-hard.

Proof: Since the problem of computing the least upper bound for the page connectivity model is NP-hard, and page connectivity model is a special case of the block connectivity model, then the LUBB is NP-hard. □

3.5. A Heuristic Algorithm for the Page Connectivity Model

Since the problem of determining the OPAS for a given page connectivity graph is a complex combinatorial optimization problem here, a heuristic procedure with $O(n^2)$ time complexity is presented; where n is the number of the nodes in the page connectivity graph. This heuristic algorithm determines a page access sequence which requires a near optimal buffer size. This algorithm is efficient when the join factor is high. For small problem size, I have compared the performance of the heuristic algorithm with a branch and bound algorithm presented in the next section. The comparison shows that the average buffer size for the heuristic method is very close to optimal (i.e., the buffer size differs from optimal buffer size only by 2 or 3 pages). Some definitions are needed before I present the heuristic method.

The *resident-degree* of a node v, written $\delta_{res}(v)$, is defined to be the number of *adjacent* nodes of v that are in the buffer. A node u is said to be *adjacent* to node v if there is an edge between them. The *nonresident-degree* of a node v, written $\delta_{nor}(v)$, is defined to be the number of the adjacent nodes of v that are not in the buffer. Therefore, the *degree of a node* v is $\delta(v) = \delta_{res}(v)+\delta_{nor}(v)$. Now the heuristic algorithm is described.

Step 1. Initialize the buffer to empty, $\delta_{res}(v)=0$ and $\delta_{nor}(v)=\delta(v)$ for all $v \in V$.

Step 2. Add to the buffer a node with the *smallest degree*. Update the resident-degree and nonresident-degree of its adjacent nodes.

Step 3. Add to the buffer a node with the *largest* δ_{res}. If there is more than one node then choose the one with the *smallest* δ_{nor}. Update the resident-degree

and nonresident-degree of the nodes adjacent to the node added to the buffer.

Step 4. Perform join among the nodes (data pages) in the buffer.

Step 5. Remove from the buffer the nodes having all of their adjacent nodes in the buffer.

Step 6. If the buffer is empty then STOP, otherwise go to step 3.

Note that steps 2 and 3 each requires one scan of the list of nodes. Therefore, the complexity of this algorithm is in the order of $O(n^2)$; where n is the number of data pages in the graph. Table 3.6 gives the trace of this algorithm for the page connectivity graph of Figure 3.2. Note that the page access sequence is *ebdca*.

I have applied the above heuristic algorithm to random page connectivity graphs of n nodes and e edges. For each value of n, 20 random graphs with e edges were created. The buffer size for each graph is then computed. Figure 3.8 shows the average buffer size as a function of the *edge factor* for n=50, 100 and 200. The *edge factor* for a graph with n nodes is defined to be the ratio between the number of edges in the graph (i.e., e), and total possible number of edges between the n nodes, which is $\frac{n*(n-1)}{2}$. In this figure the assumption is that the edge factor can have values up to 1 (i.e., a complete page connectivity graph). However, there are cases in which the value of edge factor is much less than one. For example, if n pages of relation R need to be joined with m pages of the relation S with the assumption that these pages are disjoint then the maximum value for edge factor would be $\frac{m*n}{\frac{(m+n)*(m+n-1)}{2}} < 1$.

Table 3.6. Trace of the Heuristic Algorithm for the Graph of Figure 3.2.

Content of the Buffer	Pages to be Joined	Size of the Buffer
e	---	1
e,b	(e,b)	2
b,d	(e,d),(b,d)	2
b,d,c	(b,c),(c,d)	3
d,c,a	(d,a),(c,a)	3

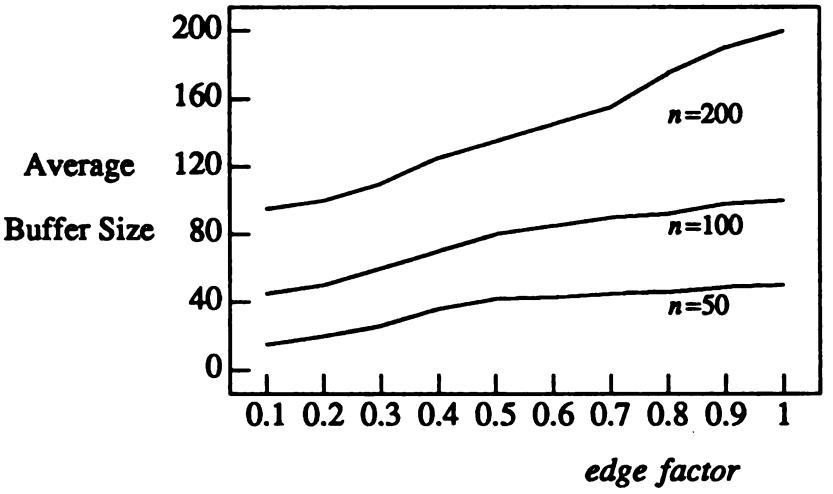


Figure 3.8. Average Buffer Size for the Heuristic Algorithm.

3.6. Determining an Optimal Page Access Sequence

In this section I first give a branch and bound algorithm to determine the optimal page access sequence. The complexity of this algorithm is in the order of $O(n!)$. I then compare the performance of the heuristic procedure, given in the previous section, with that of the branch and bound algorithm.

A tree organization is being used to facilitate the search for optimal sequence. The branch and bound algorithm chooses the smallest buffer size among all possible *paths* which start from the root and terminate at leaves. A *path* of length m from node v to node w in a graph G is an edge sequence from v to w of length m in which the edges are

distinct. The complexity of this algorithm, called it algorithm A, is $O(n!)$, where n is the number of nodes in the page connectivity graph. Before I give the algorithm some definitions are needed.

Given the page connectivity graph G(V,E) with n nodes, and an order list of nodes $v_1,...,v_n$ let

$$\alpha_i=\{v_k : (v_k,v_p)\in E \text{ for } 1\le k\le i-1 \text{ and } i\le p\le n\} \quad \text{for } 2\le i\le n-1.$$

Then the size of the buffer for a prefix of length i is $\xi_i=|\alpha_i|+1$; where, $|x_i|$ represent the size of the set x. The size of the buffer for a given page access sequence is now can be defined to be $\max\{\xi_i : i=2,...,n-1\}$. Let the *feature buffer size* for a prefix of length i of a given page access sequence to be $|\alpha'_i|+1$, where

$$\alpha'_i=\{v_k : (v_k,v_p)\in E \text{ for } 1\le k\le i-1 \text{ and } i+1\le p\le n\} \quad \text{for } 2\le i\le n-1.$$

Note that to determine if a node v belongs to the set α_i one needs to scan $n-i$ nodes. However, to determine if a node belongs to the set α'_i one needs to scan only $n-i-1$ nodes. Thus, by using α'_i instead of α_i in determining the OPAS one can reduce the expansion of the search tree by one level. In the algorithm given below, I first find an ordered list of nodes with the smallest feature buffer size and then determine its buffer size. Now I give the algorithm.

Step 1. Consider all the nodes of the graph and generate their children. Determine the feature buffer size for all the paths from the roots to their corresponding leaves.

Step 2. Find the path from root to a leaf with the smallest size feature buffer size and let B be the last node in the path. If there are more than one such path, then choose longest.

Step 3. Generate the children of the node B, and determine the feature buffer size for the new paths generated.

Step 4. If *B* has only one child then goto step 5; otherwise go to step 2.

Step 5. Compute the buffer size for the path started from the root and ended at the child of *B* and then STOP.

Figure 3.9 gives the trace of algorithm A for the page connectivity graph of Figure 3.2. The number on a node of the graph represents the feature buffer size for a path started at the root and ended at that node. The dotted edges in the graph represents the optimal page access sequence, which is the sequence *acdbe*. The size of the buffer for this sequence is 3. As mentioned earlier, this algorithm is practical only for small problems. I have compared the average buffer size required for a random graph of 5 up to 10 nodes for both the heuristic method and algorithm A. Figure 3.10 shows the result of this comparisons. As seen from the figure, the heuristic algorithm determines a page access sequence of the random graph which requires an average buffer size very close to the optimal.

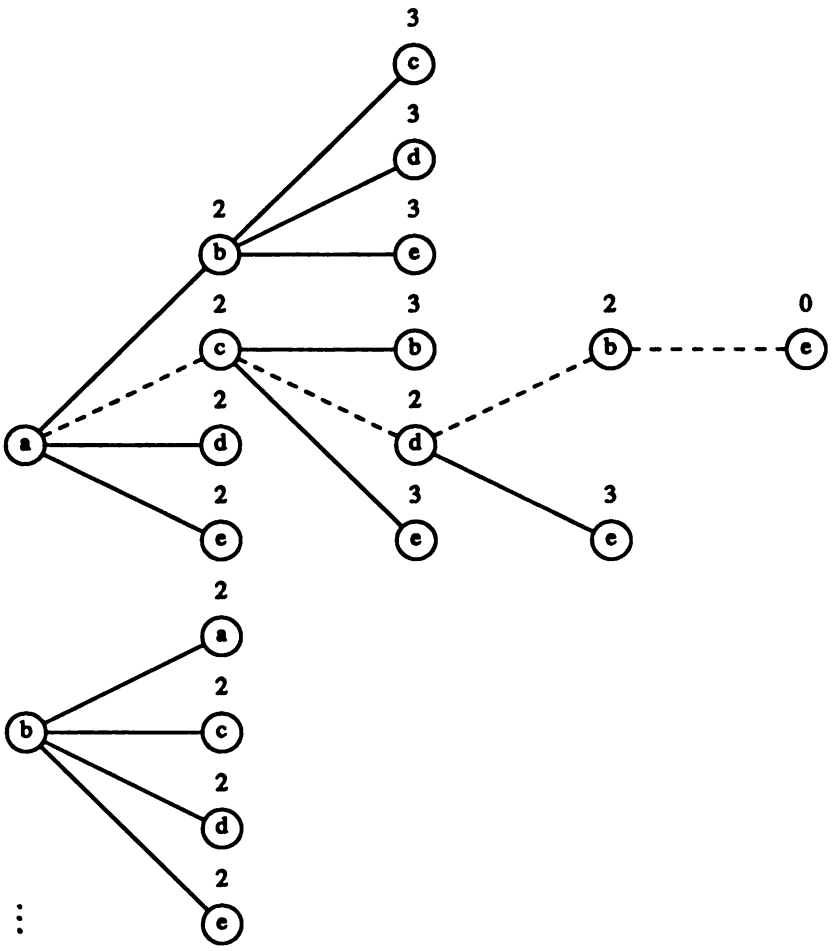


Figure 3.9. Trace of Algorithm A for the Graph of Figure 3.2.

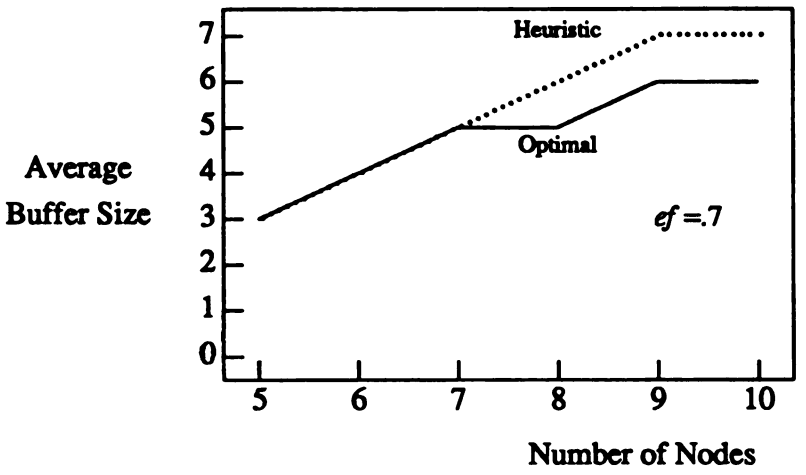


Figure 3.10. Average Buffer Size for the Heuristic and the A Algorithms.

3.7. Cost Models and Performance Comparisons

In this section the performances of the fragmentation technique [Sacc86a], the sort-merge join algorithm [Blas77] and an algorithm based on the page connectivity model are compared. This last algorithm, called it algorithm P, uses the heuristic method given in Section 3.5 to determine the near optimal access sequence. The following notations are used to derive the cost of each algorithm:

R_i Relation i

N_i Number of tuples in relation i

E_i Average number of tuples from R_i in a data page

L Average number of (key value, pointer) pairs in a leaf page of an index

M Number of pages of available main memory buffer space

P_i Join factor for relation i

Now I briefly describe each algorithm and compute its I/O cost.

3.7.1. Cost of the P Algorithm

For two relations R_1 and R_2, this algorithm, proceeds as follows:

1) Scan the indices on the join attributes of both relation in order to obtain a set of pointer pairs. The cost for this step is $\frac{N_1+N_2}{L}$.

2) Use the heuristic algorithm given in Section 3.5 to determine the near optimal page access sequence. Our assumption here is that enough memory is available to access the desired pages without any reaccesses. Therefore, the cost for this step is zero.

3) Fetch the pages according to the sequence obtained in step 2 and perform the join. The cost for this step is $\tau(P_1,N_1,E_1)+\tau(P_2,N_2,E_2)$; where $\tau(p,n,e)$ is the expected number of pages which must be accessed in order to fetch pn tuples of a relation containing n tuples distributed evenly, e per page. $\tau(p,n,e)$ has been derived in [Good80] and is given below:

$$\tau(p,n,e) = \begin{cases} n/e & \text{if } 1-e/n \le p \le 1 \\ n/e.\left(1-\dfrac{\binom{n-e}{p.n}}{\binom{n}{p.n}}\right) & \text{if } p < 1-e/n \end{cases}$$

3.7.2. Cost of the Fragmentation Technique

Fragmentation computes the natural join of two relations by recursively partitioning two unordered relations into fragments of size less than or equal to $M-1$ pages. The partitioning is done using a hashing function. The problem of performing join between two relations R_1 and R_2 is then reduced to the problem of joining the corresponding fragments of each relation. The cost of partitioning R_i into $M-1$ fragments is $2\times H_i$, where $H_i=\frac{N_i}{E_i}$, for $i=1,2$. In order for each fragment of the smaller relation, say R_2 to

fit in the buffer, partitioning can be recursively applied to each fragment produced by the previous phase. Therefore, phase k produces $(M-1)^k$ fragments. The size of the kth fragment is $\lceil \frac{H_2}{(M-1)^k} \rceil$. The total number of partitioning phase is thus $k \times \lceil \log_{m-1} H_2 \rceil - 1$, and the total cost of partitioning relation R_i is $2 \times H_i (\lceil \log_{m-1} H_2 \rceil - 1)$. The total cost of fragmentation joins is:

$$2H_1 (\log_{M-1} H_2) + 2H_2 (\log_{M-1} H_2) - H_1 - H_2 \ ;$$

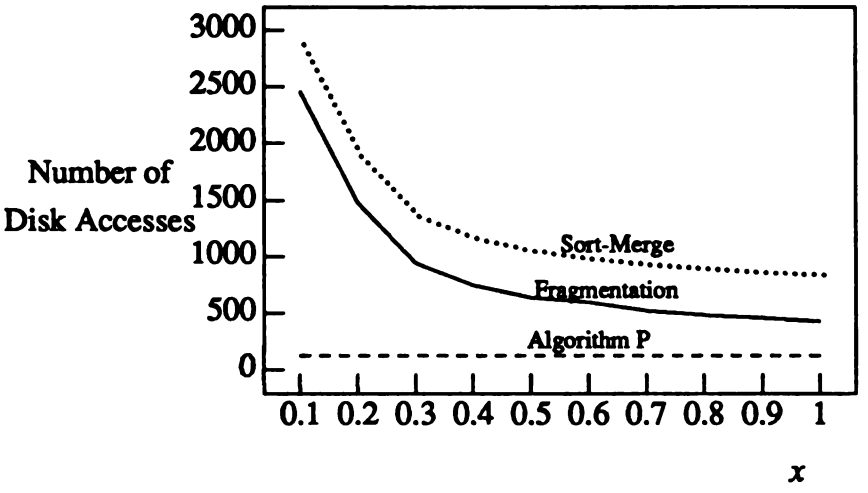
3.7.3. Cost of the Sort-Merge Algorithm

This algorithm requires the initial sorting and multiway merging of both joining relations on their join attributes, and finally merging of the two sorted relations. The cost for this algorithm is:

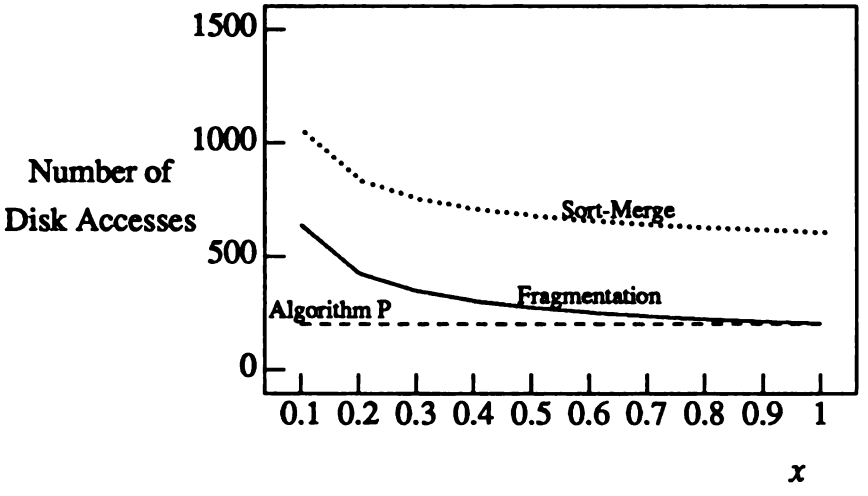
$$H_1 + 2 \times (\log_{M-1} H_1) + H_2 + 2 \times (\log_{M-1} H_2)$$

3.7.4. Cost Comparisons

Figure 3.11 shows the cost of the three algorithms as a function of x for two different join factors. x is defined to be the ratio of the buffer size to the number of pages that participate in the join (i.e., $x = \frac{M}{\tau(P_1,N_1,E_1)+\tau(P_2,N_2,E_2)}$; for $0<x\leq 1$. It is valid to assume that for every x there exist a graph with $\tau(P_1,N_1,E_1)+\tau(P_2,N_2,E_2)$ number of nodes which requires a buffer of size M (refer to Figure 3.8). The parameter values considered here are N_i=10000, E_i=100 and L=200 for i=1,2. As shown in Figure 3.11a, when the join factor is low, algorithm P outperforms the other two algorithms. However, for join factors of 0.5, only for $x \leq 0.5$ algorithm P performs better than the other two algorithms. When $x > 0.5$ algorithm P and the fragmentation technique perform almost the same because for algorithm P almost all the pages of the joining relations have to be fetched.



(a) Join Factors P_1=P_2=0.1.



(b) Join Factors P_1=P_2=0.5.

Figure 3.11. Comparisons of the Sort-Merge, Fragmentation, and P Algorithms.

CHAPTER 4

JOIN AND SEMIJOIN ALGORITHMS BASED ON THE PARTIAL-RELATION SCHEME

4.1. Introduction

An index on attribute *A* of relation *R* permits rapid access to a single tuple that has a desired value in that attribute. However, for *multi-attribute* queries involving join and restrictions several indices may need to be accessed. Hashing is also another method for fast access to the desired tuples of a relation. In addition this technique has been used for multi-attribute search [Burk76,Ullm82] or join-only queries [DeWi84, Shap86]. However, one may need to scan several hash tables in order to process a query involving restrictions and join. In [Tana79] a new access method called a *Relational Inverted Structure* (RIS) has been proposed which is a combination of indexing and hashing techniques. An RIS maintains the cross references of the joining attributes as well as the hash values of the remaining attributes of the relations. However, an RIS may be needed for every set of attributes corresponding to the same domain. Furthermore, no extensive analysis has been done on their performances. File compression techniques have also been proposed for partial match retrieval [Sack83]. These techniques are not suitable for the join operations because the join values have to be compressed uniformly across the files. Thus, coding of join attribute values require global statistics which is more difficult to apply for file compression. Here we propose a new access path to a relation, called a Partial-Relation scheme, or PR in short, in order to speed up the relational join operations.

A *Partial-Relation* scheme is a projection of the relation on the join and restriction attributes of the queries [Pram86a, Foto88]. Further, the values of an attribute in a Partial-Relation scheme are obtained by applying a transformation function to the original values. The objective of this transformation function is to reduce the size of the

attribute values. As a result, the amount of storage required by a Partial-Relation is also reduced. Known transformation functions such as hashing and or *compression* techniques may be used to reduce the attribute values. Note that a Partial-Relation is created based on the statistics of the queries. As I will show, the PRs facilitate the implementation of queries involving restrictions and join.

Two join algorithms are presented for the index-scan and the hybrid classes. These algorithms preprocess the Partial-Relations first and then join the selected tuples of the relations. The performances of these algorithms are compared with the sort-merge and the hybrid-hash join algorithms (two known best algorithms of the relation-scan class). The analysis is based on the cost of accesses to the secondary storage and the CPU usage. It has been shown that for wide range of *restriction factors*† and/or join factors the proposed algorithms perform better than the sort-merge and hybrid-hash join algorithms. We have considered join-only queries and queries involving restrictions, projections and join.

The organization of this chapter is as follows: Section 4.2 describes the Partial-Relation schemes. In Sections 4.3 and 4.4 the join and the semijoin algorithms based on Partial-Relation schemes are presented. The performances of these algorithm are compared with those of the sort-merge and the hybrid-hash.

4.2. An Access Path Based on a Partial-Relation Scheme

In this section I first describe the Partial-Relation Scheme and then discuss the advantages and disadvantages of using PRs in performing join and restrictions over using multiple secondary indices or hashing techniques.

---

† Restriction factor for a relation is defined to be the ratio between the number of tuples in the relation that satisfy the restrictions and the size of the relation.

4.2.1. Description of Partial-Relation Schemes

Let $R(A_1,A_2,\cdots,A_n)$ be a relation scheme with n attributes, $A_1,A_2,\cdots,A_n$, such that the first k attributes of R are participating in the joins or restrictions. Let $|R|$ be the total number of tuples in R, and $//R//$ be the amount of storage required by R. A Partial-Relation scheme for R, or PR(R), is defined to be a relation scheme with $k+1$ attributes where the first k attributes are those of R and the $(k+1)$th attribute is an access attribute, or AA, which points to the corresponding tuple in R. The attribute values in PR(R) are derived by applying a Value-Reducer function which, for example, can select a few characters out of all the characters of an attribute value. Here, a set of Value-Reducer functions, one for each attribute of the PR, is defined. Figure 4.1 gives an example of a Partial-Relation scheme for the EMPLOYEE relation.

Relation EMPLOYEE

rec.#	ename	address	telephone	dept	manager
1	Buckner,J.	162 Haslett,E.Lan	332-3860	Pharmacy	Brown,D.
2	Buckman,J.	299 Cleveland,E.Lan	227-1759	Shoe Repair	Smith,A.
3	Buckman,P.	160 Kalamazoo, E.Lan	335-2343	Photography	Clare,R.
4	Buckle,P.	452 Garfield,E.Lan	353-1759	Pharmacy	Brown,D.
5	Bucklin,J.	684 Cedar,E.Lan	350-1775	Photography	Clare,R.

Value-Reducer functions:
F(ename)=H(manager)=The first three characters of the attribute values.
G(dept)=The first two characters of the attribute values.

PR(EMPLOYEE)

ename	dept	manager	AA
Buc	Ph	Bro	1
Buc	Sh	Smi	2
Buc	Ph	Cla	3
Buc	Ph	Bro	4
Buc	Ph	Cla	5

Figure 4.1. An Example of a Partial-Relation.

The problem of determining optimal set of attributes for a PR is similar to the secondary index selection problem [Come78] which is known to be a NP-complete

problem. Low cost heuristics for near optimal solution have been given in [Schk75]. These algorithms are based on the statistical properties of the queries. To add a new attribute value to an existing PR one may scan the original relation and then create a new PR by using the Value-Reducer function to the values of the new set of attributes. I show in Appendix A that the cost of creating an index is more than the cost of creating a PR.

Processing a query using PRs is done in two phases. In the first phase, one performs restriction, and join or semijoin operations on the PRs first. This produces a set of pointers to the resulting tuples of the original relations. In the second phase, final join and restriction operations are performed on these resulting tuples. The PRs contain partial values and as a result operations on them may produces a few *false tuples* † . Therefore, at the end of the first phase a super set of the tuples to which the join operation should be applied is obtained. The cost of processing a query is the sum of the cost to preprocess the PRs, and the cost to process the resulting tuples. Cost of preprocessing PRs depends on $//PR//$. In order to minimize the overall cost a set of Value-Reducer functions is defined which

a) Minimizes $//PR//$, and at the same time

b) decreases the number of false tuples.

In general, these two objectives conflict with each other. By defining an appropriate set of Value-Reducer functions one may be able to avoid the problem of condition (b). However, this may not satisfy condition (a). Note that allowing false tuples in a PR may be advantageous in reducing the size of the PR. As we will show in Section 4.4.1, for a small percentage of false tuples (up to 40%) the performance of the proposed algorithms remain almost the same if the restriction factor is low. The number of false tuples

† False tuples are those tuples which seem to satisfy the query when viewed in some aspects but actually are not qualified tuples [Lum70].

depends on the ratio between the number of unique attribute values in a PR and the original relation. An analytical model is developed to compute this ratio, M, for N randomly generated unique attribute values. The ratio is given by

$$M=\frac{A^X}{N}.\left[1-\frac{\binom{A^L-A^{L-X}}{N}}{\binom{A^L}{N}}\right]$$

where A is the alphabet size and X characters of each value of length L are selected by a Value-Reducer function. The derivation of this expression is given in Appendix B. To validate the analytical result, we have computed the percentage of false tuples (i.e, $1-M$) for 1000 randomly selected persons' names of up to 20 characters. I have also computed the percentage of false tuples for 600 randomly selected telephone numbers of 7-digit long. The result of these experiments as well as that of the analytical model is given in Figure 4.2. The graph of the figure shows that the percentage of false tuples approaches 0 rapidly with an increasing value of X. Thus, by choosing only a few characters of the attribute values one can guarantee a few false tuples. The graph also shows that the percentage of false tuples for experimental data is less than that of the analytical model. This is because the adjacent characters of the experimental data are statistically dependent. The effect of a set of Value-Reducer functions on $//PR//$ is also analyzed in [Pram85a, Pram86b].

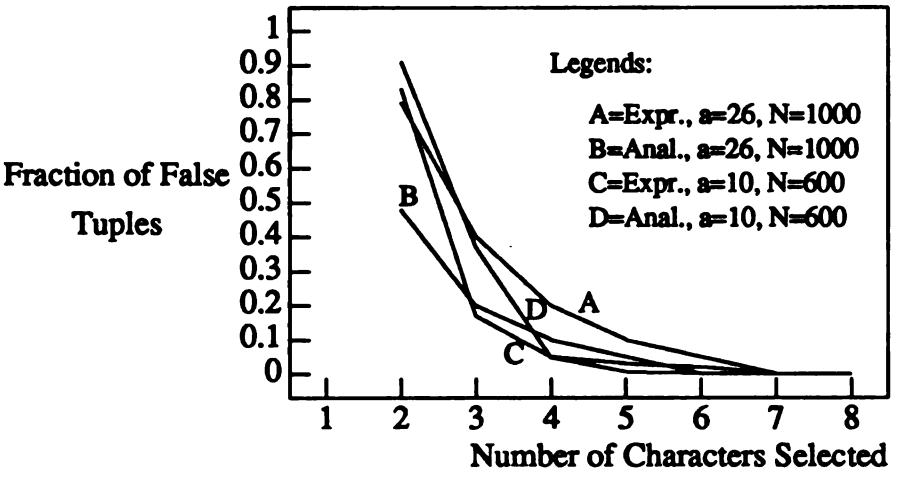


Figure 4.2. Number of Characters Selected vs. Fraction of False Tuples.

4.2.2. Partial-Relation Scheme Vs. Indexing and Hashing

As mentioned earlier, the primary function of the PRs is not to perform an efficient restriction on them as it is in conventional indexing, but to perform an efficient join. Therefore, for simple restriction queries with a very low restriction factor it is advantageous to use conventional indexing or hashing techniques. This is because PRs are scanned sequentially even for a low restriction factor. In general, PRs are suitable for applications where complex queries are processed. To process a query of this type, one needs to scan only one PR per relation. However, in methods using secondary indices, one may need to scan several indices and perform an intersection to obtain the set of keys or pointers to the relation.

Lum [Lum70] proposed a compound indexing scheme in order to retrieve all the tuples satisfying a partial-match query, in one access. A compound indexing scheme consists of a set of secondary index files with a common set of index fields in each file, but each file is ordered on different index fields. Therefore, the implementation of compound indexing requires a lot of storage. On the other hand, if a serial filing method is used, only one index file is created for all the attributes to be indexed. General retrieval through this index file, however, may require a large number of accesses because the

index is scanned sequentially. Note that PR is a special form of a serial filing method. However, the size of a PR is much smaller than that of an index because each value in PR is reduced by a Value-Reducer function. Therefore, sequential scanning of a PR is not as costly as sequential scanning of an index in a serial filing method. At the same time, because of the reduction, false tuples may result in processing a PR.

To process join-only queries with low join factors it is more cost effective to preprocess the PRs than to use the hashing techniques suggested in [DeWi84, Shap86]. This is because in hash-based join methods the original relations are scanned several times while preprocessing PRs requires at most one scan of the original relations. Further, the preprocessing also reduces the cost of the second phase considerably because the size of the relations are reduced due to low join factors. On the other hand, for very high join factor preprocessing PRs is not as effective as the hash-based join algorithms. Since PRs are centralized resources, the degree of inter- and intra-query concurrency is decreased as compared with methods using multiple secondary indices. However, the cost of updating a single PR is much less than maintaining several indices.

The overall storage requirements for a database will increase when PRs are used. However, the size of a PR is much less than the corresponding indices because of duplicate set of tuple pointers which appear in each index. Further, the storage utilization for a PR is close to 100% (a PR does not have to be ordered), where as in B-tree the average storage utilization is about 69%.

4.3. Algorithms Based on the Partial-Relation Schemes

In this section two algorithms for a relational equijoin operation between two relations, R and S are presented. The first algorithm, PRJ, is based on joining PRs, and the second algorithm, PRS, performs semijoin between the two PRs.

The performances of the above algorithms are analyzed using the following form of the query:

PROJECT (JOIN (RESTRICT(R), RESTRICT(S)))

4.3.1. Algorithm PRJ of the Index-Scan Class

This algorithm first applies the restrictions on the PRs and then join the selected tuples of the two PRs. The result of this join is a set of AA-value pairs. Using these AA-values, the tuples of the original relations are fetched and joined. The details of this algorithm are as follows:

a) Apply restrictions and projections on PR(R) by scanning it sequentially. The projection is done only on the join and the access attributes. Store the result in a file R'. Perform similar operations on PR(S) and store the result in a file S'.

b) Join R' and S' and obtain a set of AA-value pairs which point to the joining tuples of R and S. Joining of R' and S' is done by first sorting them and then merging the two sorted files.

c) Fetch the tuples of R and S using these AA-values. Then perform the final restriction, join and projections on these tuples.

4.3.2 Algorithm PRS of the Hybrid Class

This algorithm uses the semijoin and sort-merge techniques to process the above query. The semijoin operation is first performed on the PRs in order to reduce the size of the relations. The reduced relations are then joined using the sort-merge technique. For the semijoin operation a hashing function as well as Boolean arrays are used.

PRS algorithm is good for the join-only queries. Also, as explained in Section 4.4.1, for general queries given above, the PRS algorithm performs better than the PRJ algorithm for high restriction or join factors. This is because the PRS algorithm reduces the relations by means of semijoin operation and not by using AA-values. On the other hand, the PRJ algorithm uses the AA-value pairs and, therefore, for high restriction factors the number of accesses may be more than the number of pages in the relations. The

details of the PRS algorithm are as follows:

a) Apply the restrictions on the PRs and perform the semijoin on them. Store the result of this semijoin in a bit array.

b) Scan relation R, and for each tuple hash its join attribute value, and probe the bit array for a match. In case of a match store the subtuple in a temporary file R'. Create S' from S in the same way as R'. Note that R' and S' contain the entire length of key values rather than partial key and AA-values, as for the previous algorithm.

c) Join R' and S' by first sorting them on their join attribute values and then merging the two sorted files.

4.3.3. Cost Model

Here we develop a cost model based on the number of disk accesses for the proposed join algorithms. In this model the cost of eliminating duplicates from the output and forming the output relation is ignored because this cost is similar for all the algorithms. The following set of parameters are used for the model. These parameters for i=1,2, where R_1 corresponds to R and R_2 corresponds to S, are as follows:

N_i Number of tuples in R_i. N_i also represents the number of tuples in PR(i) because $|R_i| = |PR(i)|$.

E_i Average number of tuples from R_i in a data page

P_i Join factor for PR(i) or R_i (i.e., the fraction of tuples of PR(i) or R_i that participate in the equijoin)

M Main Storage space in term of page frames that are available (e.g. sort buffers)

J_i Restriction factor for PR(i) or R_i

f_i Ratio between the size of the PR(i) and R_i; where $f_i<1$

Now I derive the cost expression in terms of the number of disk accesses for the two algorithms PRJ and PRS.

4.3.3.1. Cost Expression for Algorithm PRJ

Here, the cost expression for the algorithm PRJ is derived for the following three cases:

I) The PRs are sorted and the original relations are clustered with respect to the join participating attributes.

II) The PRs are sorted but not clustered.

III) The original relations are not clustered, or if they are it is with respect to some attributes which are not participating in the join operation. Note that here we are not assuming that PRs are sorted on the join attributes.

For the first case one does not need to create R' and S'. This is because while the two PRs are scanned for restrictions, join can be performed at the same time. As a result the cost for step (b) of the algorithm is zero. The cost for scanning PR(R) and PR(S) is $\omega = \sum_{i=1}^{2} f_i N_i / E_i$. The cost to fetch tuples of R and S is $\nu = \sum_{i=1}^{2} \tau(J_1 J_2 P_i, N_i, E_i)$.

Therefore, the total cost is: $\omega+\nu$. The cost for case II is the same as above except that the value of ν changes to $\sum_{i=1}^{2} J_1 J_2 P_i N_i$. The cost for case III is the sum of the following three costs:

1) The cost to scan PR(R) and PR(S) is ω ;

2) The cost to create, sort and merge R' and S' which is zero if R' and S' fit in main memory, otherwise, it is

$$\rho 1 = \sum_{i=1}^{2} f_i J_i N_i / E_i (2 \log_{M-1} (f_i J_i N_i / (2(M-1)E_i) + 1)) \ .$$

The standard sort-merge algorithm, called SM, is being considered here. This

algorithm begins by creating the sorted runs of the tuples. These runs are then merged using an $(M-1)$-way merge. Each run on the average is twice as long as the number of tuples that can fit into a priority queue in memory[Knut73].

3) The cost to fetch the tuples of R and S is ν.

Thus, the total cost is:

ω+ν if R' and S' fit in main memory,

ω+ρ1+ν otherwise.

4.3.3.2. Cost Expression for Algorithm PRS

The assumption, for the cost expressions derived here, is that the bit array, which contains the result of the semijoin operation, will fit in the main memory. The cost of the PRS algorithm is ω+μ+δ; where $\mu=\sum_{i=1}^{2} N_i/E_i$, the cost to scan the relations, and

$$\delta=\sum_{i=1}^{2} 2J_1J_2P_iN_i/E_i(\log_{M-1}(J_1J_2P_iN_i/(2(M-1)E_i)))+J_iN_i/E_i \ ,$$

the cost to create, sort, and merge R' and S'.

4.4. Performance Evaluation

In this section, performances of the proposed algorithms are compared with the sort-merge join algorithm. This comparison is done under the assumption that large main memory buffer is not available. I also compare the performances of PRS algorithm with the hybrid-hash algorithm for the join-only queries with the assumption that large main memory buffer is available.

4.4.1. Comparisons with the Sort-Merge Join Algorithm

Here I compare the performances of the sort-merge algorithm with the proposed algorithms under the conditions that we consider typical. These conditions are:

A) Restriction indices exist and the relations are not clustered.

B) The same as condition *A* but the relations are clustered with respect to the join attributes.

C) There are no indices and the relations are not clustered.

Condition B is considered because it allows an efficient implementation of the join operation.

The parameter values that are used for the cost model in the figures are :

$$500<N_1=N_2<10^6\ ,\ P_1=P_2=1.0\ ,\ E_1=E_2=40\ ,\ M=30\ ,\ f_1=f_2=0.1$$

As mentioned in section 4.2, one of the characteristics of the set of Value-Reducer functions is to decrease the number of false tuples and, therefore, make the restriction factor for a PR very close to that of the corresponding relation. Note that this may increase the size of the PR. Figure 4.3 shows the cost of algorithm PRS as a function of percentage of false tuples for various restriction factors. As shown in the figure, for restriction factors of 0.1 and 0.01, the performance of the algorithm remain the same for up to 40% of false tuples in a PR. However, for restriction factor of 0.5, the cost of the PRS algorithm slightly increases as the percentage of false tuples increases by 10%. Here in the analysis I have assumed that the restriction factors for both a relation and its corresponding PR are the same.

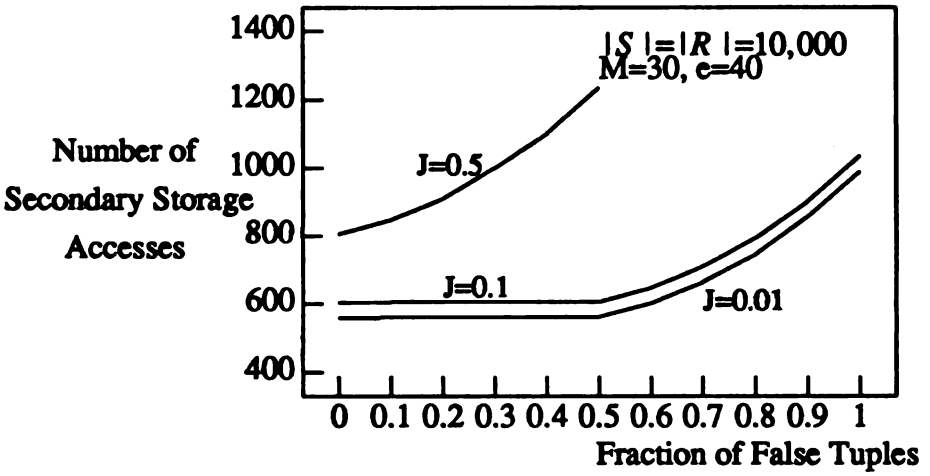


Figure 4.3. Performance of Algorithm PRS When PRs Contain False Tuples.

Both PRJ and PRS can be used under the above conditions. However, as mentioned in Section 4.3, in general, algorithm PRS performs better than the PRJ algorithm for high restriction factors while algorithm PRJ performs better for low restriction factors. This is shown in Figure 3.4 for condition A, assuming the relation sizes of 10^6. For condition C the result is the same as for condition A. The performances of the SM algorithm is also given in the figure. It is seen that the PRJ performs better for most restriction factors. But for very low and very high restriction factors the SM costs less. For high restriction factor PRS performs better than the PRJ and SM. The cost expression of the SM algorithm under the above three conditions is given in Appendix C.

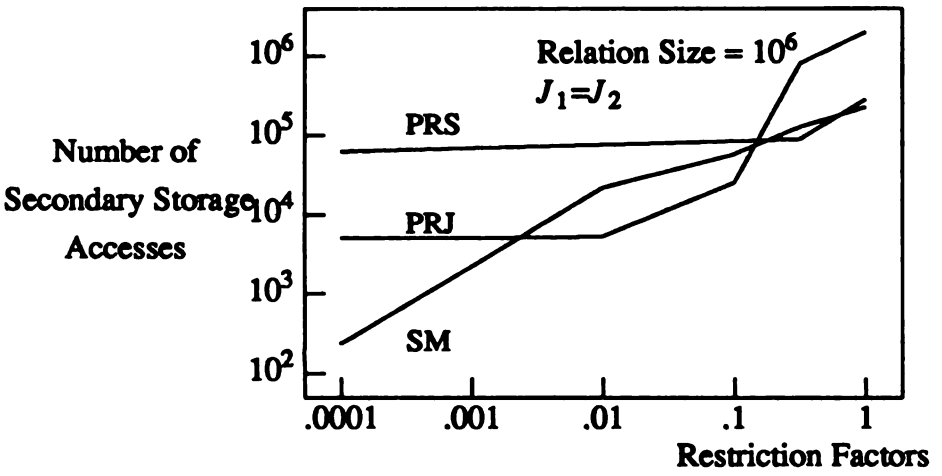
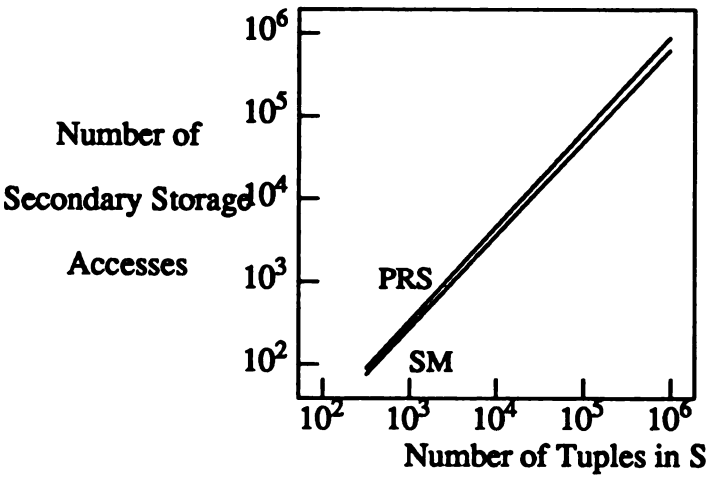


Figure 4.4. Comparisons Under Condition A for Different Restriction Factors.

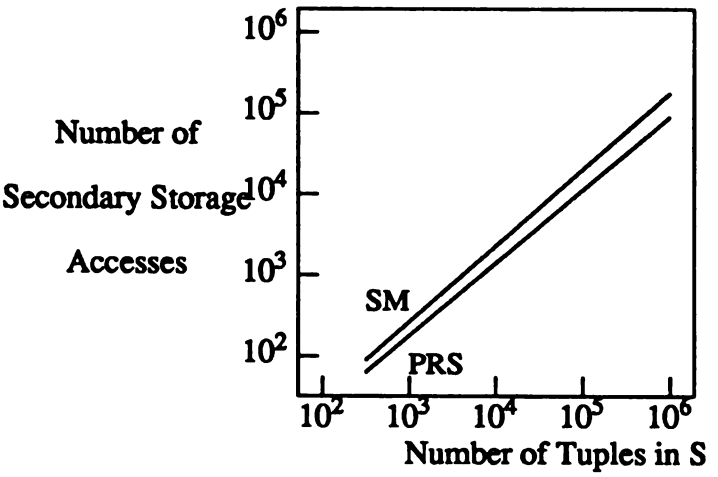
4.4.1.1. Interpretation of Results

The cost of evaluating the general query using algorithms PRJ, PRS, and the SM under conditions A through C are graphically shown in Figures 4.5 through 4.7. Each graph in a figure gives the number of disk accesses as a function of relation size. I have considered four different restriction factors. These values are 1.0, 0.5, 0.1 and 0.01 as shown separately in four graphs of each figure. In the figures only one of the PRS and PRJ algorithm, which has the minimum cost, is considered.

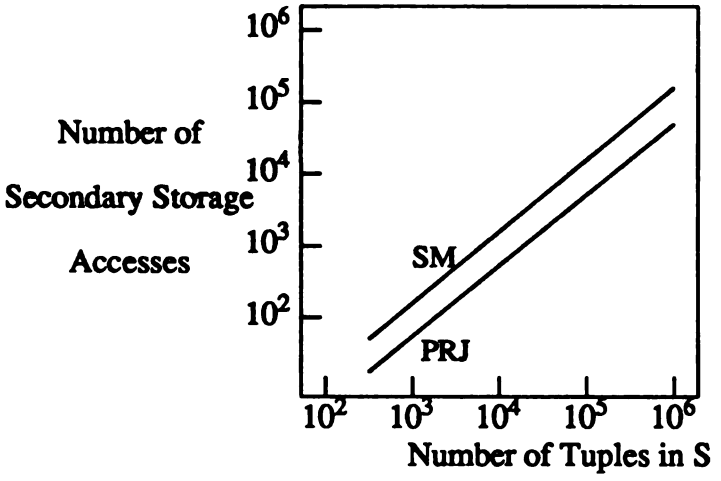
Figures 4.5a and 4.5b illustrate the cost of the SM and the PRS under condition A when restriction factors are 1.0 and 0.5. As shown in the figure, for restriction factor 1.0 algorithm PRS performs almost the same as the SM. This is because, for very high restriction factors, preprocessing does not reduce the size of the relations and, therefore, the cost of joining them remains almost the same as that of the SM algorithm. As the restriction factors decrease (e.g., $J_1=J_2=0.5$) algorithm PRS performs better than the SM algorithm. For restriction factors 0.1 and 0.01, algorithm PRJ performs much better than both the SM and the PRS algorithm. This is shown in Figures 4.5c and 4.5d. The improvement is due to the fact that preprocessing reduces the number of accesses to the relations and, therefore, it reduces the cost of processing in the second phase.



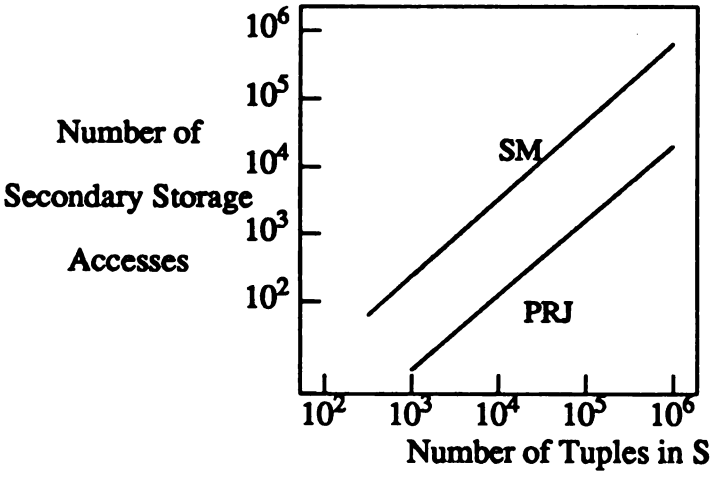
(a) Restriction Factors J_1=J_2=1.0.



(b) Restriction Factors J_1=J_2=0.5.



(c) Restriction Factors J_1=J_2=0.1.



(d) Restriction Factors J_1=J_2=0.01.

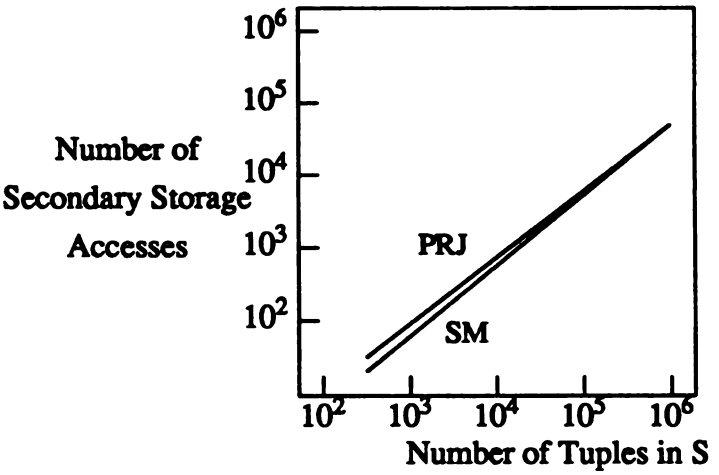
Figure 4.5. Comparisons Under Condition A.

Figure 4.6 illustrates the cost of the SM and the PRJ algorithms under condition B. For restriction factors 1.0 and 0.5, the SM algorithm performs better than the PRJ algorithm. This is because the relations are merged in the SM algorithm by scanning them only once, without using the indices. However, the PRJ algorithm, in addition to scanning the relations, also requires to scan the PRs. On the other hand as the restriction factor decreases the PRJ algorithm performs better than the SM algorithm. This is because the cost of the second phase decreases rapidly as a result of preprocessing the PRs.

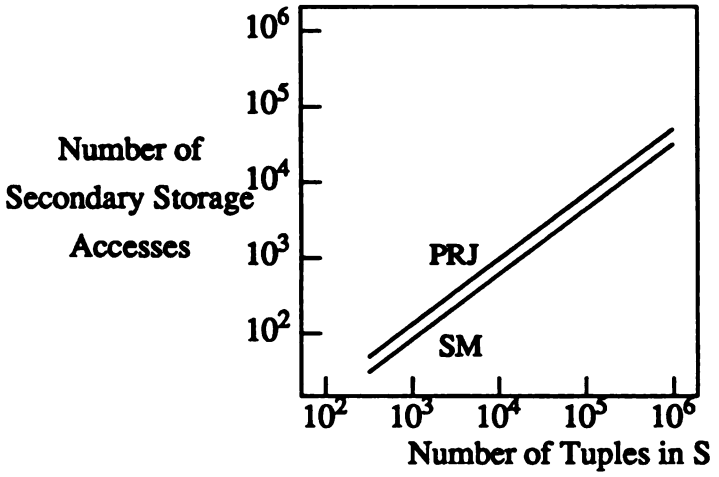
For condition C, algorithm PRS performs better than the SM algorithm for restriction factor 0.5. The result is shown in Figure 4.7. For low restriction factors (e.g. 0.1 and 0.01) algorithm PRJ performs better than the SM algorithm due to the lower cost of the second phase.

I have also compared the cost of the three algorithms for the range of the PR sizes. Figure 4.8 shows the effect of the different PR sizes on the cost of the PRJ algorithm for relation of size 10^5. As shown in the figure, for very high (e.g., 0.5) and very low (e.g., 0.001) restriction factors the SM performs better than the PRJ algorithm. For example, the PRJ algorithm has to scan the entire PRs regardless of the restriction factors, while the SM algorithm can use the indices to access the tuples.

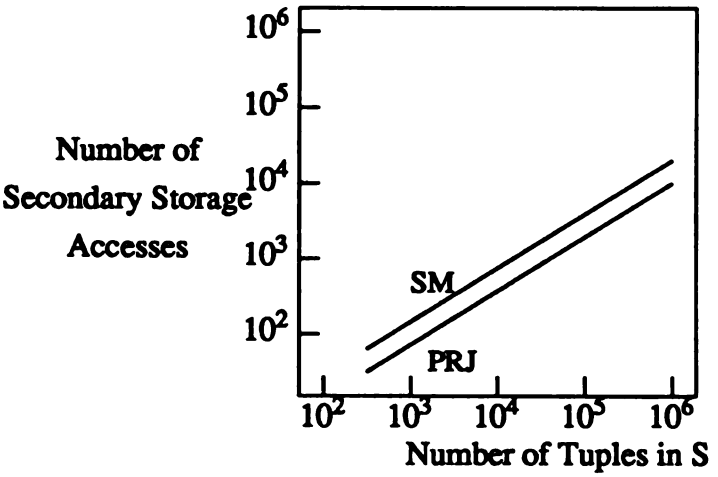
I have also done similar analysis for the memory and the page sizes in the range $20 \leq M \leq 50$ and $20 \leq E_1 = E_2 \leq 40$, respectively. We have observed similar results as discussed above.



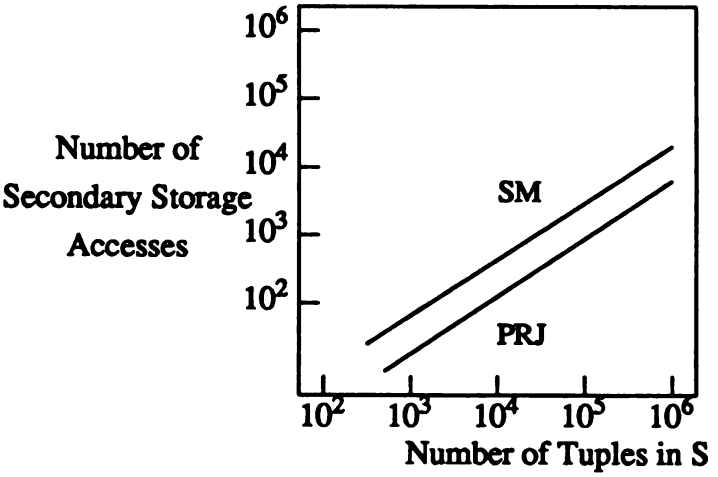
(a) Restriction Factors J_1=J_2=1.0.



(b) Restriction Factors J_1=J_2=0.5.

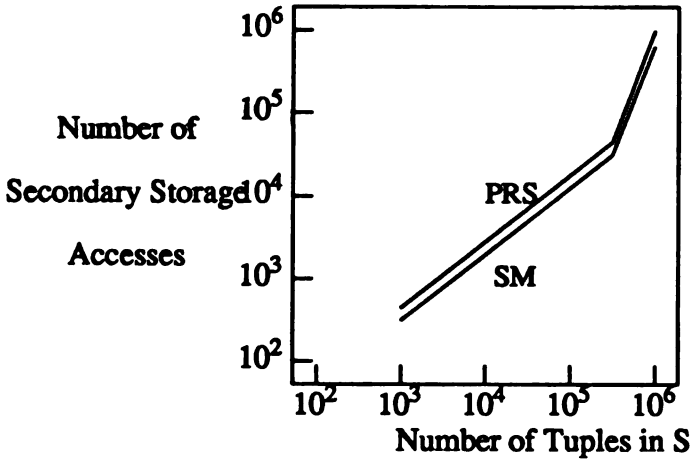


(c) Restriction Factors J_1=J_2=0.1.

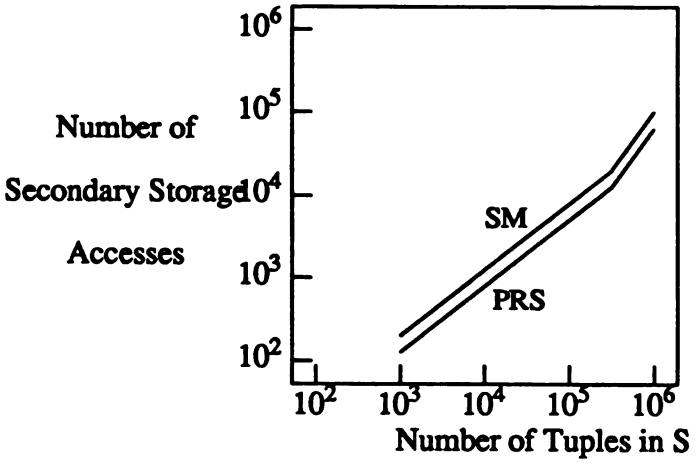


(d) Restriction Factors J_1=J_2=0.01.

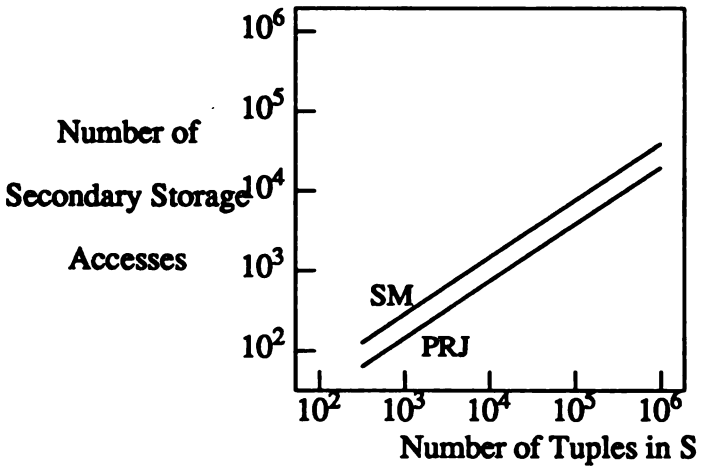
Figure 4.6. Comparisons Under Condition B.



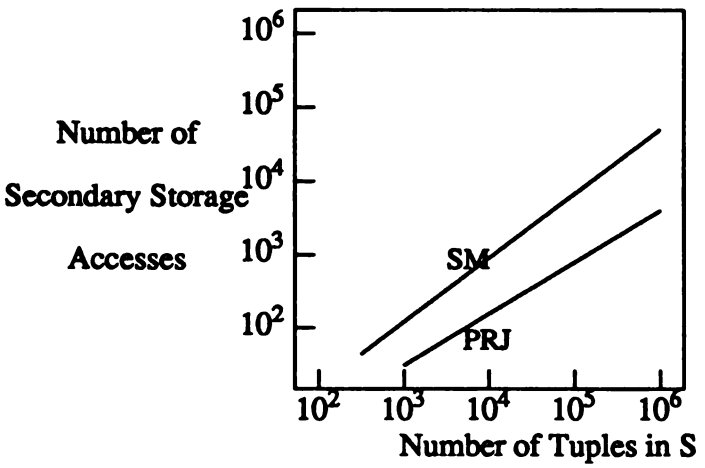
(a) Restriction Factors J_1=J_2=1.0.



(b) Restriction Factors J_1=J_2=0.5.

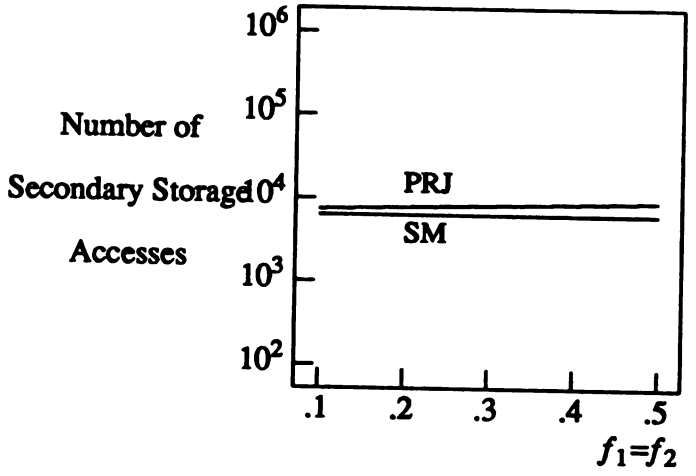


(c) Restriction Factors J_1=J_2=0.1.

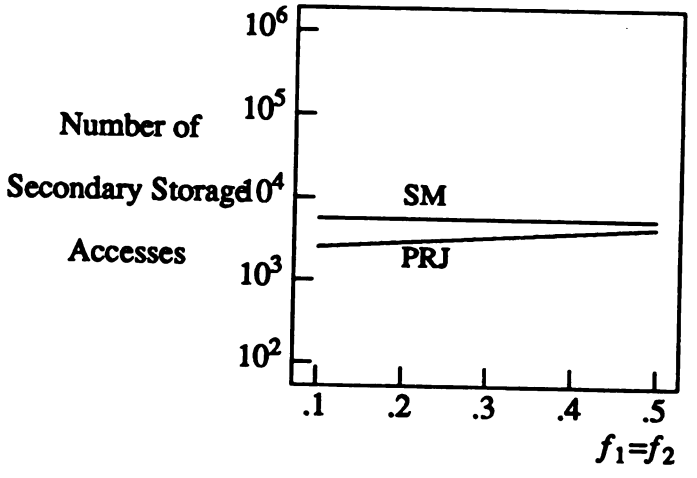


(d) Restriction Factors J_1=J_2=0.01.

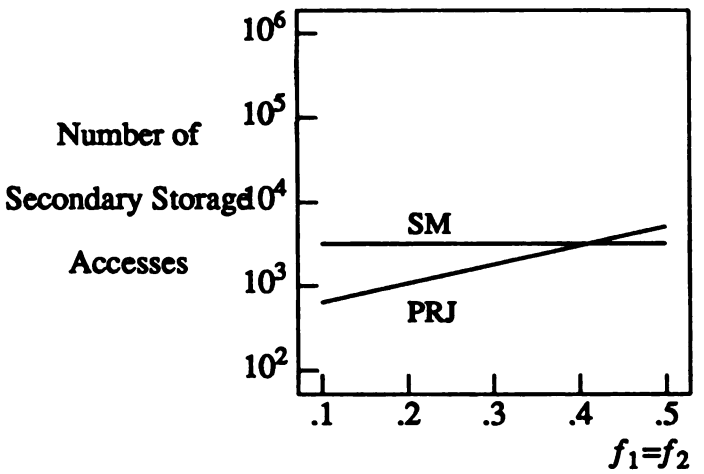
Figure 4.7. Comparisons Under Condition C.



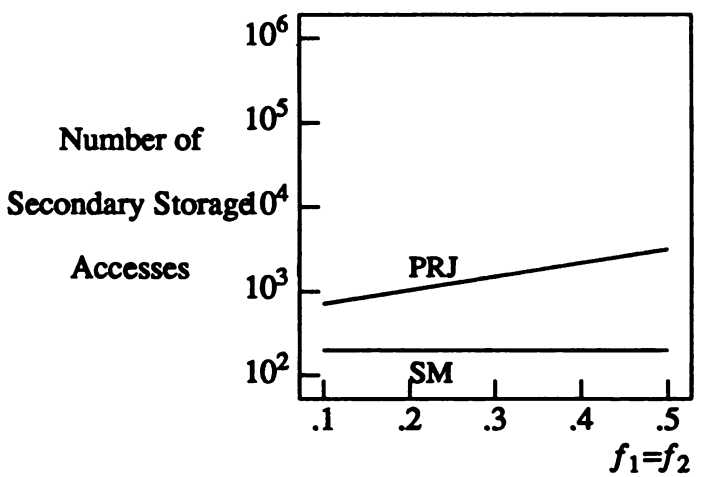
(a) Restriction Factors $J_1=J_2=0.5$.



(b) Restriction Factors $J_1=J_2=0.1$.



(c) Restriction Factors $J_1=J_2=0.01$.



(d) Restriction Factors $J_1=J_2=0.001$.

Figure 4.8. Comparisons Under Condition B for Different PR Sizes.

4.4.2. Comparisons with the Hybrid-Hash Algorithm for Join-Only Queries

Here I compare the performances of the hybrid-hash join algorithm with the PRS algorithm under the assumption that large main memory buffer is available. It has been shown in [DeWi84] that for wide range of parameter values (e.g., relation sizes, cpu time to swap two tuples, etc.), the hybrid-hash performs better than the sort-merge and GRACE-hash join algorithms. Because of the large main memory buffer, their cost model also considers CPU time. The cost for the hybrid-hash does not consider the initial cost of reading the relations because this cost is same for all the algorithms used for comparison in [DeWi84]. The preprocessing in PRS algorithm may reduce this initial cost of reading the relations. Thus, the cost comparisons also include the initial cost of reading the relations. Appendix D gives the cost expressions of the hybrid-hash and PRS join algorithms.

Figure 4.9 shows the total cost of the algorithms as a function of the join factor. As seen in the figure the PRS is better than the hybrid-hash for only low join factors. However, one can use the hybrid-hash method for the second phase of the PRS algorithm instead of using the sort-merge technique. The cost of the modified PRS (MPRS) as well as the SM algorithm are also shown in Figure 4.9. In this figure I have assumed two relations with 10000 pages each, PRs of size 1000 pages, and memory of size 1200 pages. As shown in the figure, the MPRS algorithm performs better than the hybrid-hash for most join factors. For very high join factors MPRS does not perform as good as the hybrid-hash because the preprocessing in MPRS does not reduce the cost of the second phase.

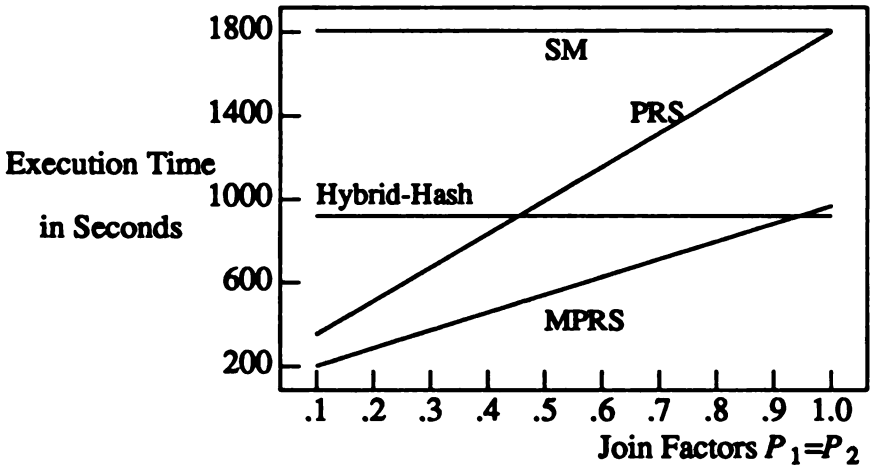


Figure 4.9. Comparisons of Algorithms PRS, MPRS, SM and Hybrid-Hash.

CHAPTER 5

CONCLUSIONS AND FUTURE STUDY

5.1. Conclusions

Until recently, main memory storage was so expensive that only small main memory buffers could be used for query processing. With the availability of cost effective large main memories, procedures are being developed to reduce the I/O cost of processing a query. This work has investigated the performance of relational join operation when a large memory buffer is available. However, I have also considered small buffer size to investigate the performance trade-off between the buffer size and the number of disk I/Os. Here, the theoretical lower bound on the number of disk accesses is achieved. Both join-only queries and queries involving selections, projections and join are considered. The objective of this thesis was to optimize the cost of relational join operation in a paging environment for a given set of constraints and assumptions.

A classification scheme for the existing join algorithms based on the availability and use of indices has been presented. The three classes of the algorithms are, the relation-scan, the index-scan and the hybrid classes. The research showed that the relation-scan class of algorithms perform best when the join and/or restriction factors are high. As mentioned in Chapter 2, most of the existing algorithms fall into the relation-scan class. For the index-scan class of algorithms several graph models are presented in order to show that the optimization problem is NP-hard. Therefore, a heuristic algorithm which has linear time complexity is given. The algorithms of the index-scan class perform best when the join and/or restriction factors are low. For the hybrid class, several algorithms which are based on the preprocessing Partial-Relations are proposed. The research showed that for a wide range of restriction and/or join factors the proposed algorithms perform better than the relation-scan class of algorithms.

In general, the algorithms of the hybrid class perform better than the algorithms of the relation-scan and the index-scan classes for medium range of join and/or restriction factors.

5.2. Future Study

In traditional database applications, queries requiring more than 10 joins are considered improbable [Kris86]. However, in nontraditional database applications, such as *expert database systems*, a need for performing hundreds or more joins is not uncommon [Ullm 85, Zani85]. An expert database system attempts to emulate the reasoning of a human expert in some knowledge domain. It does it by using both the basic facts stored as data in the database and the rules in the knowledge base [Kort86]. Here the extension of this research, (i.e., the use of the graph models as well as the Partial-Relations), in two different contexts namely the m-way join operation and the recursive queries are being discussed.

5.2.1. M-Way Join Operation

In this research the join operation between two relations is being considered. However, to process a query we may be required to join two relations, and then join the resulting relation with another relation. This is called a *3-way* join. Similarly, the m-way join of m relations, $R_1, R_2, \cdots, R_m$ requires the following sequence of $m-1$ join operations:

$$\begin{aligned} B_1 &= R_1 \bowtie R_2 \\ B_2 &= B_1 \bowtie R_3 \\ &\vdots \quad\quad \vdots \\ B_{m-1} &= B_{m-2} \bowtie R_m \end{aligned}$$

where B_i for $i=1,2,...,m-1$, is a temporary relation which holds the result of a join operation between two relations, and B_{m-1} is the output relation. Instead of performing join between relations PRs can be used to perform m-way join. Therefore, algorithms

PRS, PRJ, and MPRS can be extended to support m-way join. For example, the extension of the algorithm PRJ for performing m-way join is as follows:

$$TB_1 = PR(R_1) \bowtie PR(R_2)$$

$$TB_2 = TB_1 \bowtie PR(R_3)$$

$$\vdots \qquad \vdots$$

$$TB_{m-1} = TB_{m-2} \bowtie PR(R_m),$$

where TB_i, for $i=1,2,...,m-1$, is a PR which contains (AA_1,AA_2,*join value*) triplets, and TB_{m-1} contains (AA_1,AA_2,...,AA_m). AA_i is an access attribute value for relation R_i. To join $PR(R_i)$ with TB_{i-2}, the two PRs are first sorted on their join attribute values and then the two sorted relations are merged. Preliminary results showed that as m in m-way join operation increases, so does the improvement of the performance of our algorithms over the sort-merge join algorithm.

One way to optimize the cost of m-way join is to take advantage of the availability of conventional indices on the join attributes. Several sets of pointer pairs are obtained by joining the relevant indices on the joining relations. These pairs are then used to construct a page connectivity graph. Using this connectivity graph, an algorithm for determining optimal number of accesses to all the join participating relations needs to be developed. I am currently investigated this method.

Another issue in performing m-way join is to determine the order in which the relations or indices are being joined. For example, joining R_1 with R_2 first and then R_2 with R_3 may be more costly than joining R_3 with R_2 first and then R_1 with R_2. The problem of determining the optimal sequence in which the relations or indices are being joined is suspected to be NP-hard.

5.2.2. Recursive Queries

Knowledgebase systems must support richer query languages than the traditional database systems [Vald86]. For example a knowledgebase system must support recursion. A simple type of recursion is *transitive closure*. Transitive closures involve a

relation with two or more attributes which have the same domain. When the operation is applied to a relation *R* with two attributes *A* and *B* of the same domain, it would add to *R* all the tuples that can be deduced by transitivity from *R* [Vald86]. Therefore, this operation can be implemented using a series of joins and unions. Consider the following relation (which is taken from an example in [Vald86]):

PART (part-name, category, weight, support-part-name),

where part-name and support-part-name are of the same domain. Category and weight are attributes associated with a part-name. Then one can ask the following query:

What is the total weight of each part in category "c"

The total weight of a part *p* of category *c* is the weight of *p* plus the sum of the weights of all the parts that *p* transitively supports. This query can be implemented by a series of joins and unions on the relation PART.

Valduriez and Boral [Vald86] have discussed and analyzed two algorithms, namely the brute force and the logarithmic algorithms, for performing transitive closure. For example, the brute force algorithm joins relation *R* with itself and stores the result in a temporary relation *T*. It then joins *T* with *R* and store the result in *T*. This process repeats until the result of joining *R* with *T* is empty. At each step the result of the join is also added to the final relation. The performance of the two proposed algorithms has been compared for two different cases. For the first case, the algorithms operate directly on the base relation. In the second case, the algorithms operate on data structure called the join indices † . Their result showed that the use of join indices improve the performance of the brute force and the logarithmic algorithms. These algorithms use hybrid-hash method for performing join and a modification of it is used for performing union operation. For the future study, I would like to investigate the use of Partial-Relations instead of the join indices in performing transitive closure. As the number of joins

† A join index is an abstraction of the join of two relations.

increases one expects to get very small join factors. Therefore, this suggests the use of the graph models for performing transitive closure.

APPENDICES

APPENDIX A

Appendix A

Cost Comparison of Creating an Index Vs. Creating a PR

Here we compare the cost of creating a B-tree index structure with that of creating a PR. The cost of creating a B-tree for a relation with n tuples is as follows:

$$\frac{n}{B}+2\times(\frac{n}{e}\log\frac{n}{e})+\sum_{i=2}^{l}\frac{n}{e^i}$$

where B is the number of tuples of the relation that fit in a block, e is the number of (key,ptr) pair that fit in a block, and $l=\log\frac{n}{e}$ is the depth of the B-tree. The first term in the above expression is the cost of reading the relation, the second term is the cost of sorting(using Z−way merge-sort) the indexed column, and the remaining terms are the cost of writing the internal nodes to the disk. We sort the indexed column first and then create the leaf nodes of the B-tree. Note that while the leaf nodes of the B-tree are created we can also create the internal nodes.

The cost of creating a PR is $\frac{n}{B}+\frac{n}{p}$ where p is the number of PR tuples that fit in a block. Table A.1 shows the number of block accesses for constructing a B-tree versus that of a PR when $B=10$, $Z=10$, $p=20$, and $e=50$. We see from the table, the cost of creating a PR is much less than the cost of creating an index for large n.

Table A.1. Cost Comparisons of Creating a B-tree vs. a PR.

n	PR Cost	B-tree Cost
1000	150	152
10000	1500	1925
100000	15000	23244
1000000	150000	272449

APPENDIX B

Appendix B

Computing the Number of Unique Attribute Values in a PR

Let S be a set which contains all possible strings (i.e., attribute values) of length L generated randomly using an alphabet set of size A. Therefore, the number of possiblevalues in S is $D=A^L$. We assume that the Value-Reducer function selects the first x characters of the attribute values. We can partition S into $G=A^x$ groups with each group containing $K=A^{L-x}$ values with the same first x characters. Let $P_i(T)$ be the probability that there is no value from group i in a sample of size T randomly chosen from S. Therefore, when $D>T+K$

$$P_i(T)=\frac{D-K}{D}\times\frac{D-K-1}{D-1}\times\cdots\times\frac{D-K-T+1}{D-T+1}=\frac{\binom{D-K}{T}}{\binom{D}{T}}$$

otherwise $P_i(T)=0$. Thus, the expected number of unique values in a sample of size T is

$$\sum_{i=1}^{G}1-P_i(T)=G\times\frac{\binom{D-K}{T}}{\binom{D}{T}}.$$

APPENDIX C

Appendix C

I/O Cost of the Sort-Merge Algorithm

Here, we give the cost of the sort-merge join algorithm under the three conditions described in section 4. Under condition A the cost is:

$$\sum_{i=1}^{2} \mu_i + 2J_iN_i/E_i(log_{M-1}(J_iN_i/(2E_i(M-1)))) + 2J_iN_i/E_i$$

where $\mu_i = \min(J_iN_i , N_i/E_i)$.

The cost of the algorithm under condition B is $\sum_{i=1}^{2} \min(N_i/E_i , \tau(J_i,N_i,E_i))$. Finally the cost under condition C is

$$\sum_{i=1}^{2} N_i/E_i + 2J_iN_i/E_i(1+\log_{M-1}(J_iN_i/(2(M-1)E_i))).$$

APPENDIX D

Appendix D

Execution Time of the Hybrid-Hash, SM, PRS and MPRS Algorithms

Here we give the cost of the hybrid-hash, SM, PRS and MPRS join algorithms based on the number of disk I/Os and the cpu usage. The following parameters, which are given in [DeWi84], are used for expressing the cost of the above algorithms:

comp time to compare two key values

hash time to hash a key

move time to move a tuple

swap time to swap two tuples

IO_{SEQ} time for sequential I/O operation

IO_{RAND} time for random I/O operation

F universal "fudge" factor. This factor is used for expressing for example, the extra storage that a relation need to hold its hash table in the buffer.

For the hybrid-hash the cost of joining two relations R and S is $\sum_{i=1}^{8} h_i$, where

h_1= $(|R|+|S|)\times IO_{SEQ}$. The cost of reading the relations.

h_2= $(\lceil |R|/|M| \rceil + \lceil |S|/|M| \rceil)\times hash$. The cost for partitioning the relations.

h_3= $(\lceil |R|/|M| \rceil + \lceil |S|/|M| \rceil)\times(1-q)\times move$. The cost of moving the tuples to the output buffers. $q=\frac{|R_0|}{|R|}$.

h_4= $(|R|+|S|)\times(1-q)\times IO_{RAND}$. The cost of writing from output buffers.

h_5= $(\lceil |R|/|M| \rceil + \lceil |S|/|M| \rceil)\times(1-q)\times hash$. The cost for building hash tables for R and finding where to probe for S.

h_6= $|S|/|M|\times F\times comp$. The cost for moving tuples to hash tables for S.

h_7= $||R||\times move$. The cost for moving tuples to hash table for R.

h_8= $(|R|+|S|)\times(1-q)timeIO_{SEQ}$. The cost for reading the partitions into the memory.

The cost of the SM join algorithm is $\sum_{i=1}^{6} s_i$, where

s_1= $(|R|+|S|)\times IO_{SEQ}$. The cost of accessing the relations.

s_2= $(||R||\log_2 \frac{MR}{F}+||S||\log_2 \frac{MS}{F})\times(comp+swap)$. The cost of inserting tuples into priority queue to form initial runs. MR and MS specify the number of tuples from $R$and S, respectively, that can fit in main memory at once.

s_3= $(|R|+|S|)\times IO_{SEQ}$. The cost of writing the initial runs.

s_4= $(|R|+|S|)\times IO_{RAND}$. The cost of reading initial runs for final merge.

s_5= $(||R||\log_2 \frac{||R||\times F}{MF}+||S||\log_2 \frac{||S||\times F}{MS})\times(comp+swap)$. The cost of inserting tuples into priority queue for final merge.

s_6= $(||R||+||S||)\times comp$. The cost of joining the results of final merge.

For the PRS and MPRS algorithm the cost of the first phase (i.e., the cost of performing semijoin and creating temporary relations) is $\sum_{i=1}^{4} p_i$, where

p_1= $|PR(R)|\times IO_{SEQ}+||PR(R)||\times hash$. The cost of scanning PR(R) and hashing its join attribute values.

p_2= $|PR(S)|\times IO_{SEQ}+||PR(S)||\times(hash+comp)$. The cost of scanning PR(S), hashing its join values and compare those of PR(R).

p_3= $|R|\times IO_{SEQ}+||R||\times(hash+comp)$. The cost of scanning relation R and obtaining the tuples which participate in the join.

p_4= $|S|\times IO_{SEQ}+||S||\times(hash+comp)$. The cost of scanning relation S and obtaining the tuples which participate in the join.

After the first phase, the size of the relations are $|R|\times P_1$ and $|S|\times P_2$. The cost of the second phase for the MPRS algorithm is the same as the hybrid-hash algorithm with the assumption that the size of the relations are reduced. Therefore, the total cost for MPRS algorithm is $\sum_{i=1}^{4} p_i + \sum_{i=2}^{8} h_i$. Note that the term h_1 is not considered in this expression. This is because, while the relations are being reduced the selected tuples are being hashed. For the PRS algorithm the cost of the second phase is the same as that for the SM algorithm. Therefore, the total cost for the PRS algorithm is $\sum_{i=1}^{4} p_i + \sum_{i=2}^{6} s_i$.

For Figure 23, the following parameter values are assumed:

$comp$=3μs $hash$=9μs $move$=20μs $swap$=60μs IO_{SEQ}=10ms IO_{RAND}=25ms

F=1.2 $|S|=|R|$=10,000 $//S//=//R//$=400,000 M=1200

LIST OF REFERENCES

LIST OF REFERENCES

[Amma85] A. Ammann, M. Hanrahan and R. Krishnamurthy, "Design of a Memory Resident DBMS," *Proc. IEEE COMPCON*, San Francisco, February 1985.

[Babb79] E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," *ACM TODS*, Vol. 4, No. 1, March 1979.

[Burk76] W. A. Burkhard, "Hashing and Trie Algorithms for Partial Match Retrieval," *ACM TODS*, Vol. 1, No. 2, June 1976.

[Baye72] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta Informatica*, Vol. 13, No. 3, 1972.

[Bitt83] D. Bitton, H. Boral, D. DeWitt and K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," *ACM TODS*, Vol, 8, No. 3, September 1983.

[Blas76] M.W. Blasgen and K.P. Eswaran, "On the Evaluation of Queries in a Relational Data Base System," *Technical Report RJ 1745 (#25553), IBM Research Laboratory*, San Jose, California, April 8, 1976.

[Blas77] M.W. Blasgen and K.P. Eswaran, "Storage and Access in Relational Databases," *IBM System Journal*, Vol. 16, No. 4, 1977.

[Brat84] K. Bratbergsengen, "Hashing Methods And Relational Algebra Operations," *Proc. of 10th Int. Conf. on VLDB*, Singapore, August 1984.

[Codd70] E.F. Codd, "A Relational Data Model for Large Shared Data Banks," *Comm. ACM*, Vol. 13, No. 6, June 1970.

[Come78] D. Comer, "The Difficulty of Optimal Index Selection," *ACM TODS*, Vol. 3, No. 4, December 1978.

[Come79] D. Comer, "The Ubiquitous B-tree," *ACM Computing Surveys*, Vol. 11, No. 2, 1979.

[Date75] C.J. Date, *An Introduction to Database Systems*, Addison Wesly Publishing Co., Reading, Massachusetts, 1975.

[DeWi79] D.J. DeWitt, "DIRECT-A Multiprocessor Organization for Supporting Relational Database Management Systems," *IEEE Trans. on Computers*, Vol. C28, No. 6, June 1979.

[DeWi84] D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proc. ACM SIGMOD Conf.*, June 1984.

[Elha84] K. Elhardt and R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM TODS*, Vol. 9, No. 4, December 1984.

[Fagi79] R. Fagin, N. Pipenger, and H.R. Strong, "Extendible Hashing-A Fast Access Method for Dynamic Files", *ACM TODS*, Vol. 4, No. 3, September 1979.

[Foto88] F. Fotouhi and S. Pramanik, "Optimizing the Cost of Relational Queries Using Partial-Relation Schemes," to appear in *Information Systems*, Vol. 13, No. 1, 1988.

[Gard81] G. Gardarin, "An Introduction to SABRE: A Multimicroprocessor Database Machine," *Proc. of 6th Workshop on Computer Architecture for Non-numeric Processing*, Hyeres, France, June 1981.

[Gare79] M. Garey and D. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.

[Good80] J.R. Goodman, "An Investigation of Multiprocessor Structures and Algorithms for Database Management," *Ph.D. Dissertation*, University of California, Berkeley, December 1980.

[Gotl75] L.R. Gotlieb, "Computing Joins Of Relations," *Proc. ACM SIGMOD Conf.* , May 1975.

[Hsia83] D.K. Hasiao, Advanced Database Machines, *Prentice Hall*, Englewood Cliffs, New Jersey, 1983.

[Kits83] M. Kitsuregawa, H. Tanaka, and T. Motooka, "Application of Hash to Database Machine and its Architecture," *New Generation Computing*, Vol. 1, No. 1, 1983.

[Kits84] M. Kitsuregawa, H. Tanaka, and T. Motooka, "Architecture and Performance of Relational Database Machine," *Proc. of Int. Conf. on Parallel Processing*, 1984.

[Knot75] G.D. Knott, "Hashing Functions," *The Computer Journal*, Vol. 18, No. 3, 1975.

[Knut73] D.E. Knuth, *The Art of Computer Programming, Vol.3 (Searching and Sorting)*, Addison Wesley Publishing Company, Reading, MA, 1973

[Kort86] H.F. Korth and A. Silberschatz, *DATABASE SYSTEM CONCEPTS*, McGraw-Hill, Inc., New York, NY, 1986.

[Kris86] R. Krishnamurthy, H. Boral and C. Zaniolo, "Optimization of Nonrecursive Queries," *Proc. of the 12th Int. Conf. on VLDB*, Kyoto, August 1986.

[Lars78] P. Larson, "Dynamic Hashing," *BIT*, Vol. 18, 1978.

[Lars85] P. Larson, "Linear Hashing without Overflow Handling by Linear Probing," *ACM TODS*, Vol. 10, No. 1, March 1985.

[Lehm86a] T. Lehman and M. Carey, "Query Processing in Main Memory Database Management Systems," *Proc. ACM SIGMOD Conf.*, May 1986.

[Lehm86b] T. Lehman and M. Carey, "A Study of Index Structure for Main Memory Database Management Systems," *Proc. of 12th Int. Conf. on VLDB*, Kyotto, August 1986.

[Lela85] M. Leland and W. Roome, "The Silicon Database Machine," *Proc. 4th Int. Workshop on Database Machines*, Grand Bahama Island, March 1985.

[Litw76] W. Litwin, "Virtual Hashing: A Dynamically Changing Hashing," *Proc. of 4th Int. Conf. on VLDB*, 1976.

[Litw80] W. Litwin, "Linear Hashing: A New Method for File and Table Addressing," *Proc. of 6th Int. Conf. on VLDB*, 1980.

[Lome83] D.B. Lomet, "Bounded Index Exponential Hashing," *ACM TODS*, Vol. 18, No. 1, March 1983.

[Lum70] V.Y. Lum, "Multi-attribute Retrieval with Combined Indexes," *Communication of ACM*, Vol. 13 No. 11, November 1970.

[Lum71] V.Y. Lum, P.S.T. Yuen, and M. Dodd, "Key-to-Address Transformation Techniques, A Fundamental Performance Study on Large Existing Formatted Files," *Comm. of ACM*, Vol. 14, No. 4, April 1971.

[Lum73] V.Y. Lum, "General Performance Analysis of Key-to-Address Transformation Method using an Abstract File Concept," *Comm. of ACM*, Vol. 16, No. 10, October 1973.

[Merr81] T. Merrett, Y. Kambayashi, and H. Yasuura, "Scheduling of Page-Fetches in Join Operations," *Proc. of 7th Int. Conf. on VLDB*, Cannes, France, 1981.

[Miss82] M. Missikoff, "A Domain Based Internal Schema for Relational Database Machines," *Proc. ACM SIGMOD Conf.*, June 1982.

[Moto83] T. Moto-Oka and K. Fuchi, "The Architectures in the Fifth Generation Computers," *Proc. of IFIP83 World Congress*, 1983.

[Pram85a] S. Pramanik and F. Fotouhi, "An Index Database Machine - An Efficient M-Way Join Processor," *Proc. Eighteenth Annual Hawaii International Conf. on System Sciences*, Hawaii, January 1985.

[Pram85b] S. Pramanik and D. Ittner, "Use of Graph-Theoretic Models for Optimal Relational Database Accesses to Perform Join," *ACM TODS*, Vol. 10, No.

1, March 1985.

[Pram86a] S. Pramanik and F. Fotouhi, "Join and Semi-Join Algorithms Based on Partial- Relation Schemes," *Proc. 6th Advanced Database Symposium*, Kyotto, August 1986.

[Pram86b] S. Pramanik and F. Fotouhi, "Index Database Machine," *The Computer Journal*, Vol. 29, No. 5, October 1986.

[Pram86c] S. Pramanik, "Performance Analysis of a Database Filter Search Hardware," *IEEE Trans. on Computers*, Vol. C35, No. 12, December 1986.

[Rich87] J.R. Richardson, H. Lu, and K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms," *Proc. of ACM SIGMOD*, May 1987.

[Sacc86a] G.M. Sacco, "Fragmentation: A Technique for Efficient Query Processing," *ACM TODS*, Vol. 11, No. 2, June 1986.

[Sacc86b] G.M. Sacco and M. Schkolnick, "Buffer Management in Relational database Systems," *ACM TODS*, Vol. 11, No. 4, December 1986.

[Sack83] R. Sacks-Davis, and K. Ramamohanarao, "A Two Level Superimposed Coding Scheme for Partial Match Retrieval," *Information Systems*, Vol. 8, No. 4, 1983.

[Schk75] M. Schkolnick, "Secondary Index Optimization," *ACM SIGMOD*, 1975.

[Schu79] S.A. Schuster, H.B. Nguygen, D.A. Ozkarahan, and K.C. Smith, "RAP.2: An Associative Processor for Databases and Its Applications," *IEEE Trans. on Computers*, Vol. C28, No. 6, June 1979.

[Seve74] D.G. Severance, "Identifier Search Mechanisms: A Survey and Generalized Method," *Computing Surveys*, Vol. 6, No. 3, 1974.

[Shap86] L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM TODS*, Vol. 11, No. 3, September 1986.

[Shib84] S. Shibayama, "A Relational Database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor," *New Generation Computing*, Vol. 2, No. 2, 1984.

[Shul84] R.K. Shultz and R.J. Zingg, "Response Time Analysis of Multiprocessor Computers for Database Support," *ACM TODS*, Vol. 9, No. 1, March 1984.

[Smit75] J. Smith and P. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," *Comm. ACM*, October 1975.

[Su79] S.Y.W. Su, L.H. Nguyen, A. Emmam, and L.Lipovski, "The Architectural Features and Implementation Techniques of Multicell CASSM," *IEEE Trans. on Computers*, Vol. C28, No. 6, June 1979.

[Tana79] K. Tanaka, C. Viet, Y. Kambayashi, and S. Yajima, "A File Organization Suitable for Relational Database Operations," *Lecture Notes in Computer Science 75*, Springer-Verlag, Berlin Heidelberg, 1979.

[Tera83] Teradata Corporation, DBC/1012 *Database Computer Concepts and Facilities*, Inglewood, CA, April 1983.

[Vald84] P. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *ACM TODS*, Vol. 9, No. 1, March 1984.

[Vald85] P. Valduriez, "Join Indices," *MCC Technical Report*, No. DB-052-8, Austin, Texas, 1985.

[Vald86] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices," *Proc. of 1st Int. Conf. on Expert Database Systems*, Charleston, April 1986.

[Ullm82] J.D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockvill, Maryland, 1982.

[Ullm85] J.D. Ullman, "Implementation of logical query languages for databases," *ACM TODS*, Vol. 10, No. 3, September 1985.

[Zani85] C. Zaniolo, "The Representation and Deductive Retrieval of Complex Objects," *Proc. of 11th Int. Conf. on VLDB*, 1985.