DISK I/O PERFORMANCE OF LINEAR RECURSIVE QUERY PROCESSING

by

Simon Hon Ming Mok

B.A., University of Winnipeg, 1984

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

July 1987

ISBN 0-315-42603-9

APPROVAL

Name: Simon Hon Ming Mok

Degree: Master of Science

Title of thesis: Disk I/O Performance of Linear Recursive Query Processing

Examining Committee:

Chairman: Dr. B. Bhattacharya

Dr. W. S. Luk
Senior Supervisor

Dr. R. F. Hadley
Committee Member

Dr. W. Yu
External Examiner

Date Approved: July 21, 1987

ABSTRACT

This thesis presents an empirical simulation study of the *disk I/O* performance of three *Linear Recursive Query Processing* algorithms using the *compilation approach* in *Deductive Database* systems.

The primary objective of this research is to analyse the I/O behaviour of database query processing with a new cost metric, the number of disk retrievals (disk I/O), in contrast to the conventional cost metric, the number of page accesses (page I/O). Disk I/O is a more suitable cost metric because it takes into account technological advances in hardware and software. Linear recursive query processing is chosen as a basis for this research because it requires iterative execution of sequences of relational database operations and repetitive accessing of large amounts of data. The secondary objective of this research is to examine the efficiency of three well-known recursive query processing algorithms at the disk I/O level.

The research results demonstrate that it is important to include disk I/Os as an additional parameter in the cost formula of a query processing strategy when a large buffer space is available. Also, a methodology is suggested to estimate the Maximum Buffer Requirement (MBR) of a query processing algorithm.

The research results also show that in order to minimize disk I/Os, special attention must be paid to reference locality. Specifically, an algorithm can achieve good disk I/O performance if the processing of one large relation is completed before the processing of another large relation begins. The simple method developed is also shown to be useful for inspecting the reference locality of the algorithms.

Finally, it is shown that it is not only important to choose the right algorithm for processing recursive queries, it is equally important to choose the right strategies for implementing low-level relational database operations.

DEDICATION

To my late father.

ACKNOWLEDGMENTS

This work could not have been completed without the support of my senior supervisor Dr. W. S. Luk. I am grateful for his motivation and the provision of equipment for this research. I would also thank Dr. R. F. Hadley for his valuable constructive criticism of this work.

This research is partially supported by the SFU Open Graduate Scholarship and the Challenge 1987 Funding.

My sincere thanks to the support from the fellow graduate students in Radandt Hall. Especially, Ms. Sharon Hamilton and Rev. Robert Neville for their long hours devoted to editing this thesis.

Finally, my deepest thanks and appreciation to my wife, Marilyn, for her constant support and understanding throughout this entire effort.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

This thesis presents an empirical simulation study of the *disk I/O* performance of three *Linear Recursive Query Processing* algorithms using the *compilation approach* in *Deductive Database* systems. The primary objective of this research is to analyse the I/O behaviour of database query processing with a new cost metric, the number of disk retrievals (disk I/O), in contrast to the conventional cost metric, the number of page accesses (page I/O). Disk I/O is a more suitable cost metric because it takes into account technological advances in hardware and software. Linear recursive query processing is chosen as a basis for this research because it requires iterative execution of sequences of relational database operations and repetitive accessing of large amounts of data. The secondary objective of this research is to examine the efficiency of three well-known recursive query processing algorithms at the disk I/O level.

This chapter discusses the importance of using disk I/O as the cost metric and introduces the general concepts of Deductive Database systems and recursive query processing. Chapter 2 presents the five algorithms considered in this study, along with related work on the performance of these algorithms. The model in which the I/O behaviour is studied, together with its implementation, is presented in Chapter 3. Chapter 4 contains the analysis and interpretation of the simulation results. Several implementation issues which may affect the simulation results are discussed in Chapter 5. Finally, Chapter 6 contains a summary of this research and suggestions for further research.

1.1 Performance Metrics

Along with the theoretical aspects of relational database systems, it is necessary to consider performance issues. In particular, the performance of database query processing strategies must be studied. Many such studies have been done over the years to identify the factors which affect the efficiency of the different methods of evaluating queries. The results obtained from these studies are used to fine tune database systems to improve their overall efficiency. The metrics used to measure the processing efficiency are the CPU costs and the I/O costs. The CPU costs are generally evaluated in terms of the number of tuples being processed while the I/O costs are measured by the number of page accesses needed to answer a query (*page I/Os*). This thesis will focus on the I/O costs of relational database query processing.

The I/O costs of relational database query processing have been studied by many researchers in various performance studies (e.g. [34]). However, most of the studies either ignore the cost of disk accesses or equate the number of page accesses to the number of disk accesses. The latter assumption has until now been accepted as a close approximation to reality. Recently, however, some researchers have challenged this assumption because it does not take into account three major developments: the improvement in system software and hardware performance, the proven inefficiency of conventional buffering schemes in a database environment, and the decrease in costs of hardware such as main memory chips. We shall discuss each of these developments in detail.

It is claimed in [12] that the number of page I/Os is not a suitable metric for evaluating I/O costs. Instead, the number of disk I/O operations (*disk I/Os*) is proposed to take into account developments in system software and hardware. In high performance file systems such as the 4.2 BSD [21], for example, adjacent file pages are often clustered together on the disk. This is in contrast with primitive file systems, which allocate adjacent pages to widely separated disk addresses. This change, together with the increase in the transfer rate between the disk and the processor, means that adjacent file pages can now be accessed in a single disk I/O operation. Thus, the number of disk I/O operations is a more suitable metric than the number of page I/O operations for evaluating the I/O costs of relational query processing.

Another drawback of using page I/Os to evaluate I/O costs is that cost formulas based on the conventional model do not take into account the effects of system factors such as the available buffer size and the page replacement policy in use. For example, the cost formula for a *nested-loop indexed join* on two relations is conventionally formulated as follows [34]:

$$I/O\ cost = NumOfPage(outer) + NumOfTuple(outer) * (w_1 * Depth(Index) + w_2 * j * NumOfTuple(inner))) * Page\text{-}I/O\text{-}cost$$

where *outer* and *inner* denote the two relations, j is the *join selectivity*, w_1 is the probability that the index page is not in the buffer and w_2 is the probability of having to fetch a data page when matching tuples are found. In this cost formula, the size of the buffer and the buffer management scheme are determined by the two probabilities w_1 and w_2. In most performance studies, w_1 and w_2 either are not specified or are arbitrarily chosen. In other words, the size of the buffer and the buffer management scheme are usually assumed to be those of a typical operating system. It is argued in [36], however, that relational database accesses can be classified into a set of reference patterns which may not be well served by conventional buffer management schemes. A number of other research papers [9,19,27,28,32] have shown that the pattern of page references exhibited by relational database accesses is very regular and predictable. The conclusion drawn from these studies is that for an operating system to provide good buffer management, it must be able to accept advice from the Database Management System (DBMS) concerning the page replacement strategy. The DBMS can also use specialized buffer management schemes such as the *Hot set* algorithm [32] and the *DBMIN* algorithm [9] in order to improve the throughput on disk accessing.

Finally, as semiconductor memory becomes cheaper, it is no longer uncommon to find large memory buffers being made available to database systems. In fact, Main Memory Database systems are currently an active research area in the database community (e.g. [6]). While there is some doubt that main memory will ever be cheap enough to make disk storage redundant, it is clear that, in the near future, there will be plenty of buffer space available to boost the performance of complex database processing. As more and more memory becomes available, the difference between the disk I/Os and the page I/Os will be more significant, because a request for an index or data page of a relation may not actually require retrieval of the requested page from the disk if the page happens to be found in the buffer at that moment. Therefore, the size of the buffer space is a very important factor in determining the disk I/Os, and this is why it is primary focus of this research project.

The primary objective of this thesis is to study the I/O behaviour of relational database query processing with special emphasis on the number of disk I/O operations, and in particular, the effect of the buffer size on performance. We will also study the relationship between the old metric, the number of page I/Os, and the new metric, the number of disk I/Os. A particular type of query processing, *Recursive Query Processing*, is chosen to be a case study in this research because recursive query processing requires sequences of relational database operations to be executed, and large amounts of data to be transferred between the secondary storage, the disk, and the processor. Recursive query processing is an advanced query facility used in *Deductive*

Database (DDB) systems which apply *logic* to conventional relational databases. A general description of the general area of logic and database research and recursive query processing in DDB systems is presented in the next section.

1.2 Logic and Database

In recent years, there has been growing research interest in the connection between logic and databases; its primary goal being the use of logic as an inference system and a representation language in databases. It has been shown by numerous researchers and summarized in [11] that mathematical logic, primarily first-order logic, provides a precise framework under which many classical database problems can be formalized and studied in detail. Using logic as a formal theoretical basis, many researchers have studied database topics such as representing and extending existing query languages, modelling and maintaining integrity constraints, optimizing query processing, and representing various kinds of data dependencies.

A conventional database, as first characterized by Nicolas and Galliair [10], can be considered from the viewpoint of logic to be an *interpretation* of a *first-order theory* (also known as the *model theoretic view* in [31]). Under such a viewpoint, queries and integrity constraints are treated as formulas whose truth values are to be determined in the interpretation. Other researchers have worked on extending the representation and manipulation capabilities of databases by considering a database to be a *first-order theory* itself (also known as the *proof-theoretic view* in [31]). From such a viewpoint, queries and integrity constraints are seen as *theorems* to be proved; this is precisely the concept behind *Deductive Database* (DDB) systems. For a detailed and formal description of DDB systems, the reader is referred to [11].

A DDB is a database in which new facts can be derived from facts that are explicitly stored in the database. The primary goals of research into DDB systems are 1) to incorporate the functionalities of logic, such as deduction, into the relational database model and 2) to increase the expressive power of database systems to handle queries more sophisticated than those that require only simple data retrieval. From an operational point of view, as defined in [11], a DDB consists of the following parts:

1. a set of axioms which are elementary facts,
2. a set of axioms which are deductive rules, and

3. a set of integrity constraints.

The elementary facts consist of the finite set of data which is stored in the database, and are collectively called the *Extensional Database* (EDB). The *Intensional Database* (IDB) consists of the set of deductive rules and the integrity constraints, and corresponds to general knowledge of the world modelled by the DDB. There is no definite dividing line, however, between what might be considered a deductive rule or an integrity constraint. In current research, deductive rules are limited to *definite* clauses without functions. Functions are not considered so that finite and explicit answers can be found to queries. Deductive rules are limited to definite clauses so that DDB systems will be consistent under the *Closed World Assumption* of [30].

Because of the limitations on queries in DDB systems, their inferencing capabilities are quite limited. In fact, the major objection to DDB systems raised by many researchers, primarily from the AI community [5], is that they provide only a very restricted version of inferencing which excludes a lot of complex general knowledge of the world. Further, they point out that the logic programming language PROLOG is as good as, if not better than, DDB systems in terms of resolving logic queries. The most significant response to this criticism from researchers following the database school of thought [5] is that although the PROLOG language may be very powerful and ~~expressive, it does~~ not provide the necessary features of a general database management system, which include efficient query facilities, functions for integrity constraints, and maintenance of deduced facts. A major advantage of DDB systems, despite their current logic limitations, is that they incorporate existing database techniques for handling large amounts of data efficiently.

The description of DDB systems presented here is based on the perspective of the database school of thought. The general direction of research in this perspective is to introduce gradually more and more deductive capabilities into database systems as efficiency permits. For an interesting list of open questions and contemporary views on the general area of logic and databases, the reader is referred to [11] and [5].

To further the aims of DDB systems, a great deal of research effort has been directed towards the goal of resolving queries using not only the explicit facts stored in the database, but also implicit information that can be derived from the explicit facts. This is known as ***recursive query processing*** because unlike the execution of simple queries which require access only to explicit facts, the execution of recursive queries often requires iterative processing of a sequence of operations such as deductive rule resolution and database accesses.

There are two general approaches to resolving recursive queries: the ***interpretive approach*** and the ***compilation approach.*** The interpretive approach [24] works with a problem solver in which the resolution of deductive rules and the accessing of facts in the extensional database are interleaved during the process of resolving the query. In the compilation approach [29], the problem solver uses only the deductive rules until a point is reached at which either the query is solved or all that remains is to search for facts in the Extensional database.

Both approaches have advantages and disadvantages. Algorithms using the interpretive approach are smarter in the sense that decisions about the relevance of facts towards the answer of the query can be made dynamically at run time. However, it is expensive to access a large database iteratively in a tuple–at–a–time fashion, as currently required by the interpretive approach. Some techniques which explore the idea of obtaining a set of tuples at each database access, rather than a single tuple at a time, are currently being developed [7,22,23,25].

The major advantage of the compilation approach is that access to the database is delayed until the end of the deductive process, allowing global optimization of database accesses by incorporating already–known techniques from conventional database theory. The major drawback of this approach is that the classes of rules that can be compiled into simple formulas is very limited. In fact, the study presented in [3] indicates that the largest class of rules that can be processed by currently known compilation–based algorithms is the class of *Bottom–up evaluable* rules. Any computation using only a set of bottom–up evaluable rules can be carried out without materializing infinite intermediate results. The bottom–up evaluability criterion ensures that the set of values for body variables is finite at each step. However, there may be an infinite number of steps. Expanding the class of compilable rules is another active research area in DDB systems. For formal descriptions of various known compilation techniques, the reader is referred to [8,18,29,37].

Two additional issues which are beginning to draw the attention of many researchers are the performance and ease of implementation of various recursive query processing strategies. In this research we have chosen to study these two issues of recursive query processing from the database perspective ~~and~~ using disk I/O operations as the performance metric. Also, we focus on recursive query processing strategies which use the compilation approach, because the clear separation of database accesses from deductive rule resolution makes the approach amenable to implementation and analysis with a conventional relational database model. As mentioned earlier, a DDB consists of an IDB and an EDB. The IDB contains a search routine and an inference engine which only operate on the deductive rules, while the EDB is a conventional relational database. Our research is focused on the EDB.

A recent paper by Han [15] identified a set of five different types of recursive query rules, which can be summarized by the following three canonical forms of recursive rule clusters:

1. The *canonical transitive closure* cluster,
2. The *canonical linear recursive* cluster, and
3. The *canonical non-linear recursive* cluster.

Of these three canonical forms of recursive clusters, the first two can be compiled into simple formulas using a simple step-wise method described in Section 2.2. The last form can not be compiled into simple formulas, and other techniques such as the *stack-directed query compilation algorithm* [14] have been developed for processing queries of this form.

This research uses the linear recursive rule cluster for the study of disk I/O operations for two reasons. First, the linear recursive rules require a more complex sequence of relational operations, such as Join, Select, Project, and Union, than the transitive closure cluster, which has also been studied extensively in current research. This will provide an appropriate basis for evaluating the disk I/O performance metric. Second, numerous papers [2,3,18,16,17,39] have addressed performance issues related to this particular set of recursive rules. Although these studies have either ignored the cost of disk accesses or used the page I/Os as the metric for measuring the I/O cost, their results provide us with some fundamental qualitative insights into the design of our experiments and the interpretations of our results. Moreover, this study places a different perspective on the current research into recursive query processing.

Before we leave this discussion, we should caution the reader that what we have described here is a very superficial view of the general area of logic and databases. Much of the discussion was based on the database school of thought. As well, since our research only focuses on the very simple class of linear recursive rules and queries, the descriptions of the various algorithms are all tailored to this class. This section has highlighted some of the interesting questions and perspectives of the general area of logic and database research. References are provided to guide the interested readers to more detailed and formal discussions of these issues. To close off this chapter, we recall Ullman's reminder in [38] that one has to start with the simplest issues in order to pursue an ultimate ideal in the future.

CHAPTER 2

A RECURSIVE QUERY AND ITS PROCESSING STRATEGIES

This chapter introduces the recursive rules, the query and the processing strategies that are studied in this thesis. Related work on evaluating the performance of recursive query processing strategies is also discussed.

2.1 A Recursive Query

This thesis studies a set of linear recursive rules written in function-free Horn Clause form. The set contains two Horn clauses, (1) and (2) below, which are generalized forms of the *same generation* example shown in [3]. The terms *up* and *down* are two base predicates that represent the *ancestor* and *descendant* relationships of two individuals, while the *flat* predicate contains tuples of individuals who are cousins of each other (i.e. they are of the same generation). Predicate *r* therefore represents the two ways that two individuals in the database can be cousins (i.e. they are of the same generation). In the rest of this thesis, the term *relation* will be used to refer to a *predicate*. The two rules are shown below:

r(X,Z) :- flat(X,Z). *(1)*
r(X,Z) :- up(X,Y), r(Y,W), down(W,Z). *(2)*

where *up*, *flat*, and *down* are base relations existing in the relational database, *r* is a virtual relation which involves recursion, and *X*, *Y*, *W*, and *Z* are vectors of variables.
The following rule defines the recursive query,
? :- r(a, X). *(3)*
where *a* is a constant vector and *X* is the vector variable to be retrieved from the virtual relation *r*. The answer to the query is an array of integers which is derived either directly or indirectly from rules (1) and (2).

2.2 Compilation of the Recursive Query

As described briefly in Section 1.2, there are two stages to resolving a recursive query using the compilation approach [29]. In the first stage, the set of relevant rules is processed to obtain one or more compiled formulas of the recursive query. These compiled formulas are sequences of relational operations involving data relations which will be generated if they are not already explicitly stored in the database. During the second stage, an iterative program is generated based on the compiled formulas, and the answer to the query is obtained by retrieving facts (data) from the database.

Three different strategies were implemented in this project. These algorithms are: Henschen–Naqvi [18], Magic Set [2] and Counting [2].[1] This study focusses primarily on the second stage of resolving the query, so only a general description of recursive rule compilation is presented in this section. The input to the compilation process is the original set of rules and the query in the case of the Henschen–Naqvi algorithm whereas the input is a transformed set of rules and the query in the cases of the Magic Set and Counting algorithms. The details on how the rules are transformed will be described when each of the two strategies are discussed in the following sections of this chapter.

The following discussion is based on the compilation method that is described in [17]. Linear recursive rules (1), (2) and the query from Section 2.1 are the input to the compilation. The general idea is to expand rule (2) recursively. The following sequence of expansions is obtained by using step–wise recursive calls to rule (2).

$r(X,Z)$:- $up(X,Y_1)$, $up(Y_1,Y)$, $r(Y,W)$, $down(W, W_1)$, $down(W_1, Z)$.

......

$r(X,Z)$:- $up(X,Y_k)$, $up(Y_k,Y_{k-1})$, ..., $up(Y_1,Y)$, $r(Y,W)$, $down(W, W_1)$, ..., $down(W_{k-1},W_k)$, $down(W_k,Z)$.

[1] In fact, two other algorithms, Semi–Naive [1] and Double Wavefront [17], were also implemented. The disk I/O performance of these two algorithms is significantly worse than those of the other three algorithm, thus they are not included in this thesis.

where $Y_1, Y_2, \ldots, Y_k, W_1, W_2, \ldots, W_k$ and Z are vectors of variables.

By calling rule (1) in the above sequence, the solution to the query is obtained by processing the following sequence of non-recursive query sets and taking the union of the results of each step.

flat(a,Z).

up(a,Y), flat(Y,W), down(W,Z).

up(a,Y_1), up(Y_1,Y), flat(Y,W), down(W,W_1), down(W_1,Z).

...

up(a,Y_k), up(Y_k,Y_{k-1}), ...up(Y_1,Y), flat(Y,W), down(W,W_1), ..., down(W_{k-1},W_k), down(W_k,Z).

The expansion of the above sequence terminates when no new solutions are found in the database. [2] The above sequence of query sets can be represented simply as the following query series:

$\sigma_a up^k \bullet flat \bullet down^k$

where $\sigma_a up$ means a *Selection* on the corresponding attributes of relation *up* according to the constant vector *a*, $\bullet$ denotes the *Join* operation on the corresponding join attributes of two relations, and up^k represents performing the Join operation on corresponding join attributes of relation *up* $k-1$ times. The range of the index k is from 0 to n where n is the number of iterations up to the termination point. In our experiments, because randomly generated data relations are used, some databases generate empty relations, and some generate cyclic results for some large number of iterations. To avoid the problems of determining the termination points in this study, the query is executed for a specific number of iterations. The format and notation used to describe the algorithms in the following sections is adapted from [17]. In the following discussion, the concatenation of relation names denotes the result of joining the individual relations. For example, *upflatdown* is the result of joining the *up* relation with the *flat* relation, followed by a join of the intermediate result with the *down* relation.

[2] We have not considered the termination problem in this study; for a detailed treatment of it the reader is referred to [8,13,18,26,35].

2.3 The Algorithms

Before we discuss the three algorithms, a straightforward way to process the query series $\sigma_a up^{k} \bullet flat \bullet down^{k}$ is presented to serve as a basis for understanding the other algorithms.

The simple algorithm, referred to as the Naive algorithm in [35], executes the series in a bottom-up fashion using an iterative loop. The bottom-up approach starts from the base relations and keeps assembling them to produce virtual relations until they generate the answer to the query. The algorithm starts by selecting the first set of results from the relation *flat*. At the first iteration, the three data relations *up*, *flat*, and *down* are *joined* on their corresponding attributes to produce the intermediate relation *upflatdown*. This intermediate relation, defined as the ***wavefront relation*** or the ***frontier relation*** in [17], serves as the starting point for the other iterations. In general, at the i^{th} iteration, the execution proceeds as follows:

1. The relation *up* is joined with the intermediate relation $up^{i-1}flatdown^{i-1}$, which is saved from the $i\text{-}1^{th}$ iteration, to yield the relation $up^{i}flatdown^{i-1}$.
2. The relation $up^{i}flatdown^{i-1}$ is joined with the relation *down* once to produce another intermediate relation $up^{i}flatdown^{i}$.
3. A selection is then performed on the relation $up^{i}flatdown^{i}$ to obtain the set of answers to the query at this iteration. The intermediate relation $up^{i}flatdown^{i}$ is then saved as the wavefront relation for the $i\text{+}1^{th}$ iteration.

The execution plan at the i^{th} iteration can be simply represented as the following sequence:

$$\sigma_a(up \bullet (up^{i-1}flatdown^{i-1}) \bullet down)$$

Since this algorithm proceeds from the center of the query series at each iteration, it is also known as the *Central Wavefront* algorithm in [17]. It terminates when no new solutions are found in the database. The answer to the original query is obtained by taking the union of the entire set of results obtained at each iteration. The general process flow of this algorithm is as follows:

$$\sigma_a flat$$
$$\sigma_a(up \bullet flat \bullet down)$$
$$\sigma_a(up \bullet (upflatdown) \bullet down)$$
....

....
$$\sigma_a(up \bullet (up^{k-1}flatdown^{k-1}) \bullet down)$$

The total number of *join* operations is *2k* in this algorithm. In spite of the small number of join operations, there are two major weaknesses. First, there are two joins on three entirely unrestricted relations at each iteration. This could generate very large intermediate relations and easily cause a combinatorial explosion. It could also generate a large number of tuples which are irrelevant to the query. These problems occur with the bottom–up approach because the algorithm has no knowledge of what query it is trying to solve. Secondly, the results produced at the *i–1*th iteration are completely reproduced at the *i*th iteration, leading to a large number of duplicate results. In the remainder of this section each of the three algorithms – Henschen–Naqvi, Magic Set and Counting is described. They will be compared qualitatively wherever it is appropriate.

2.3.1 The Henschen–Naqvi Algorithm

The Henschen–Naqvi (HN) algorithm was originally presented in [18]. In this study, a simplified version which can be directly applied to the linear recursive query is examined. The simplified method is essentially the *Single Wavefront* algorithm described in [17].

Unlike the Naive method, the HN algorithm employs a top–down approach. The top–down approach is basically the same as the *performing selection first* approach described in [17]. In this approach, the algorithm starts from the query and keeps expanding it by applying the rules to the derived relations. Algorithms using the top–down approach are, in general, more efficient because they "know" which query is being solved, but are often more complex. The major advantage of this approach is that the size of the relations that are joined during the iterations can be significantly reduced. This approach has been shown, both analytically and experimentally, to be very efficient in certain situations compared to the bottom–up approach. Similar conclusions can also be drawn from the simulation results described in chapter 4 of this study.

The algorithm starts by selecting the first set of results from the relation *flat*. The intermediate wavefront relation that is passed on from iteration to iteration, however, is a restricted partial transitive closure of the relation *up*. In general, the execution at the i^{th} iteration goes as follows:

1. The relation *up* is joined with the intermediate relation $\sigma_a up^{i-1}$ to derive the relation $\sigma_a up^i$. This relation will be saved for the $i+1^{th}$ iteration.
2. The relation $\sigma_a up^i$ is then joined with the relation *flat* to produce the relation $\sigma_a up^i flat$.
3. The answer to the query at this iteration is obtained by joining the relation *down* to the relation $\sigma_a up^i flat$ i times.

The execution plan at the i^{th} iteration is represented as follows:

$$(\sigma_a up^{i-1} \bullet up) \bullet flat \bullet down \bullet \ldots \bullet down$$

The processing flow of the HN algorithm is depicted as below.

$$\sigma_a flat$$
$$(\sigma_a up) \bullet flat \bullet down$$
$$(\sigma_a up \bullet up) \bullet flat \bullet down \bullet down$$
$$(\sigma_a up^2 \bullet up) \bullet flat \bullet down \bullet down \bullet down$$
$$\ldots$$
$$\ldots$$
$$(\sigma_a up^{k-1} \bullet up) \bullet flat \bullet down \bullet \ldots \bullet down$$

Although this algorithm is able to reduce considerably the sizes of the relations that are joined at each iteration, it introduces the problem of a large number of repetitive joins on the relation *down*. This can be seen at the i^{th} iteration where the relation *down* is joined with itself i times. Consequently, the number of join operations, which is $(k^2+5k)/2 - 1$, is much higher than for the Naive and Semi-Naive algorithm. This particular aspect of the algorithm was examined in this study; the results of our observations are discussed in chapter 4.

2.3.2 The Magic Set Algorithm

The Magic Set (MS) algorithm uses both the bottom-up and top-down approaches to answer the query. This technique of combining the two approaches is developed from the *sideways information passing* strategy in [33]. It uses a hypothetical top-down evaluation of the query to transform the original rules into equivalent rules that can be implemented efficiently using a bottom-up method.

The formal descriptions of the Magic Set algorithms is complicated and difficult to understand. The essential ideas of this algorithm are described in the context of the simple linear recursive query shown in Section 2.1.

Recall the two rules and the query from Section 2.1.

r(X,Z) :- flat(X,Z). *(1)*

r(X,Z) :- up(X,Y), r(Y,W), down(W,Z). *(2)*

? :- r(a,X). *(3)*

Before we proceed further, we must first define the concept of ***relevant data*** [2]. Intuitively, a piece of data is relevant if it might, depending on the database, be essential to the establishment of a piece of data that is in the answer to the query. Relevant data can, however, be redundant if ~~there is~~ other relevant data which also ~~leads~~ to the same answer. The set of relevant data is defined to be the ***magic set*** in this algorithm.

The first step in transforming the rules is to derive the magic set, which is used as a filter to reduce the size of the relations which will be involved in the bottom-up evaluation. This is done by marking all the ***up*** ancestors of the constant vector *a* and then applying the rules in a bottom-up fashion only to the marked ancestors. For rules (1), (2) and the query (3), the magic set consists of two kinds of relevant data: the ***direct*** relevant data and the ***indirect*** relevant data. The direct relevant datum is the query constant vector *a*, because a set of answers can be retrieved directly from the relation *flat* by using the constant vector *a*. The indirect relevant data is obtained from relation ***up*** of rule (2). The only set of data in relation ***up*** which will potentially lead to an answer is the set of data which is reachable directly or indirectly from the query constant *a*. Therefore, the restricted transitive closure of the relation ***up***, starting with the constant vector *a*, constitutes the set of indirect relevant data of the magic set.

The following two rules define the magic set for this application; they simply define a restricted transitive closure of the relation *up* starting with the constant vector *a*.

magic(a). (4)

magic(Y) :- magic(X), up(X,Y). (5)

The second step in transforming the rules is to apply the magic set to the rules so that the sizes of the relations to be evaluated are reduced. This is achieved by insisting that the values of the first attribute of the virtual relation *r* ~~be~~ in the magic set. Thus, rules (1) and (2) are simply rewritten as follows by adding the magic set to the front of the body of the rules.

r(X,Z) :- magic(X), flat(X,Z). (6)

r(X,Z) :- magic(X), up(X,Y), r(Y,W), down(W,Z). (7).

The four rules (4), (5), (6), and (7), are then compiled into a set of non-recursive query series which will be evaluated using a basic method. Using the same method of compilation as described in section 2.2, the following two compiled formulas are generated.

$\{a\} * up^k$ (8)

$\sigma_a (magic * up)^k \bullet (magic * flat) \bullet down^k$ (9)

where $\{a\}$ denotes the initial relation which contains the query constant vector *a*. Formula (8) produces the relation *magic* which is used in the other formulas. Since the *magic* rules do not refer to the recursive relation *r*, they are evaluated first. The join operations between the relations *magic* and *up* and between the relations *magic* and *flat* can be extracted from formula (9). The complete non-recursive query series which can now be evaluated using a bottom-up method, is shown below.

$\{a\} \bullet up^k$ (10)

$magic \bullet up$ (11)

$magic * flat$ (12)

$\sigma_a (magicup^k \bullet magicflat \bullet down^k)$ (13)

Formula (10) produces the relation *magic*, formulas (11) and (12) produce the relation *magicup* and *magicflat*, and formula (13) generates the answers to the query. Formulas (10) and (13) are evaluated by a bottom-up method called Semi-Naive [1] method. The Semi-Naive method uses the same approach as the Naive method except that at each iteration the query series is evaluated using only the new results which are generated from the previous iteration. This means that rather than passing the entire intermediate relation (i.e. the wavefront relation) produced at the i^{th} iteration to the $i+1^{th}$ iteration, the difference of the relation produced at the i^{th} iteration and the relation produced from the $i-1^{th}$ iteration is passed to the $i+1^{th}$ iteration. The general process flow of executing these two query series is described as follows:

1. Deriving the magic set (i.e. $\{a\} * up^k$)

   Define the wavefront relation, w_i, produced at the i^{th} iteration, as follows:

   $w_i = \{a\}$ $i=0$

   $w_i = w_{i-1} * up - w_{i-1}$ $0<i<=k$

   The processing flow is as follows:

$$\{a\}$$
$$(w_0 * up) - w_0$$
$$(w_1 * up) - w_1$$
$$....$$
$$....$$
$$(w_{k-1} * up) - w_{k-1}$$

2. Deriving the answer to the query (i.e. $magicup^k * magicflat * down^k$)

   The wavefront relation produced at the i^{th} iteration is defined as:

   $w_i = (magicflat)$ $i=0$

   $w_i = (magicup * w_{i-1} * down) - w_{i-1}$ $0<i<=k$

   The general processing flow is as follows:

$$\sigma_a(magicflat)$$

$$\sigma_a(magicup \bullet w_0 \bullet down - w_0)$$

$$\sigma_a(magicup \bullet w_1 \bullet down - w_1)$$

$$\ldots$$

$$\ldots$$

$$\sigma_a(magicup \bullet w_{k-1} \bullet down - w_{k-1})$$

The MS algorithm gives the advantage of binding the processing only to the set of relevant data (the magic set) while avoiding the repetitive processing of the *down* relation, done by the HN algorithm. The performance of these two algorithms is expected to be similar (see [3]) because the duplicate work done by the HN algorithm (in the processing of the *down* relation at each iteration) is offset by the fact that the MS algorithm works with *binary* relations, while HN uses only *unary* relations for intermediate processing. The authors also claim that when *up* and *down* are identical the analytical expressions for the performance of these two algorithms become identical. However, our research shows that this is not the case when the disk I/O performance of the algorithms are considered. The difference is due to the fact that the two algorithms follows different patterns in accessing the three base relations. The accessing order of the base relations of the algorithms will be discussed in detail after the next strategy is presented.

2.3.3 The Counting Algorithm

The Counting algorithm [2] is a smart version of the Magic Set strategy. Recall that the magic set marks all the ***up*** ancestors of the constant vector ***a*** and then applies the rules in a bottom-up fashion to only the marked ancestors. The major drawback of the MS strategy is that the entire magic set~~ (i.e. all the relevant~~ data) is applied to the relation ***flat*** and ***down*** at each step. Thus there is potential redundancy caused by using too much relevant data at each iteration. The design objective of the Counting algorithm is to minimize the set of relevant data at each step.

In the Counting strategy, the ancestors of the constant vector ***a*** are numbered by their distance from it. The magic set, therefore, contains a group of subsets called the ***counting sets***. Each of these counting sets contains the relevant data for a particular level. At each step, instead of using the entire magic set, the strategy uses only the appropriate counting set, thus reducing the set of relevant data at that particular level.
As with the MS algorithm, the counting sets are defined by the following two rules.

counting(a,0). (13)
counting(Y,I) :- counting(X,J), up(X,Y), I=J+1. (14)

where *a* is the constant vector and *I* ranges from 1 to *k* where *k* is the number of desired iterations or the number of iterations up to the termination point. The original rules (1) and (2) are transformed by adding the appropriate counting set to the front of the body of the rules. The new rules are :

r(X,Z,I) :- counting(X,I), flat(X,Z).
r(X,Z,I) :- counting(X,I), up(X,Y), r(Y,W,J), down(W,Z), I=J-1.
and the query becomes:
? :- r(a,X,0).

It turns out that the first attribute of the virtual relation r is redundant. Therefore, the rules can be optimized into:

$r(Z,I)$:- $counting(X,I)$, $flat(X,Z)$. (15)

$r(Z,I)$:- $r(W,J)$, $down(W,Z)$, $I=J-1$. (16)

and the query becomes:

$?$:- $r(X,0)$.

The new set of rules consists of rules (13), (14), (15) and (16), and can be transformed into the following set of compiled formulas. The counting set at the i^{th} level is denoted as $counting_i$ and the derived virtual relations at the i^{th} level are f_i and r_i.

$counting_0 = \{(a,0)\}$ (17)

$counting_i = counting_{i-1} * up$ where $1<=i<=k$ (18)

$f_i = counting_i ** flat$ where $0<=i<=k$ (19)

$r_i = (r_{i+1} ** down) + f_i$ where $0<=i<=k-1$, and "+" is the relational *union* operation. (20)

The processing of these four formulas is divided into three phases. First, all the counting sets are computed by formula (18) using formula (17) as the starting point. Second, formula (19) derives a set of temporary relations which will be used by formula (20). Third, formula (20), initiated by setting f_k to r_k, derives the partial answer to the query at each level. The final answer to the query is contained in relation r_0 after the processing is over. The process flows of these three phases are as follows:

<u>**phase 1**</u>

$\{(a,0)\}$

$\{(a,0)\} * up$

$\{(a,0)\}up * up$

....

....

$\{(a,0)\}up^{k-1} * up$

phase 2

$counting_{k} * flat$

$counting_{k-1} * flat$

$counting_{k-2} * flat$

....

....

$counting_0 * flat$

phase 3

For simplicity, the result relation of each level is denoted by r_i.

$f_k = r_k$

$f_{k-1} + (r_k * down)$

$f_{k-2} + (r_{k-1} * down)$

....

....

$f_0 + (r_1 * down)$

In spite of its elegance, the Counting algorithm does not work if there is *cyclic* data (data directly or transitively rederived by itself) or if there is *asynchronous* data (data rederived at different iteration levels) in the database. This is because in such cases phase 1 of the algorithm, in which the counting sets are computed, will not terminate.

2.4 Accessing Order of Base Relations

There is a fundamental difference among the three algorithms described here which has not been considered in current literature. The difference is the order in which each algorithm accesses the base relations. The HN algorithm involve all three base relations at each iteration. The MS and CN algorithms, by contrast, operate on each of the base relations in turn. In other words, the execution of the HN algorithm proceeds in a *horizontal* manner whereas the execution of the MS and CN algorithms proceeds in a *vertical* manner. Using this distinction, we classify the first group of algorithms as *level-first* and the second group as *stage-first*. The effect of this difference, which will be discussed in detail in chapter 4, can only be realized when the algorithms are evaluated from the perspective of the disk I/Os.

For example, the set of relevant data used by the HN, MS, and CN algorithms is in each case the set of *up* ancestors reachable from the query constant *a*. The processing costs of this part of the three algorithms should be the same if the cost metric is either the number of tuples being processed or the number of page requests. The processing cost is not the same, however, if the cost metric is the number of disk accesses because in HN the accessing of the *up* relation is interleaved with the accessing of the *flat* and the *down* relations whereas in the other two algorithms the accessing of the *up* relation is done all at once. The number of disk accesses needed depends significantly on the size of the available buffer space. This feature of the algorithms is actually a major factor in determining their disk I/O performance.

The MS algorithm is actually not strictly a stage-first algorithm, because although the *up* and *flat* relations are accessed individually, the latter part of the processing involves interleaved accessing of three different relations (*magicup, magicflat* and *down*) at each iteration. Thus, the MS algorithm can be seen as a combination of a stage-first and level-first algorithm. This feature of the MS algorithm further complicates the process of estimating the disk I/Os, as will be shown in chapter 4. In the next section, related work on evaluating the processing efficiency of recursive queries is reviewed.

2.5 Related Work

Although the theoretical aspects of recursive query processing have been studied by many researchers, there has been relatively little work done on evaluating the performance and implementation of recursive query processing strategies which use the compilation approach; In the few performance studies in current literature, different methods and costs metrics were used to evaluate the strategies. Three such studies [2,3,17] are reviewed in this section. The strengths and shortcomings of these studies, which have in part motivated this research, will also be discussed.

In [2], Bancilhon *et al.* discuss informally the performances of four strategies, HN, MS, CN and another strategy, *Reverse Counting*, not considered here. The efficiency of the algorithms is examined in terms of their complexity, which is measured by the predicted number of tuples that are processed. Their study shows that the performance of different strategies depends greatly on the characteristics of the database. More important, they point out that more research is needed to better understand the problem of efficient processing of recursive queries.

A second comprehensive study [3] surveys and compares eleven different algorithms. The algorithms are evaluated analytically on four different queries, including the linear recursive query considered here,[3] and with three predefined database configurations. The size of the intermediate relations generated during execution is used as the cost metric. The results indicate that there are three major factors which influence the processing efficiency: the duplication of work, the set of relevant data, and the arity of the intermediate relation (more generally, the attributes least involved in the intermediate processing).[4]

Although this study presents a general picture of the performances of various strategies, the problem of the efficiency of recursive query processing is not fully explored, for three reasons. First, the qualitative results on the processing efficiency of various strategies needs to be validated quantitatively. Second, previous analysis of the algorithms was based on predefined configurations of data. It is not always possible, however, to know exactly what the data set looks like beforehand. Thus, the algorithms should be evaluated using randomly generated data. Third, using

[3]The linear recursive query is considered to be the most complicated query in the study, in spite of the limitations pointed out here in Section 1.2.

[4]The first factor, duplication of work, has been studied extensively in [15].

the size of the intermediate relations as the cost metrics gives a *static* picture of the algorithm, which is inaccurate because intermediate relations can come and go during execution. In our research, intermediate relations are released immediately when they are no longer needed, and thus, the space requirements of the different algorithms can be evaluated *dynamically*. Nevertheless, this comprehensive study [3] provided us with many rudimentary ideas concerning the implementation and analysis of recursive query processing.

A third performance study, by Han and Lu in [17], is more closely related to our research. Four algorithms (including HN and three other methods not considered here) were examined analytically and experimentally on the same linear recursive query we are studying. The database on which the methods were tested contains randomly generated data. Their evaluation is based on the *selectivity* [5] of the *join* and *select* operations on the relations with the CPU and I/O costs as the cost metrics. They conclude that *performing selection first, making use of wavefront relations,* and *reducing the arity of intermediate relations* [6] are important heuristics for efficient recursive query processing.

The fundamental difference between our research and the third one described above is that we have chosen to study one particular aspect of the processing efficiency of recursive queries, namely the I/O costs. As well, we consider the I/O costs from the perspective of disk I/Os rather than the page I/Os, which are used in [17]. Finally, we examine not only the HN algorithm which is shown to be efficient in [17], but also two other strategies (MS and CN) which are shown to be efficient in [3].

[5]The term *selectivity* will be defined in Section 3.3.

[6]A similar conclusion is drawn in [3].

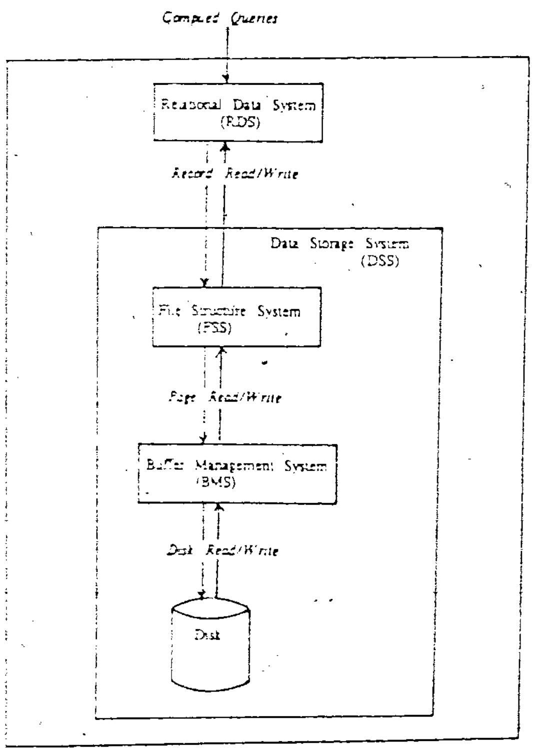
CHAPTER 3

ARCHITECTURE OF THE EXTENSIONAL DATABASE SYSTEM

The previous chapter described various processing strategies for linear recursive queries. That part of the processing can be seen as being mainly carried out in the IDB of the DDB system. This chapter concerns the other part of the DDB system - the Extensional Database (EDB) system. A general model of the EDB along with the different levels of I/O activities involved in executing the query (expressed in terms of relational operations) in the EDB is presented. The implementation of this model, by means of a simulation program, is also described in this chapter. In short, the objective of this chapter is to present the experimental set-up and the specific implementation strategies that are used for low-level operations in the simulation.

3.1 General Model

The Extensional Database (EDB) system consists of two major components: a Relational Data System (RDS) and a Data Storage System (DSS). The latter is further sub-divided into two software modules and a disk in which the database resides. The two software modules are a File Structure System (FSS), which is the interface between the RDS and the DSS, and a Buffer Management System (BMS), which maintains a buffer pool for temporary storage of data relations. The general structure of the model is shown in Fig. 3-1.



Compiled Queries
Relational Data System (RDS)
Record Read/Write
Data Storage System (DSS)
File Structure System (FSS)
Page Read/Write
Buffer Management System (BMS)
Disk Read/Write
Disk

Fig. 3-1 The general structure of an Extensional Database System

The input to the EDB system is a query which is expressed in terms of relational database operations. Each module in the model communicates with its adjacent modules through a set of operations as indicated in the diagram. The primary task of the RDS is to execute the various relational operations requested by the query. Data relations are accessed one tuple at a time during the execution of each relational database operation, so each operation is a sequence of *tuple-read* and *tuple-write* requests. These requests are the input to the DSS.

The FSS, which is the top layer of software within the DSS, presents a record-level abstraction to the RDS. Sequential and direct data-accessing methods are the primary functions provided by the FSS. This module is also responsible for creating and destroying files and maintaining structured files such as B-tree index files. The File Structure System communicates with the Buffer Management System in single-page units.

The BMS contains a buffer manager and a buffer pool of pages. The three tasks that the manager is responsible for are: 1) servicing the page requests from FSS, 2) reading pages from and writing pages to the disk, and 3) when the buffer is full, deciding, based on a replacement policy, which page in the buffer should be replaced.

The primary objective of this model is to reflect the different levels of I/O activities involved in executing the query in the EDB system. Three different I/O activities are considered - the *record I/Os, page I/Os*, and *disk I/Os*. Record I/Os [7] are measured as the traffic between the RDS and the DSS. This metric was used in the testing of the implementation as an approximation of the CPU costs and the complexity of various recursive query processing strategies. The correctness of the implementation is ensured simply by comparing the measurement of this metric with the known performance results of the strategies in current literature, such as the three examples discussed in [2].

The second and third I/O activities (the page I/O and the disk I/O) are the primary interests in this research. The page I/O, as shown in Fig. 3-1, is measured as the number of page read/write requests issued by the FSS to the DMS. The disk I/O statistics are gathered by keeping track of the traffic (i.e. the number of pages being read and written) between the the buffer pool of pages in BMS and the disk. This model ignores the effect of *double paging* that is, the effect of

[7]For this particular metric, it is assumed that each tuple of a relation is a logical record in a file. Under this assumption, this metric is simply a measure of the number of tuples being read and written.

demand–paging at the level of the operating system is not considered. This phenomenon of double paging has been studied by a number of researchers (e.g. [20]).

The above model of an EDB system is simulated in this study. Each layer of the model is implemented as a module which consists of a set of subroutines and a set of primitives. The subroutines are written in terms of the primitives defined at that level, and implement the set of primitives of the level above. That is, the primitives at each level are actually subroutine calls to the modules of the next lower level and they provide the channels of communication among adjacent modules. The necessary hardware, such as the system buffer pool and the disk, is simulated by a set of data structures. In the following sections, the implementation of each module in the simulation will be presented. The algorithms that were implemented at each level in each module and the sets of operations which operate across the layers in the model will be described next.

3.2 Implementation

This section describes the implementation details of five simulation modules which correspond to the levels of the EDB model described in the last section. For each module, the strategies employed for the various operations and the format of the primitives will be described. In the following discussion, a *relation* is a two dimensional table, the rows of which are the *tuples* and the columns of which are the *attributes*.

3.2.1 Test Program Module

The test programs are a collection of prototypical implementations of the recursive query processing algorithms. These algorithms are Henschen–Navqi, Magic Set, and Counting algorithms. The details of these algorithms can be found in Sections 2.3. Appendix A contains the pseudocode of these algorithms written in terms of relational database operations. Each of these subroutines implements the compiled form of the recursive queries (i.e. the object program) defined in Section 2.1.

The test program module acts as the starting point of the simulation. A subroutine is selected by the monitor program to resolve the recursive query. The execution of a subroutine is a sequence of relational database operation requests issued to the database operation module. Each of these operation requests specifies an operator, one or two data relations to be operated on, and a data relation where the final results will be stored. The set of operations defined for this module is :

1. Join(A, B, C).
2. Select(A, C, constant).
3. Project(A, C).
4. Diff(A, B, C).
5. Union(A, B).

where A and B are data relations to be operated on, C is the result relation, and *constant* is the query constant. These operations are implemented in the Relational Data module. Along with these operations, a test program can also issue a *FreeSpace* command to release the space occupied by a relation.

3.2.2 Relational Data Module

The Relational Data module contains a set of five subroutines, each of which implements one of the relational database operations listed in the previous section. The set of primitives which underlie the implementation are as follows:

1. Sequential record-read/write.
    a. ReadTuple(name).
    b. WriteTuple(name).
2. Direct record-read/write.
    a. ReadTuple(name, key).
    b. WriteTuple(name, key).

where *name* is the relation to which the read/write applies, and *key* specifies which data tuple is to be read or written.

In the remainder of this section, the implementation of the relational operations is discussed. The relations involved are either unary or binary, and the following notation is used to specify which attribute to consider.

1. att1(A) - denotes the first attribute of the relation A.
2. att2(A) - denotes the second attribute of relation A.

Join

A Join operation is expressed as Join(A, B, C). A and B are the relations to be joined and C is the result relation. The Join is implemented using the *Nested-Loop* join method in which one of the relations is chosen to be the *outer relation* while the other relation is used as the *inner relation.* The algorithm consists of two nested loops. In the outer loop, each tuple from the outer relation is retrieved sequentially. The value of the join column from the retrieved tuple is used as the key to retrieve the matching tuples from the inner relation in the inner loop. To facilitate the matching of tuples a non-clustered B-tree index [4] is built on the join column of the inner relation.

In our implementation, the smaller relation is always chosen to be the outer relation while the larger relation is used as the inner relation. Three cases can occur, depending on the kinds of relations used. These three cases are described below.

Case 1 : A, B and C are all binary relations.

1. join attributes are :- att2(A) and att1(B).
2. result relation C contains att1(A) and att2(B).

Case 2 : A is unary, B is binary, and C is unary.

1. join attributes are :- att1(A) and att1(B).
2. result relation C contains att2(B).

Case 3 : A is unary, B is binary, and C is binary.

1. join attributes are :- att1(A), and att1(B).
2. result relation C contains att1(B), att2(B).

In case 3 the Join operation is equivalent to a sequence of Select operations, using each of the distinct values from relation A as the constant.

Select

The Select operation is expressed as Select(A, B, constant). A is the relation from which tuples are selected, B is the result relation, and *constant* is the value by which the selection is applied. The implementation assumes that att1(A) is the selection column. The result relation can be either unary or binary. Two methods of selection are implemented. First, if an index already exists on the selection column, all the tuples which have the value of *constant* are retrieved using the index file. Second, if there is no index, relation A will be scanned sequentially once and all the tuples which have the value of *constant* are selected and written to relation C.

Project

The Project operation is expressed as Project(A, B). A is the relation to be projected and B is the result relation. Project is implemented by reading all the tuples from relation A once sequentially and then writing all the distinct tuples to relation B. The implementation assumes that the removal of duplicate tuples in relation A is done automatically, and therefore the costs of it is ignored.

Diff

The Diff operation is expressed as Diff(A, B, C). A and B are the relations to be operated on and C is the result relation. The Diff operation maps each tuple from relation A onto relation B and the result relation contains all the tuples in relation A which do not exist in relation B. Each tuple in both relations is read once sequentially. Tuples from relation A which also exist in relation B are eliminated, and all the remaining tuples from A are written to C.

Union

The Union operation is expressed as Union(A, B). In this operation, all the tuples from relation B are appended to relation A. The result relation is in A, and relation B is left intact. In the implementation each tuple from relation B is read once sequentially and is written to relation A sequentially. This implementation is not a *disjoint union* (in which all the tuples in B which are already in A would not be written again to the result relation). In other words, the result relations of a disjoint union only contains distinct tuples from the two original relations. If a disjoint union is desired, the Project operation will have to be applied to the relation A.

3.2.3 File Structure Module

The File Structure module consists of five subroutines and two sets of data structures. Four of the subroutines are interrelated. The first set of data structures is a group of three one-page buffers which are used for the Read operations. The other data structure is a one-page buffer used for the Write operations. The five subroutines are arranged as shown in Fig. 3-2.

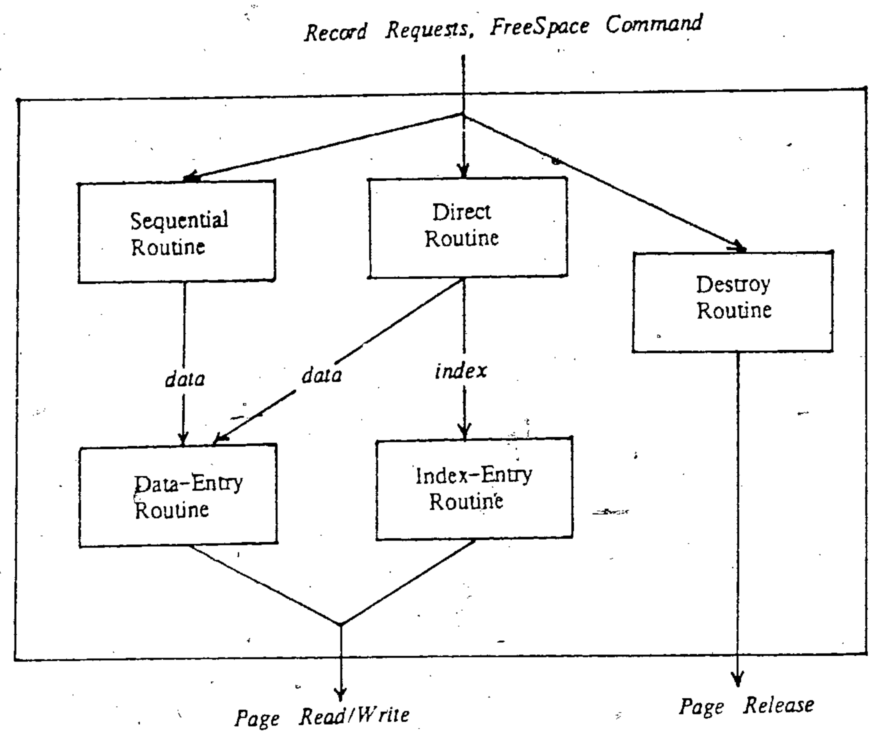


Fig. 3-2 The General Structure of the File Structure Module.

The set of primitives used by the module is a sequence of page operations. They are:

1. PageRead(name, id).
2. PageWrite(name, id).
3. PageRelease(name, id).

where *name* is the relation of which a page is required and *id* denotes the required page. *PageRead* and *PageWrite* read from and write to a single page, respectively. *PageRelease* is used to execute the *FreeSpace* operation, in which all the space occupied by a relation either in the buffer or in the disk are released. This section describes the five subroutines of the File Structure module.

The Destroy subroutine is responsible for issuing a sequence of PageRelease commands to the Buffer Management module when a FreeSpace is received. The Sequential subroutine is responsible for sequential reading and writing of a tuple to a particular page of a data relation. This module interacts only with the Data-Entry routine in order to read or write a tuple. The Direct module is responsible for direct reading and writing of tuples. A B-tree index is implemented for direct data accessing. Two cases are handled by the Direct module. First, if an index on the referenced column of the relation exists, the Index-Entry routine is called to search the index file in order to locate the required data tuple. Once the tuple is located, the Data-Entry routine is called to read or write the tuple. Second, if the index does not exist, a B-tree index is built, by the Index-Entry routine, on the column which is used for the direct accessing. Once the index is built, it is used to read or write the required data tuple as described.

The one-page buffers are used by the Data-entry and Index-entry routines. If the required data tuple or index entry is in the page which is currently in the one-page buffers, no page requests are issued. However, if the required entry is not in the buffers, either a PageRead or PageWrite request will be sent to the next module - the Buffer Management module.

3.2.4 Buffer Management Module

The Buffer Management module consists of a subroutine, which implements the buffer manager, and a data structure which simulates the system buffer pool of pages. The manager is responsible for processing all the page requests from the File Structure module and for swapping pages to and from the system buffer pool and the disk. When a PageRead or PageWrite request is received, the manager tries to locate the required page in the buffer pool. If the page is there, it is sent to the File Structure module in the case of a read, or its content is modified according to the request sent by the File Structure level in the case of a write. If the required page is not in the buffer, the manager decides which page in the buffer can be replaced and issues a PageSwap command to the Disk module. The replacement policy implemented in the simulation is the Least-Recently-Used (LRU) scheme. If the indicated page is in the buffer when a PageRelease command is received, the page will be released; if the page is not in the buffer, the corresponding command will be sent to the Disk module.

3.2.5 Disk Module

The implementation of this module is simple. It consists of a subroutine which is responsible for locating and releasing pages of the disk and a large data structure which simulates the disk. There are two types of information stored on the disk, data relations and index files. Relations or index files are stored in the page units, and it is assumed that no two relations or index files share a page. This may lead to internal page fragmentation during a simulation, a problem which will be discussed in chapter 5. The size of a page is characterized by the number of tuples or index entries it can hold. A page of a unary data relation (i.e. arity = 1) holds twice as many tuples as a page of a binary data relation because the implementation assumes that the length of a unary tuple is only half of a binary tuple. Similarly, a page of an index file is assumed to hold ten times as many index entries as a page of a binary data relation. In other words, the ratio of the number of entries among the binary data relations, unary data relations and index files is 1 : 2: 10. This ratio was arbitrarily chosen, and serves the simple purpose of reflecting the differences in length of the various kinds of data involved in the simulation.

At the beginning of the simulation, three data relations are loaded onto the disk. Throughout the simulation, various intermediate data relations or index files may also be stored on the disk, and at the end, the final result of the recursive query will be stored as a data relation on the disk. In the next section, the synthetic database which is used in the simulation is described.

3.3 The Database

This section describes the synthetic database which is used in the simulation. There are three basic relations, in the synthetic database, referred as *up*, *flat*, and *down*. Each relation contains 1000 tuples and each tuple has two integer attributes. In other words, each relation is a table of 1000 rows, each with two columns of integers. The columns of integers of each relation were randomly generated and are uniformly distributed over a given range.

The database can be characterized by three parameters: the size of the relations, the selection selectivity of the *up* relation, and the join selectivity between relations. These parameters are defined as follows.

1. The size of a relation A, denoted ||A||, is defined to be the number of tuples it contains.
2. The selection selectivity of a relation A, denoted as SSa, is defined as the ratio of the size of the result relation after the selection to the size of the original relation A.
   For example, SSa = ||R|| / ||A||
   where R is the result relation.
3. The join selectivity between the relation A and B, denoted as JSa.b, is defined as the ratio of the size of the result relation after the join to the product of the size of A and size of

B. (Note that A and B can be identical, in which case the join attributes are the two columns of the relation.)

$JS_{a,b} = ||R|| \;/\; (||A|| \times ||B||)$

where R is the result relation.

Four join selectivities (JSup.up, JSup.flat, JSflat.down, and JSdown.down,) between the three basic relations and a selection selectivity (SSup or simply SS) on the relation *up* are used in modelling the synthetic database in this study. To simplify the task of controlling the volume of data during the simulation, we assume that all four join selectivities take on one single value at any one time. Thus, we use a single variable, JS, to denote the join selectivity. By restricting the ranges of the values of the join column and the selection column, we are able to control the join selectivities between relations and the selection selectivity on the relation *up*. The details of how to control these selectivities can be found in Appendix B.

To eliminate the dependency of the results on a particular database, a different database was used for each test run in the simulation. The columns of integers in each relation that make up the database were randomly generated using the UNIX system time, in microseconds, as the seed.

CHAPTER 4
ANALYSIS OF RESULTS

This chapter presents the simulation results regarding the disk I/O performance of the various recursive query processing strategies. The effects of four parameters on the disk I/O performance of the algorithms are examined. The first two parameters, Selection Selectivity (SS) and Join Selectivity (JS), are related to the volume of data that is processed. The other two parameters, Buffer Size and Page Size, are related to the system configuration on which the recursive query strategies were executed. The values of the four parameters used in the simulation were varied as follows:

1. Selection Selectivity (SS) : 0.001, 0.005, 0.05, 0.1, 0.3, 0.5,
2. Join Selectivity (JS) : 0.001, 0.0005, and 0.0001,
3. Buffer Size : 10, 25, 50, 75, 150 (pages).
4. Page Size : 20 tuples/page and 40 tuples/page.

For each distinct combination of values for the four parameters, the simulation program was executed five times, each time with a different set of base relations generated using a different seed for the random number generator. All five algorithms were run against the database, and the page I/Os and the disk I/Os were recorded. Then the average values of each cost metric over the five runs was calculated. The algorithms were also run against some specific databases which were designed to magnify the differences among the three algorithms.

4.1 Page I/O and Disk I/O

In this section, the relationship between the two cost metrics - the page I/O and disk I/O is discussed. The observed differences between disk I/O and page I/O is mainly due to the effect of buffering. A request for an index or data page will require a retrieval of the requested page from the disk only if this page happens not to be in the buffer at that moment. In the following discussion, we make a distinction between *access* and *retrieval*. By *retrieval*, we mean the page is retrieved from the disk, while *access* simply refers to a page request from the program. The size of the buffer is a very important factor in determining the number of retrievals compared to the number of accesses. If the buffer is very large in comparison to the data processed, then the necessary disk-based data pages need to the fetched into the buffer only once. If on the other hand, the buffer can hold only one page of data (or indices) at any one time, the number of disk retrievals is almost the same as the number of page accesses.

The relationships between the page I/Os and disk I/Os of the three algorithms are summarized in Fig. 4-1. The values of JS and SS are fixed at 0.001 and 0.005, respectively, and the page size is 20 tuples per page. Each graph shows the changes in disk I/O, data page I/O, and total page I/O for one algorithm as the buffer size is increased. *Total page I/O* is the total number of pages accessed, including all the data and index pages, while *data page I/O* is the number of page accesses to data relations, and *disk I/O* is the total number of disk retrievals incurred by both data and index pages. The behaviour of the three I/O activities is very similar in all three algorithms and the following observations can be made.

1. The two page I/O curves on each plot are relatively flat compared to the disk I/O curves. This supports the simple fact that the effects of the buffer size are not reflected in the cost metric of page I/Os. While page I/O's remain constant, the disk I/O's decrease as the buffer increases in size.
2. The number of total page I/O's is much greater than the number of disk I/O's in every plot. In contrast, the number of data page I/O's is much closer to the number of disk I/O's. This is because a large number of the page accesses required during the execution are to index pages, which are likely to be found in the buffer. This particular aspect of the page I/O metric will further be discussed in the next chapter. More important, the plots show that the number of data page accesses is closely related to the number of disk retrievals in all three algorithms. This fundamental observation will be used in a later section of this chapter to help estimate the number of disk retrievals of each algorithm. Also, notice that when the buffer size is small, (i.e. < 50 pages) the data page I/O curves are closer to the disk I/O curves. As buffer size increases, the two curves become further apart. This implies that when the buffer is not large enough to hold all the necessary data pages, the disk I/O of the algorithms can be estimated roughly by their page I/O's. As buffer size increases, the correspondence between the data page I/O and disk I/O of the algorithms becomes less apparent.

We now consider the differences between page I/Os and disk I/Os of the three algorithms in terms of their performance. For the same set of parameters as before, the performance of the three algorithms, measured by the number of data page accesses and disk retrievals is shown in Fig. 4-2 and Fig. 4-3, respectively. The I/O behaviour of the three algorithms is quite different in these two plots. When measured by data page I/O (Fig. 4-2), the three curves are strictly ordered and never cross. This is to be expected, as the page I/Os of the algorithm should not be affected by changes in buffer size. CN and HN have relatively low data page I/O's while MS incurred more. When measured by disk I/O, on the other hand (Fig. 4-3), the HN and MS curves cross when the buffer size is between 50 and 75 pages. Also, when the buffer is small, the HN curve is similar to the CN curve, while when the buffer is large, the MS curve is similar to that of CN. These two graphs clearly indicate that the performance of the algorithms can appear to be quite different when different cost metrics are used. Also, different algorithms react differently to increases in the buffer size. For most of the simulation experiments conducted in this study, the data page I/O curves of the three algorithms never cross, as in Fig 4-2. When the disk I/Os are considered, the HN and MS curves sometimes cross over as in Fig 4-3. In the reminder of this chapter, we will concentrate on the disk I/O performance of the three algorithms under various data volume and system configurations.

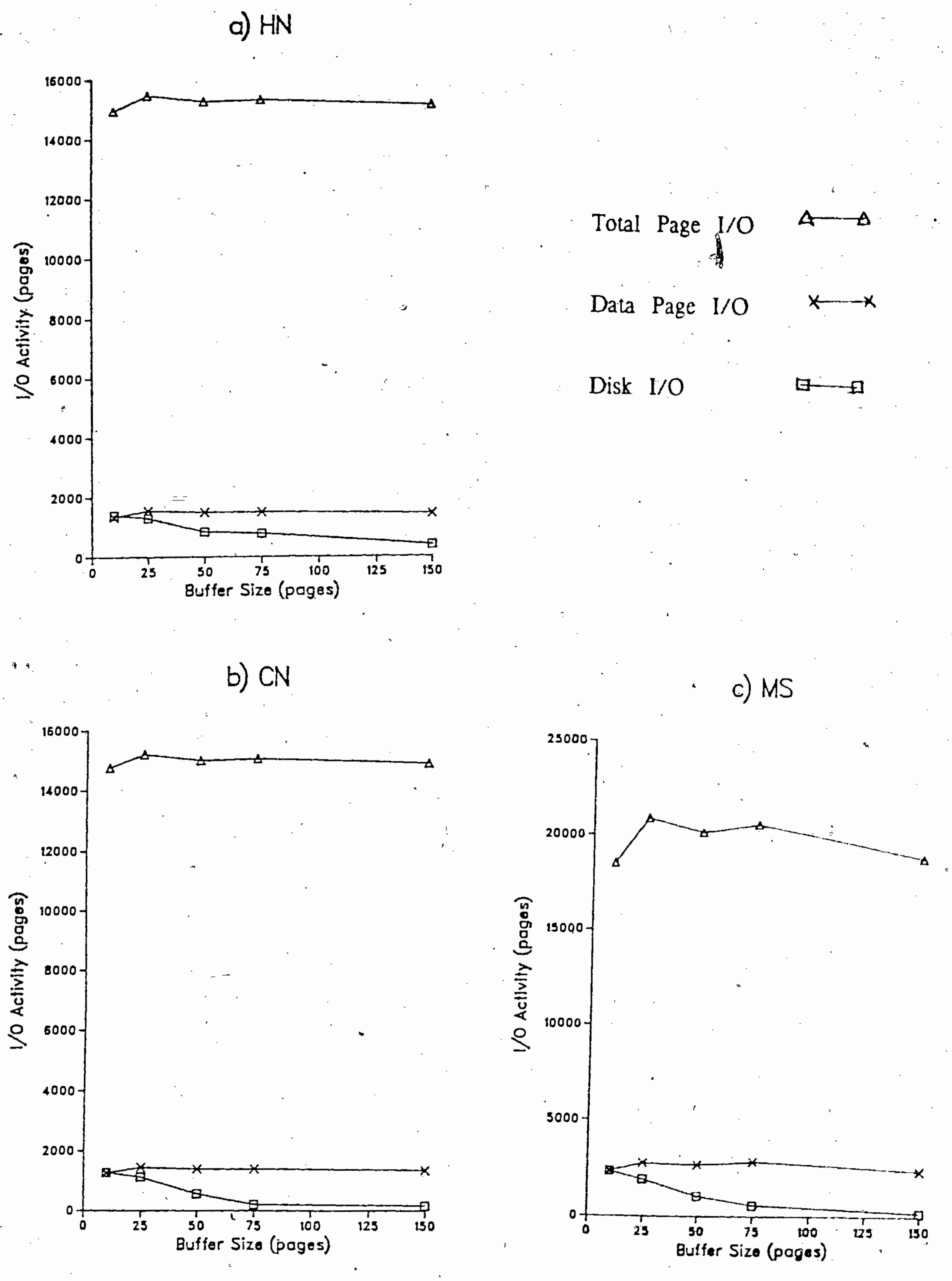


Fig. 4-1 The I/O activities of the algorithms: HN, CN and MS

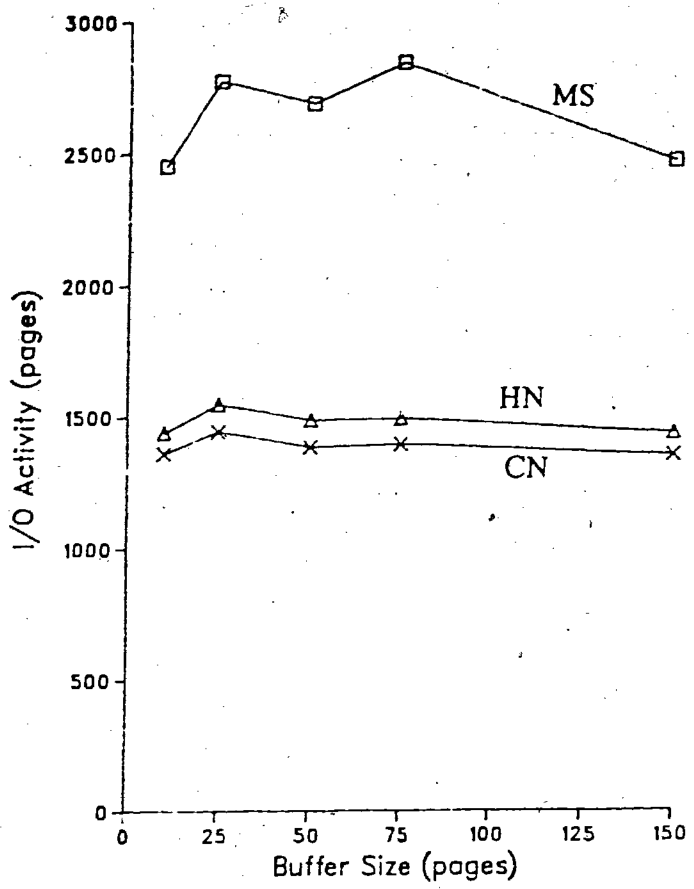


Fig. 4-2 Data page I/O performances of the algorithms: HN, CN and MS

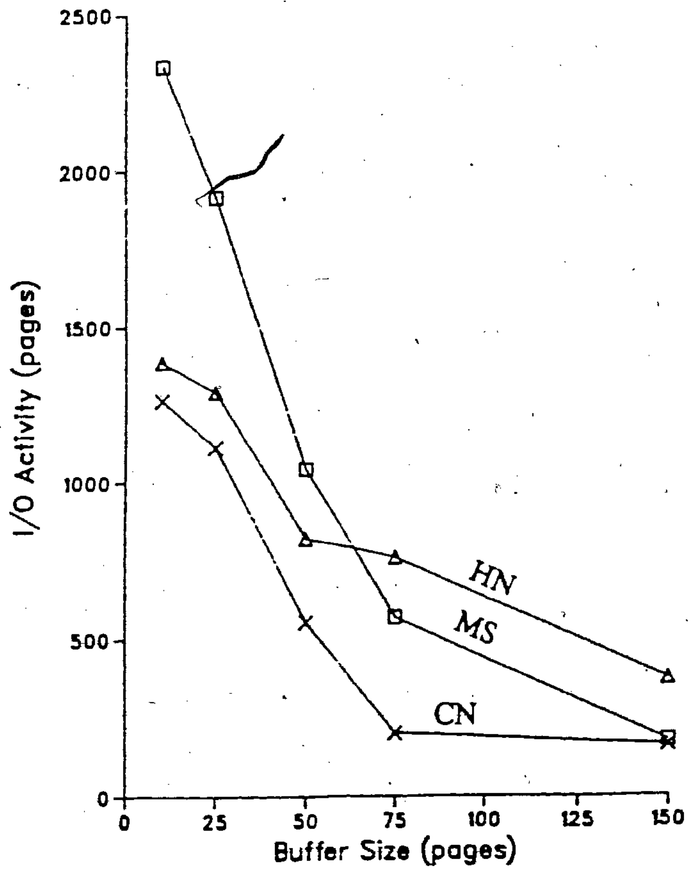


Fig. 4-3 Disk I/O performances of the algorithms: HN, CN and MS

4.2 Disk I/O Performance

In general, for small SS values and small buffer sizes, the simulation results show great variation so that the average disk I/O for each distinct set of parameter values may not present an accurate picture of behaviour. This shows up quite clearly in the plots, where for small buffer sizes, the number of disk I/Os sometimes increases as the buffer increases in size. In retrospect, we should have used the same database to obtain readings for small buffer sizes and small SS values. In this way, the apparent inconsistency could have been eliminated.

The remainder of this chapter is organized as follows. The next section presents and interprets the experimental results obtained from the simulation. Section 4.4 discusses in detail two algorithm-specific factors which may affect the disk I/O performance of the three algorithms. These two factors along with some other less obvious ones are used in Section 4.5 to explain the differences between the disk I/O performance of the three algorithms. The disk I/O performance of the three algorihtms are evaluated on two artificially constructed databases adapted from [2] in Section 4.6.

4.3 Observations and Interpretations

This section presents and discusses observations about the disk I/O performance of the three algorithms HN, CN, and MS with respect to the four parameters described earlier. The algorithms are compared by showing the disk I/Os performance curve of each algorithm as it relates to increase in buffer size. As in the previously described figures, the X-axis represents the buffer size, and the Y-axis represents the disk I/Os (in pages) incurred by the algorithm. Six sets of six graphs each are presented in Fig. 4-4 to Fig. 4-9. The first three sets (Fig. 4-4 to Fig. 4-6) are for simulations in which the page size is 20 tuples/page, while the last three sets (Fig. 4-7 to Fig. 4-9) are for simulations with a page size of 40 tuples/page. Each graph contains one curve for each algorithm. Each set of six shows what happens when the JS is held constant and the SS is varied. For each page size, three JS and and six SS values are tested.

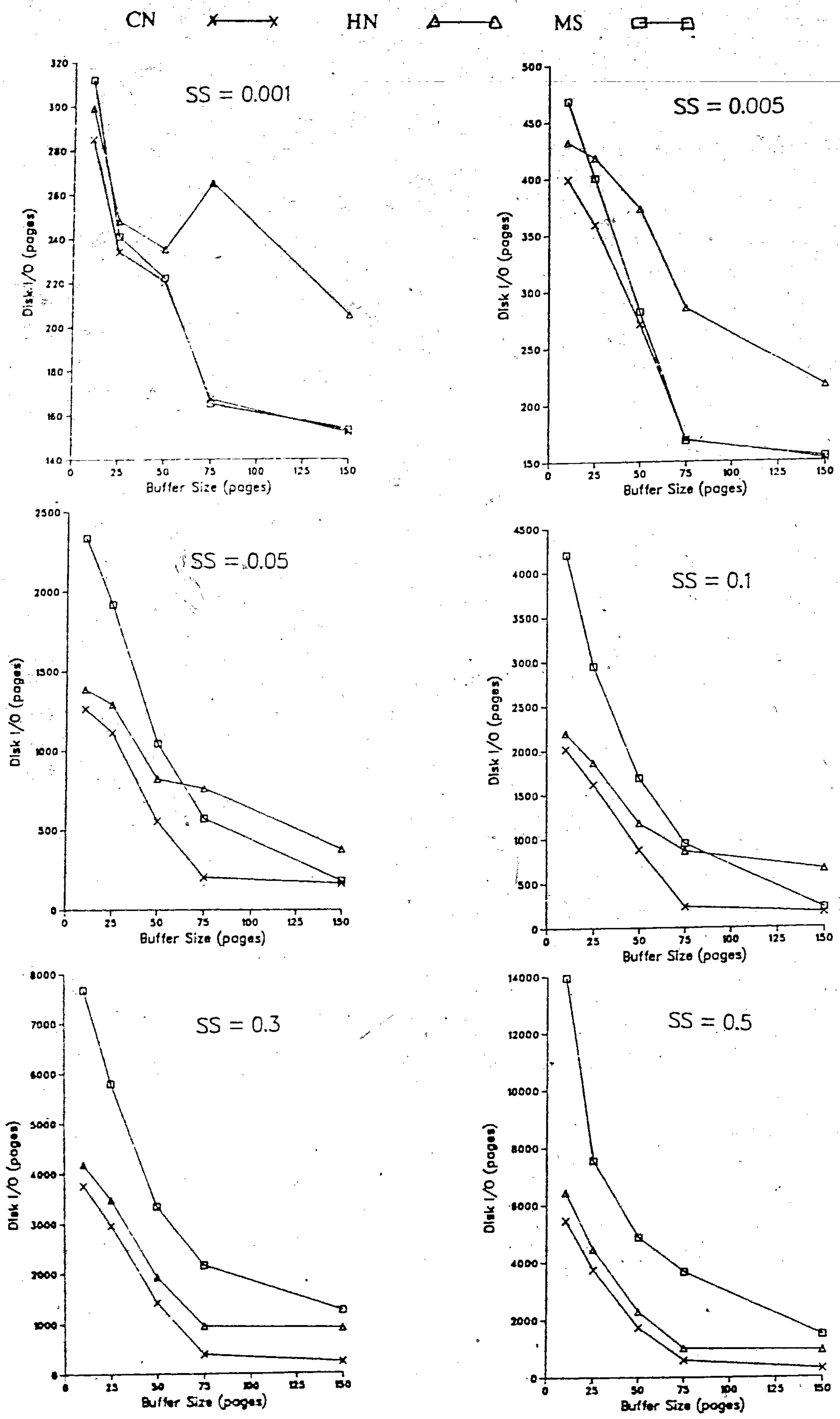


Fig. 4–4 JS = 0.001 and Page-Size = 20 tuples/page

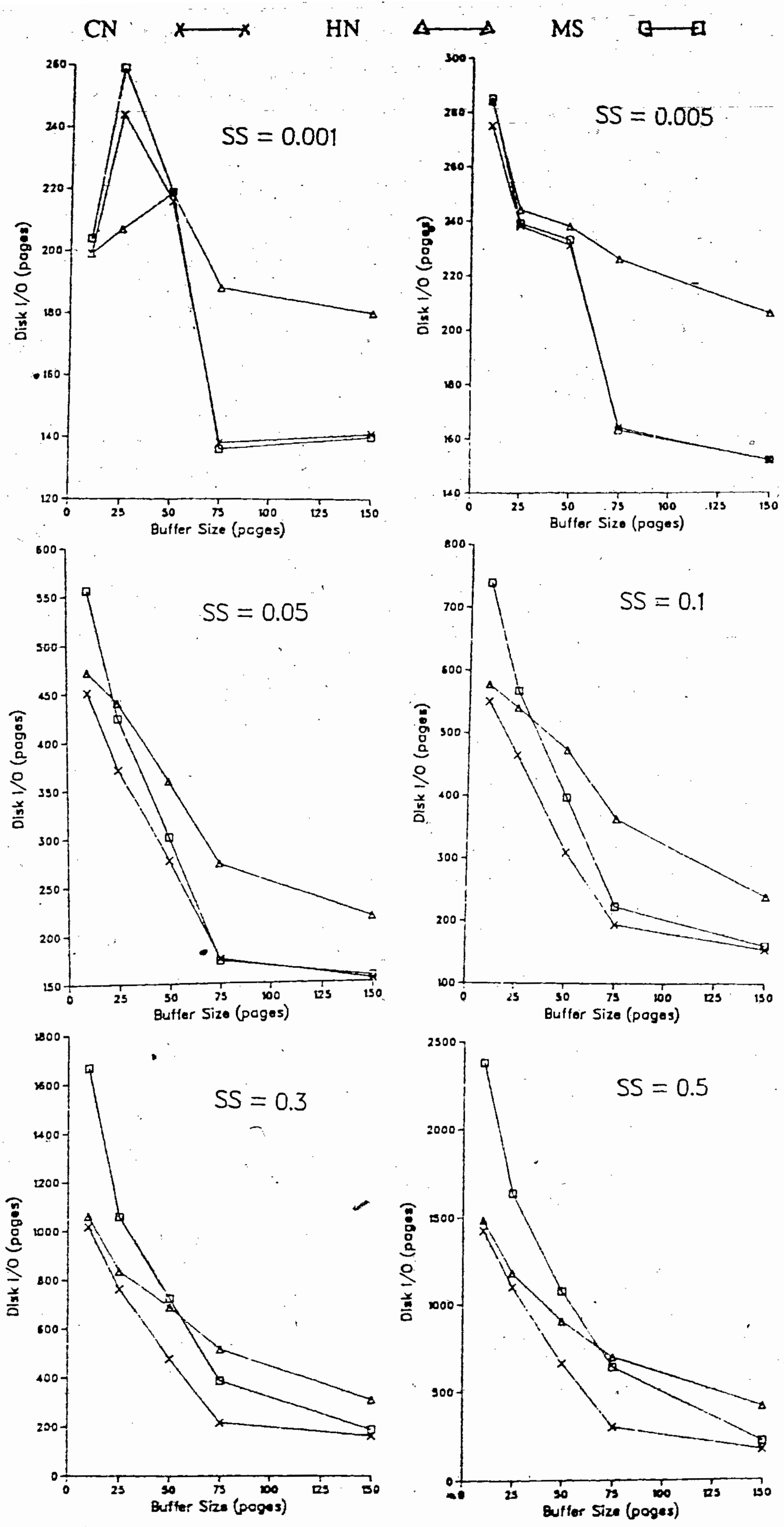


Fig. 4-5 JS = 0.0005 and Page Size = 20 tuples/page

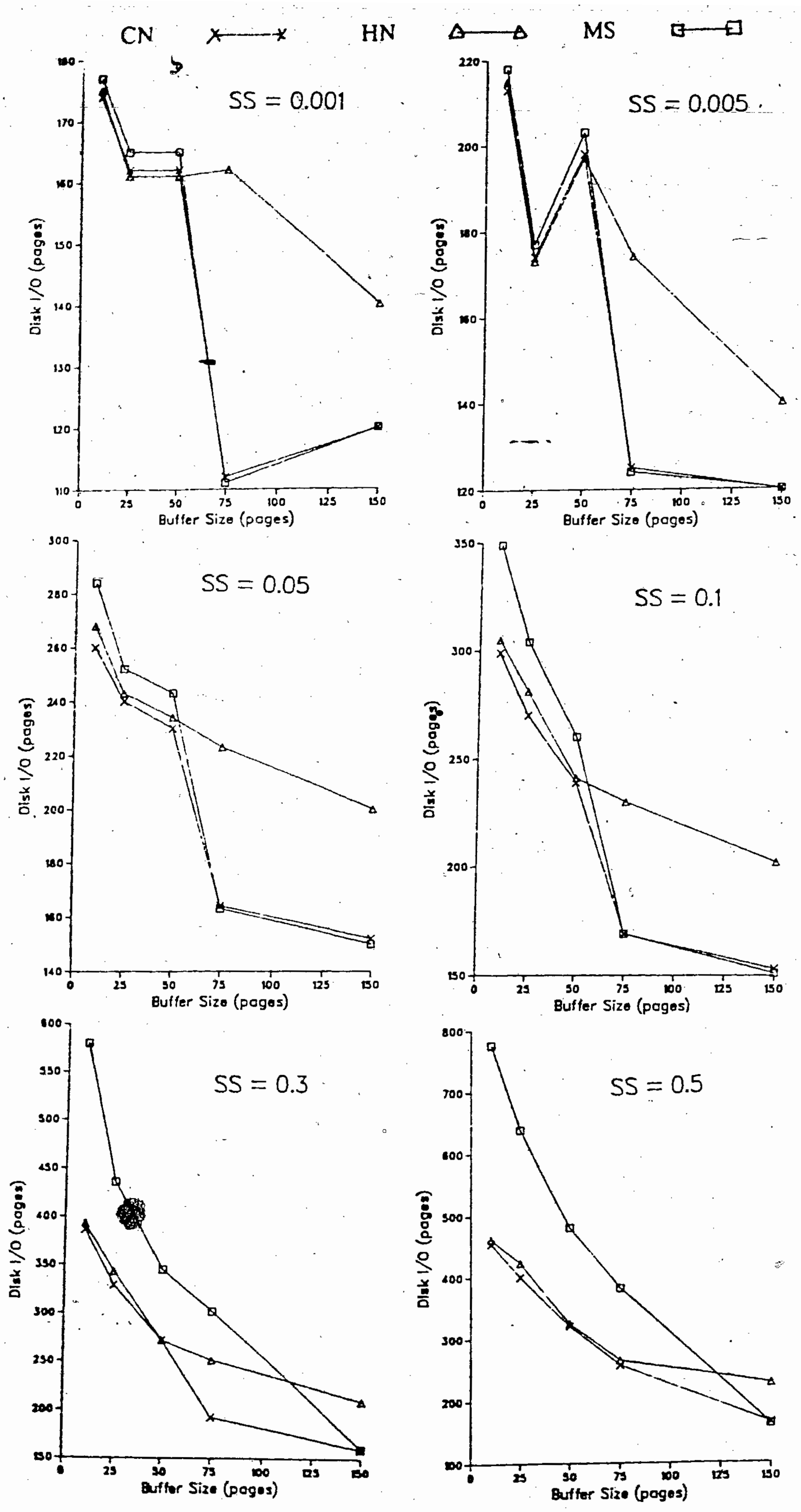


Fig. 4-6 JS = 0.0001 and Page Size = 20 tuples/page

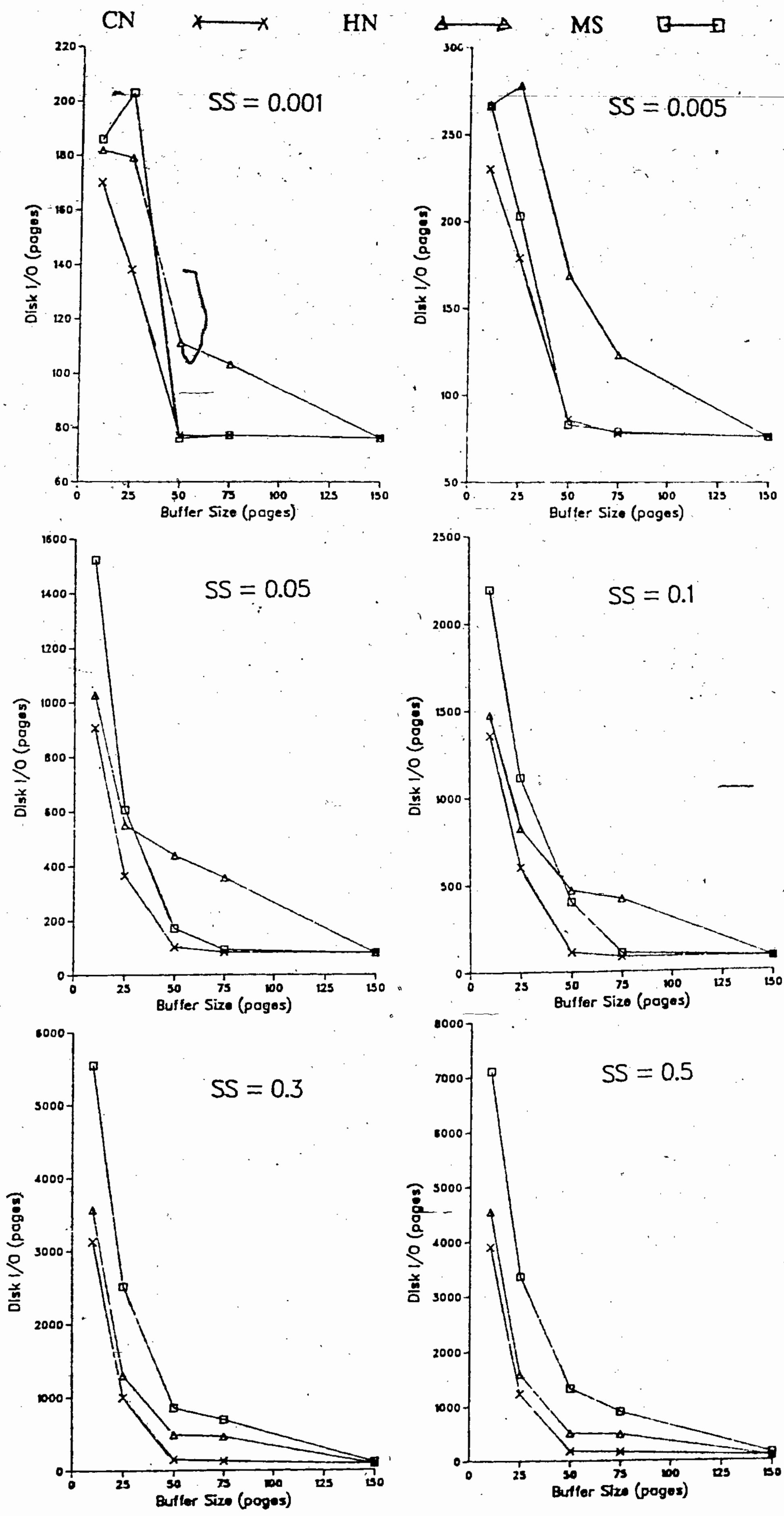


Fig. 4-7 JS = 0.001 and Page Size = 40 tuples/page

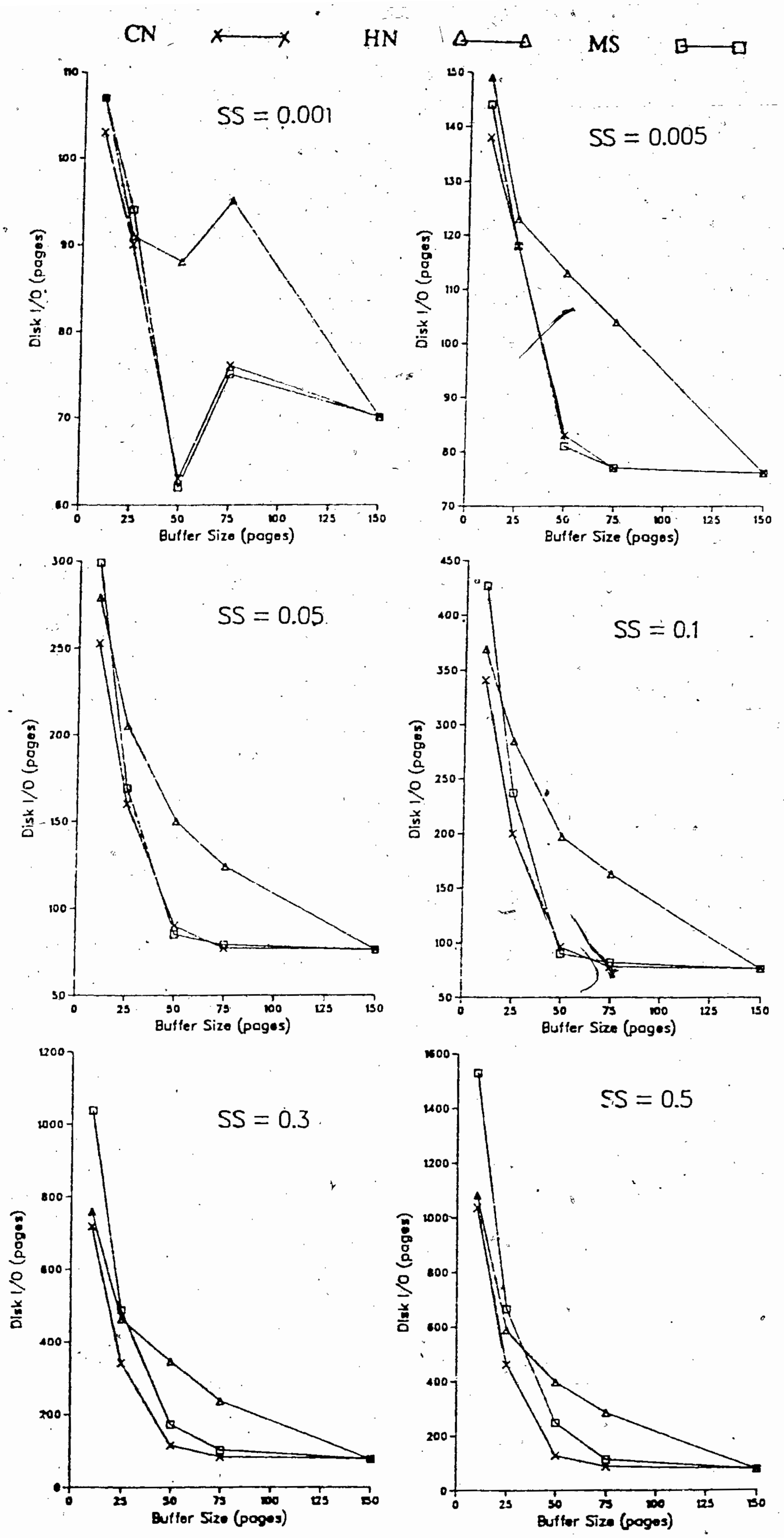


Fig. 4-8 JS = 0.0005 and Page Size = 40 tuples/page

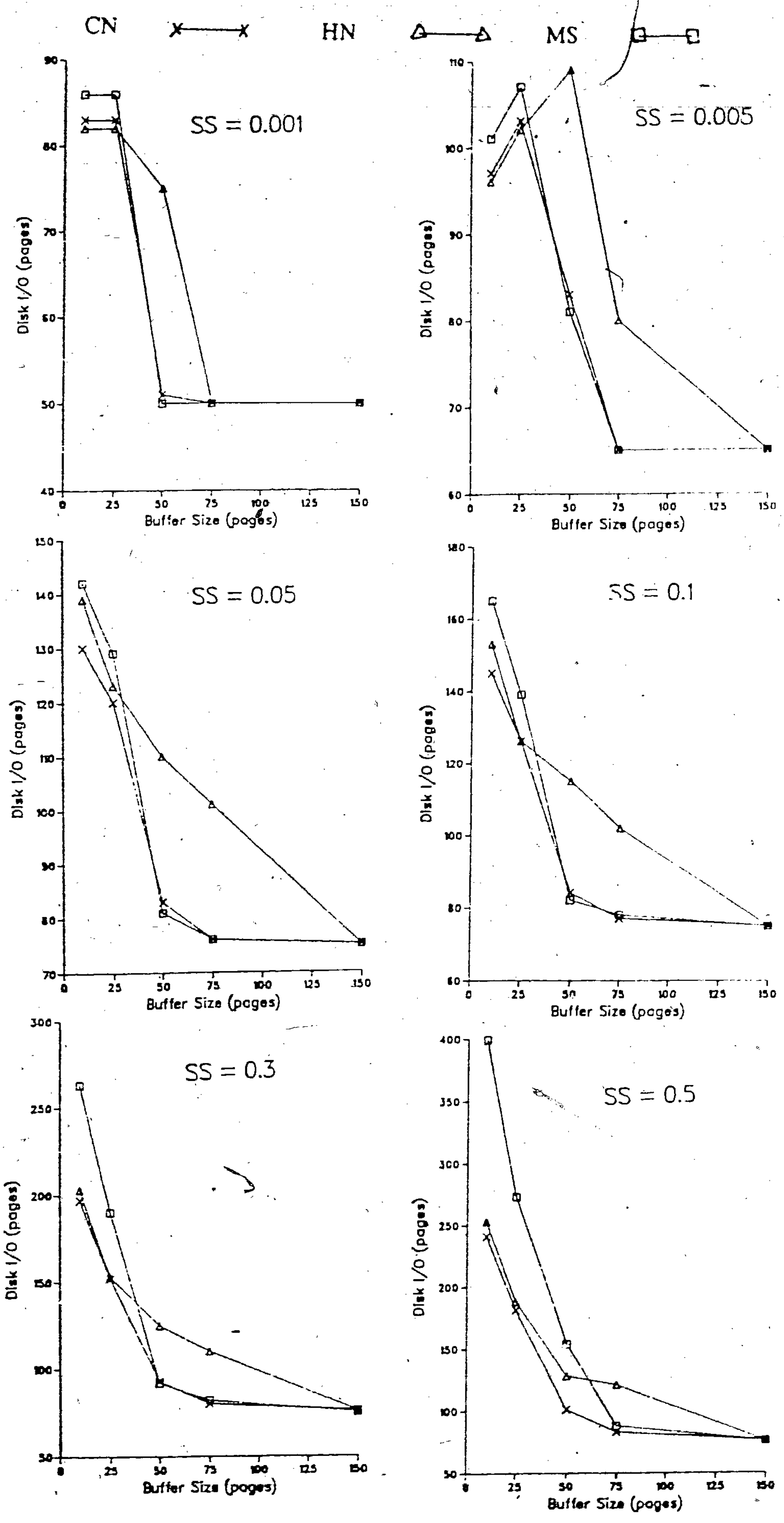


Fig. 4-9 JS = 0.0001 and Page Size = 40 tuples/page

Let us first consider the curves when the page size is 20 tuples per page, and the three base relations *up*, *flat*, and *down* are thus each 50 pages in size. The following observations are made based on the plots of Fig. 4–4, Fig. 4–5, and Fig 4–6.

1. The CN curve is always below the other two curves for all buffer sizes, showing that the CN algorithm outperformed the other two algorithms in all the situations that were considered in the experiments.
2. The HN and MS curves cross over in all plots except for two plots in Fig. 4–4 with high JS and SS values. The MS curve starts high, and moves below the HN curves as the buffer size increases.
3. For each JS value, the MS and HN curves cross over at higher buffer sizes as the SS value increases. When JS is 0.001 (Fig. 4–4 the highest value) the two curves never cross for the two highest SS values (i.e. 0.300 and 0.500).
4. For graphs with low SS values, MS and CN have similar curves even though MS starts at a higher point on the left. As SS values increase, HN and CN curves become similar, and in some cases, parallel to each other.

These observations lead to the following interpretations:

1. The MS algorithm has good disk I/O performances (in the sense of being close to the CN algorithm) when the buffer size is large and the SS values are low. When the data volume handled by the algorithm increases (the combined effect of high SS and JS values), the disk I/O performance of the MS algorithm deteriorates rapidly.
2. The disk I/O performance of the HN algorithm does not respond to large buffer size increases as drastically as the other two algorithms. With a high data volume, the sub–optimal HN performance barely improves when the already large buffer size (i.e. 75 pages in the buffer compared to 50 pages for each base relation.) is doubled to 150 pages.

The algorithms behave in much the same way when the page size is increased to 40 tuples per page (Fig. 4–7 to Fig. 4–9). In fact, they behave as if the page size were only 20 tuples per page and the buffer size were halved. The only difference is that all three curves converge to the same point when the buffer is 150 pages and the page size is 40 tuples. This is because, for these buffer and page sizes, the buffer is large enough to hold all the data and index pages.

required for the entire execution. In this case, all three algorithms require the minimum number of disk I/O's; that is the number required to bring the three base relations into the buffer. Consequently, we will ignore the page size parameter for the rest of this discussion, and treat the page size as a constant value of 20 tuples per page, unless stated otherwise.

4.4 Analysis

The simulation results showed that the disk I/O performance of all the algorithms improves as the buffer size increases. This result is not surprising, as conventional wisdom tells us that the larger the buffer is, the less disk traffic there will be. The interesting question, however, is how does each of the three algorithms react to the size of the buffer? It is obvious from the graphs that not all the algorithms react in the same way to the increase in buffer size. To answer this question, we identify two factors that may affect the disk I/O performance of each algorithm. These two factors are related to the amount of data that the algorithm has to handle and how this data is processed. To this end, we examine the *working set* and the *relation reference pattern* of each algorithm. These two factors are described in the following two subsections.

4.4.1 Working Set

We define two different concepts of working set: *global working set* (WS-G) and *local working set* (WS-L). The global working set includes all index and data pages of all base and intermediate relations generated throughout the whole execution process. A buffer size as large as or larger than the WS-G will never cause any additional disk I/Os over and above the minimum, since each relation needs to be read once. The local working set refers to the index and data pages of the base and intermediate relations that are required for processing a certain part of the algorithm. As described in Section 2.5, each of the three algorithms proceeds in iterations (stages), which may be *stage-first* iterations as in CN or *level-first* iterations as in HN, or a combination of the two as in MS. Thus, the WS-L for each algorithm will be different. The size of WS-L is important because it indicates, roughly, the buffer requirement for each iteration in each algorithm. The Maximum Buffer Requirement (MBR) is defined to be the optimal size of the buffer such that any increase in it will not yield better disk I/O performance. The MBR of an algorithm may be less than the size of its WS-G. For example, if each relation is accessed only once in succession, the MBR for this algorithm is roughly 50 pages (plus space for intermediate and index relations) while the WS–G size will exceed 150 pages. On the other hand, if all three relations are accessed once in each iteration, the MBR is roughly equivalent to the WS–G size. Therefore, MBR is more suitable for the purpose of allocating buffer space for the algorithms, and WS–G is not analysed. We will now consider the WS–L for each algorithm. Recall that intermediate relations that are no longer required for further processing are released immediately.

1. CN

There are three iterations: Up–stage, Flat–stage and Down–stage, in the CN algorithm. Each stage involves the base relation bearing the same name and a number of intermediate relations. In the Up–stage and Flat–stage iterations, one set of intermediate relations is non–disposable because it is required for the next stage of processing. In the final Down–stage, each of the intermediate relations is released immediately after being processed. The size of the intermediate relations (i.e. the Counting sets) created during the Up–stage depends on the SS values. The size of the intermediate relations created during the Flat–stage depends on the size of the counting sets and the JS values. These intermediate relations are utilized during the Down–stage. Thus the WS–L at each stage contains one of the base relations and a number of intermediate relations. This means that the maximum buffer requirement for each stage is roughly the size of the base relation (which is 50 pages when the page size is 20 tuples/page and 25 pages when the page size is 40 tuples/page) plus the size of the intermediate relations. The index tables for the various join and select operations also take up some buffer space, but for 50 or 25 pages of the base relations, the index tables take up only about 2 to 6 pages, and hence will be ignored.

2. HN

The WS–L in the HN algorithm is quite different from that of the CN algorithm, because of the different structure of the two algorithms. Unlike the CN algorithm, the HN algorithm employs a level–first iteration. This means that all three base relations are either entirely or partially involved during each iteration. In our specific implementation, the WS–L of HN can even be different at the different iterations. In the first iteration, the WS–L involves the three entire base relations. This is implementation–specific, as we required index tables to be built for each join operation, and the construction of an index table for a relation requires all the data pages of that relation to be brought into the buffer. Since all three base relations will be joined in turn during the first iteration, the WS-L for this iteration contains at least the three entire base relations. The impacts of this particular aspect of the implementation will be discussed in chapter 5. On top of this, the WS-L of the first iteration includes all the intermediate relations generated during the various operations plus the wavefront relation (created by joining a selection of the relation *up* with itself) that is needed for the next level of processing. For the subsequent iterations, it is difficult to predict the WS-L because it depends on the number of data pages pointed to by the index tables during each join operation. The maximum WS-L is of course all three base relations plus some intermediate relations, but the actual buffer space requirement at each of the second to the fifth levels depends significantly on the SS and JS values, which determine the number of data pages of each base relation which will be needed and the size of the intermediate relations generated.

3. MS

The MS algorithm is different from the other two algorithms in that it uses both the stage-first and level-first iteration methods. The stage-first iteration requires the following pre-processing:

a. generation of the *magic set*,
b. join of the *magic set* with the *up* relation, (i.e. *magicup*) and
c. join of the *magic set* with the *flat* relation (i.e. *magicflat*).

We classify (a) and (b) as Up-stage processing and (c) as part of the Flat-stage processing. After (a) and (b), the *up* and *flat* relations are no longer needed as they are replaced by *magicup* and *magicflat*, respectively. The WS-L for the Up-stage portion involves the *up* relation and the *magic set*, which is the intermediate relation created and passed on to the Flat-stage portion. The size of the *magic set* is determined entirely by the SS values.

After the pre-processing, there are five iterations (levels) of processing involving two intermediate relations, *magicup* at all levels and *magicflat* at the first iteration, and the base relation *down*. The size of the two intermediate relations is determined by the JS and SS values. At each level *i*, there is one join between ***magicup*** and ***magicflat***$_i$ (***magicflat***$_0$ is ***magicflat***), which varies in size from one level to another. Due to our implementation idiosyncrasies, one index table may have to be built at each iteration as the varying intermediate relation may shrink in size from being larger than the fixed intermediate relation (***magicup***) to being smaller than it. The same thing is true also for the join between the join result of these two intermediate relations and the ***down*** relation. For a large magic set (i.e. high SS and JS values), the intermediate relations can be larger than the base relations. Thus, it is very difficult to predict even the maximum buffer requirement of the iteration when the JS and SS values are high. For small data volumes, the buffer requirement at iteration *i* is approximately equal to the size of the two intermediate relations (***magicup*** and ***magicflat***$_i$) and the ***down*** relation plus some of the other intermediate relations generated along the way.

In this section, we have defined the concepts of the global working set and the local working set. The derivation of the local working set of the three algorithms has provided us with some intuitive, although not precise, indications of the buffer requirement of the three algorithms. The local working set of all three algorithms depends, to a certain extent, on the values of SS and JS. In order to gain a better understanding of the buffer requirements of the algorithms, these two variables JS and SS are considered in the next section.

4.4.2 Relation Reference Pattern

The working set analysis described in the last section deals with how each of the algorithms handles the data it has to process. We now consider the amount of data that each algorithm has to handle during the execution. It is obvious that we are dealing with quite complex algorithms. However, a simplified model of execution of each algorithm is sufficient to show the effects of data volume on its disk I/O performance. Thus, a simplified model was developed to show the relation reference pattern of each algorithm. Using this model, we are able to show the sequence of different relations being accessed in various relational database operations throughout the execution of an algorithm. We are also able to quantify the number of data pages that may be accessed from each relation. According to the results of some preliminary test runs, we found that most of the disk I/Os are incurred by three relational operations: *join, select*, and *diff*. Thus, only these three operations are considered in the model. Notice that the *diff* operation is only used in the MS algorithm. The objective of this model is to estimate the MBR of the three algorithms. Our simple model has the following characteristics.

1. Only data pages of base relations are considered. The index pages are not considered because while they account for a large percentage of the page I/Os, they only generate 5% to 30% of disk I/Os, as shown in Fig 4-1. Thus, the probability that a request for an index page will cause a disk I/O is very low in comparison to a request for a data page.[1] Intermediate relations which are small because they are only single column relations (i.e. arity = 1), are not considered. The only intermediate relations that are considered are thus *magicup* and *magicflat*, which are created during the execution of the MS algorithm.
2. When a *select* is performed on a relation, it is assumed that the entire relation is read. This means that all the data pages of a relation will be processed at least once. In our notation, the select operation is represented by *R*, where *R* is the relation in question.
3. When a *join* is performed, if an index does not already exist for the larger relation, one will be built for it. The index construction operation is represented by *R*, where *R* is the relation in question. Like the select operation, this operation requires all the data pages of the relation to be read once.
4. When a *join* is performed using an index to the larger relation, the number of page requests is equal to the number of tuples in the smaller relation. In our notation, the number of page accesses is represented by x(R) where x is the (maximum) number of requests for data pages of R.
5. When a *diff* is performed on relation *A* and *B*, relation *A* is usually quite small. Thus, the number of page requests for this operation is assumed to be the number of data pages of the second relation *B* and it is denoted as *B*.

Based on the above model, we now show for each algorithm the reference pattern of the base and intermediate relations. For simplicity, we assume that JS = 0.001. With this join selectivity, the size of the domain for the join attributes of the two relations is 1000.[2] If one of the two relations involved in the join operation has 1000 tuples (i.e. as do all the base relations in this study), and there exists an index for this relation, then for each tuple in the other relation, there

[1]This aspect of the page I/O and disk I/O is discussed in chapter 5.

[2]The relationship between the join selectivity and the domain of the join attributes is described in detail in Appendix B.

will be on the average one matching tuple in the base relation. Thus, if the smaller relation has *n* tuples, there will be, on average, *n* pages of the base relation requested in the join operation. The extension of the above result to different JS values should be straightforward, although the expression may be much more complicated. The value of variable SS multiplied by 1000 is represented by the symbol *ss*. After the select operation of the constant vector *a* applied to the base relation *up*, the result relation has *ss* tuples. Given the join selectivity of 0.001, when the result relation from the select operation is joined with any base relation, the resulting relation will again have *ss* tuples. The constant 5 will appear frequently in a relation reference pattern since there are 5 iterations to consider. The maximum buffer requirement (MBR) is estimated by the maximum number of page requests between two occurrences of the same relation in the relation reference pattern. The reference pattern and the MBR of the three algorithms are shown in the following.

I. CN

up -> up -> 5ss(up) ->	Up–stage
flat -> 5ss(flat) ->	Flat–stage
down -> 5ss(down)	Down–stage

The up–stage processing starts with a select operation on the relation *up*, followed by an index construction on the same relation and then 5 join operations. This corresponds to the derivation of the counting sets at all 5 levels. In the Flat–stage, there is the index construction on the *flat* relation and 5 join operations with the same relations. Similarly, the Down–stage consists of an index construction on the *down* relation and 5 join operations. The maximum number of data page requests between two occurrences of the same relation in the above reference pattern is 50 pages, which is the size of each base relation. Therefore, the MBR of CN is 50 pages. With this buffer size, each page of the three base relations is read once and only once.

2. **HN**

up -> flat -> ss(flat) -> down -> ss(down) -> up -> ss(up)	First level
ss(flat) -> 2ss(down) -> ss(up) ->	Second level
ss(flat) -> 3ss(down) -> ss(up) ->	Third level
ss(flat) -> 4ss(down) -> ss(up) ->	Fourth level
ss(flat) -> 5ss(down)	Fifth level

The sequence of operations in the first level is: a select operation on relation ***up***, an index construction on relation ***flat***, a join operation on relation ***flat***, an index construction on relation ***down***, a join operation with relation ***down***, an index construction on relation ***up***, and then a join operation with relation ***up***. Each of the second to the fifth levels involves a join operation with relation ***flat***, a number of join operations with ***down***, and a join operation with relation ***up***. In order to provide enough buffer memory so that all three base relations need to be retrieved into the buffer only once, the buffer must be large enough to hold all pages requested between two occurrences of any of the three base relations. Since each base relation contains 50 pages, the MBR of HN is as least 150 pages. Additional buffer requirements for index pages and other intermediate relations are ignored in this model.

3. **MS**

In the following reference pattern, the derived intermediate relation at level i is represented as *magicflat*$_i$ (where *magicflat*$_0$ is *magicflat*) and its size is represented as I_i.

up -> up -> 5ss(up) ->	Up-stage
5ss(up) ->	compute *magicup*
flat -> 5ss(flat) ->	compute *magicflat*
magicup -> magicflat$_0$ *->* I_0*(magicup) -> down -> 5ss(down) -> magicflat*$_0$ *->*	First level
magicflat$_1$ *->* I_1*(magicup) -> 5ss(down) -> magicflat*$_1$ *->*	Second level
magicflat$_2$ *->* I_2*(magicup) -> 5ss(down) -> magicflat*$_2$ *->*	Third level
magicflat$_3$ *->* I_3*(magicup) -> 5ss(down) -> magicflat*$_3$ *->*	Fourth level
magicflat$_4$ *->* I_4*(magicup) -> 5ss(down) -> magicflat*$_4$ *-> magicflat*$_5$	Fifth level

The first three steps correspond to the pre-processing part of the MS algorithm described in section 4.1 of this chapter. In the Up-stage part, the magic set is derived by first performing a select operation on *up*, then building an index for the same relation and then performing 5 join operations with *up*. In the next two steps, the ***magicup*** and ***magicflat*** relations are derived. In the above reference pattern we assume that the magic set contains *5ss* elements, with a maximum value of 1000 (i.e. the maximum size of the ***up*** relation). For tractability, we also assume ***magicup*** is larger than the intermediate relation ***magicflat***$_i$ so that the index for the join between the two relation is on ***magicup***. However, in the actual implementation the index may not always be built on the ***magicup*** relation. This aspect of the MS algorithm is discussed in the next chapter.

The first level starts with an index construction on the relation ***magicup***. The join operation between ***magicup*** and ***magicflat***$_0$ is represented by the terms ***magicflat***$_0$ -> I_0***(magicup)*** in the first level, unlike the join operations in HN and CN. Both of the relations involved in this join are shown in the pattern. This is because, as described earlier as a characteristic of the model, these two intermediate relations are both binary relations. Thus, data page requests on these two relations are comparable to the page requests on base relations. In the rest of the first level, there is an index construction on relation ***down***, a join with this relation, and then a ***diff*** operation involving ***magicflat***$_0$ at the end. In each of the second to the fifth levels, there is a join operation of ***magicflat***$_{i-1}$ with ***magicup***, a join operation with ***down***, and a difference operation with ***magicflat***$_{i-1}$ at the end. The ***magicflat***$_{i-1}$ relation at level *i* is the result of the previous level of processing.

The reference pattern of MS is more complex than that of HN and CN, primarily because of all the intermediate relations at each level. As mentioned earlier, since the page requests of these intermediate relations cannot be ignored, it is more difficult to estimate the MBR of the MS algorithm. A conservative estimate of the MBR, based on the above reference pattern, is the total size of ***magicup***, ***magicflat***$_i$, ***magicflat***$_{i+1}$ and ***down***. While the maximum size of ***magicup*** and ***magicflat***$_0$ (i.e. magicflat) is 50 pages, the maximum size of the other ***magicflat***$_i$ ($i > 0$) is difficult to estimate. For small *ss* values, in general, the MBR is not much more than 50 pages, since ***magicup***, ***magicflat***$_i$, and ***magicflat***$_{i+1}$ will be quite small, and the size of ***down*** is the dominant factor. For large SS values, however, the MBR could well exceed 150 pages.

4.5 Explanation of Experimental Results

We are now ready to explain the differences in the disk I/O performances of the three algorithms based on the analysis of the working set and the relation reference pattern from the last section. In general, CN has a much smaller MBR than the other two algorithms. The MBR of HN is larger than that of CN, but it is more predictable than that of the MS algorithm. When the actual buffer size is large enough (e.g.; 150 pages in the buffer when the page size is 40 tuples), the disk I/O performance of all three algorithms is the same. (see Fig. 4-7, Fig. 4-8 and Fig. 4-9) The disk I/O performance is quite different when the buffer is not large enough for all three relations. In the following discussion, we will examine pairs of algorithms to explain the reasons for the differences in their I/O performance.

1. CN vs HN

CN has a much smaller MBR than HN, which makes it outperform HN by a wide margin when the actual buffer size is greater than the MBR of CN but less than that of HN. This explains why HN always performs poorly in comparison with CN when the buffer size is either 75 pages or 150 pages (20 tuples/page). It should be noted that although we have predicted that the MBR of CN is 50 pages, it does not reach its minimal disk I/O until the buffer is 75 pages. This is because the reference pattern model does not consider the space occupied by index pages and some other intermediate relations. In the actual simulation, therefore, more than 50 pages are actually required by the CN algorithm. When the buffer is 75 pages or 150 pages, the disk I/Os of CN are not seriously affected because these buffer sizes are well beyond the MBR of CN.

When the buffer is smaller than the MBR of both CN and HN, the two algorithms should perform quite similarly because the two algorithms have about the same number of page requests if the difference in the reference pattern is ignored. They do not perform identically, however, and this is due to three minor differences between the two algorithms which do not show up in the reference pattern. They are:

a. removal of duplicates during the down-stage by CN,

b. reference locality,

c. the number of intermediate relations.

(a) has been cited by a few researchers as the main reason for the performance difference of CN and HN (e.g. [23]). However, the difference persists even for small SS values which produce few duplicates from one level to the next. For larger SS values, it is difficult to attribute the large performance differences in the Down-stage entirely to the duplicate removal. Our experiments show that (b) and (c) also contribute to the difference in the actual number of page requests between the two algorithms. CN has an edge over HN with regard to reference locality since the accesses to the base relations are localized in CN. Also, our present implementation of HN requires a larger number of intermediate relations than CN. HN employs roughly $O(n^2)$ intermediate relations in comparison to $O(n)$ for CN, where n is the depth of recursion (i.e. 5 in this case). Since each intermediate relations incurs extra page requests, the difference between the disk I/Os of the two algorithms becomes more apparent when the buffer size is small.

2. MS vs HN

As mentioned earlier, the MBR of MS is more difficult to predict than that of the other two algorithms. The MBR of MS depends very much on the JS and SS values, whereas the MBR of HN is always close to the total size of the three base relations. For low JS and/or SS values, *magicup* and *magicflat*$_i$ will be quite small in size, so the MBR of MS will be quite small. This explains why MS has good performance, compared to HN, for low values of JS and SS when the buffer size is large. For larger JS and SS values (i.e when the data volume is high), MS could have a larger MBR than HN, which explains why their performance curves do not cross when JS = 0.001 and SS = 0.300 and 0.500. The shifting of the crossing points from left to right (i.e. in the direction of increased buffer size) is a clear indication of the change in the MBR of MS. When the data volume is very low, the MBR of MS is very close to that of CN. Thus, MS outperforms HN at a small buffer size because the MBR of HN is much larger than that of MS. As the data volume increases, the MBR of MS also increases but it may well still be smaller than the MBR of HN. Therefore, MS outperforms HN at a larger buffer size. Finally, when the data volume is very high, the MBR of MS will exceed the MBR of HN, and the two curves never cross under these circumstances.

We now consider the performance comparison between MS and HN when the buffer size is smaller than the MBR of either. In general, for small buffer size, the page I/Os of the algorithms more or less determine the disk I/O performance. It is clear from the reference pattern that MS generates more page I/Os then either HN and CN for all values of JS and SS. Thus, it is expected that HN should outperform MS when the buffer size is small. This is indeed the case. The curves show that when the buffer size is smaller than the MBRs of both MS and HN, the MS curve is always above the HN curves. The poor performance of MS when the buffer size is smaller than its MBR is also due to the fact that it has a much larger WS-L than HN in the 2nd to the 5th iteration levels. This means that when the data volume is high, the difference in disk I/Os between HN and MS is proportionally much larger than the difference in page I/Os.

There are some situations in which HN does not perform as well as MS when the buffer size is smaller than the MBR of either of them. Notice that the cross over point in the curves of SS = 0.001 and 0.005 (Fig. 4-4) occurs before the buffer size reaches 25 pages, which is smaller than the MBR of MS at that configuration. This means that MS is better than HN for these SS values and a buffer size of 25 pages, because when SS is very small, the disk I/Os of both algorithms are generated mainly at the beginning of the execution (i.e. the first level in HN and the pre-processing parts of MS.) since the relations involved during the latter parts of the execution (i.e. form the 2nd to the 5th levels) are quite small. In such situations, MS will have a better reference locality than HN because the accesses to the base relations are localized. This can be seen by comparing the relation reference pattern of HN at the first level with that of MS during pre-processing. As a result of this localization of accesses, the disk I/Os generated by MS will then be less than that of the HN algorithm.

The above analysis shows that the CN algorithm is generally the best algorithm with respect to disk I/O performance because it has the smallest and most predictable MBR of the three algorithms. While the performance of HN is not as good as that of CN, it is more stable than MS because the MBR of MS varies significantly according to the data volume. An interesting question that remains is "Is CN always better than any other algorithm, with respect to disk I/O performance?" In the next section, this question is addressed with two artificially constructed databases adapted from [2].

4.6 Vulnerability of CN

This section addresses the question of whether CN always performs better than the other two algorithms with respect to disk I/Os. It is shown in [2] that for some specific databases, CN does perform worse than HN and MS in terms of time complexity. We suspect that for some specific databases, CN may well have poorer disk I/O performance than MS and HN, but for different reasons. As we have shown, CN and HN access the three base relations with very different patterns, with MS somewhat in between. CN accesses each relation in turn, one at each level (stage). In contrast, HN access all three relations at each level. Our implementation, while it limits the number of iterations to 5 seems to benefit CN more than the other algorithms, no significant change in the performance was observed when the depth was increased to 7. It is difficult to generate a database randomly that can make CN perform poorly, so we have adapted two sample databases from [2] to enable us to focus on particular aspects of the performance of the three algorithms. The two sample databases and the simulation results are described in the following sections.

4.6.1 Sample Database I

The first sample database contains the following data in the three base relations *up, flat* and *down*, with *n* proportional to the number of tuples in the *up* and *down* relations.

up	(a_i,a_{i+1}), $1<=i<n$,
	(a_1,a_i), $3<=i<=n$.
flat	(a_n,b_n).
down	(b_i,b_{i-1}), $2<=i<=n$.

In the following diagram, it helps to see *up* as "going up": if *up(a,b)*, then we place *b* above *a* and draw an arc from *a* to *b*. We see *flat* as going sideways, if *flat(a,b)*, we place *b* to the right of *a* and draw the arc *a* —> *b*. Finally, *down* represents arcs that go down: when *down(a,b)*, we place *b* below *a* and have an arc *a* —> *b*. For the case of *n* = 5, the sample database can be represented as shown in Fig. 4-10. The dashed lines illustrate facts that will eventually be inferred.

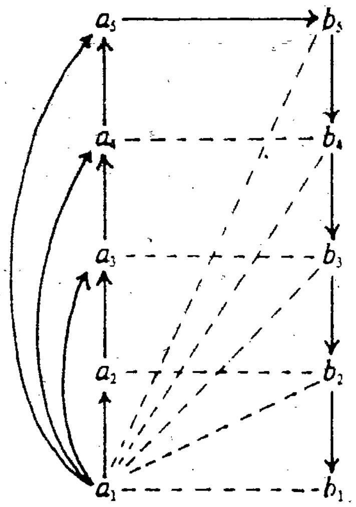


Fig. 4–10 Sample database I for $n = 5$.

In the simulation, n is 25, which means that the processing will continue to depth 25. The linear recursive rules in Section 2.1 are applied to this database with the query

$? :- \ r(a_1,X).$

In this and the next sample database, each page contains 20 tuples.

It is shown in [2] that MS runs in $O(n)$ time against this database while both CN and HN run in $O(n^2)$ time because in computing the transitive closure of the *up* relation, both CN and HN re–derive a large number of common data at each iteration.

The outcome of our simulation of the three algorithms against this database, measured in terms of disk I/Os, is shown in Fig. 4–11. As with the randomly generated database, HN and MS do not perform as well as CN for the small buffer size. As the buffer size increases, MS outperforms the other two for the same reason as stated above. The most interesting observation is that as the buffer size increases to approximately 11 pages, HN, which suffers from the same problem as CN, reaches its optimal performance. This is the point at which all base relations being accessed at each iteration can be accommodated by the buffer. CN, on the other hand, has to store all temporary relations derived at all 25 levels, so its maximum buffer requirement is much larger than 11 pages. As a result, CN will not respond to the increase in buffer size until its MBR, which is much larger than that of HN or MS in this case, is reached.

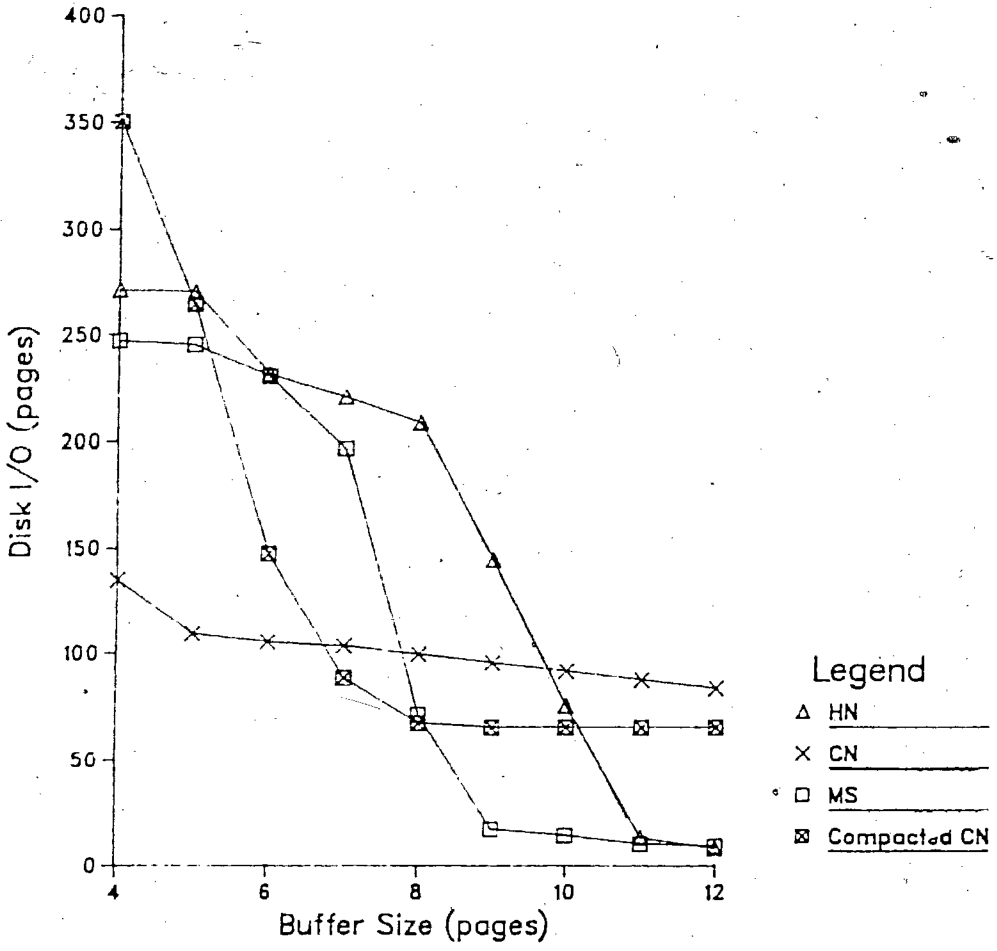


Fig 4-11 Disk I/O performance on Sample Database I

4.6.2 *Sample Database II*

The second sample database is similar to the first one. The data in the three base relations *up*, *flat* and *down* is defined as follows.

up	(a_i, a_{i+1}) $1<=i<n$.
flat	(a_i, b_i) $2<=i<=n$.
down	(b_i, b_{i-1}) $2<=i<=n$.

Using the same notation for the tuples in the base relations, this sample database is represented in Fig. 4-12 for n = 5.

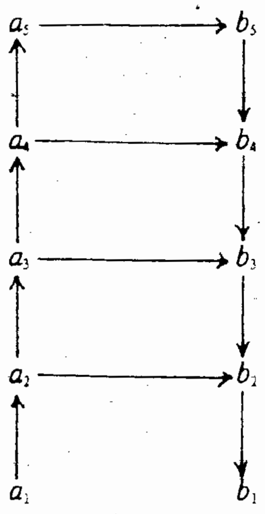


Fig. 4-12 Sample database II for n = 5.

The linear recursive rules and the query of the last sample database are applied to this database. It is shown in [2] that against this sample database, both CN and MS run in $O(n)$ time while HN runs in $O(n^2)$ time. This is due to the removal of duplicated tuples by CN and MS during the processing of the *down* relation.

It turns out from our simulation that MS is the best algorithm of the three to use in this database (see Fig. 4-13). (In fact, the optimal algorithm for this database is Semi-Naive which terminates after the first iteration.) The performances of HN and CN seem to be identical to that for sample database I. This, however, is due to the fact that in our implementation, the intermediate relation generated by CN at each iteration is stored in a separate page to maintain its unary arity. In this sample database, each intermediate relation generated by CN at each level consists of only one tuple, and all of these partially filled pages, accumulated from 25 levels of iteration have to be stored. Thus it is ***internal page fragmentation*** that causes the poor performance of CN in this sample database. If all the intermediate relations are compacted together to produce a single relation of arity 2 (an extra column is needed to record the level number of each tuple), CN (called the Compacted CN in Fig. 4–13) outperforms HN as it no longer suffers from internal fragmentation. This technique does not improve the performance of CN in sample database I, because the poor performance of CN there is due to the large number of data re–derived at each level.

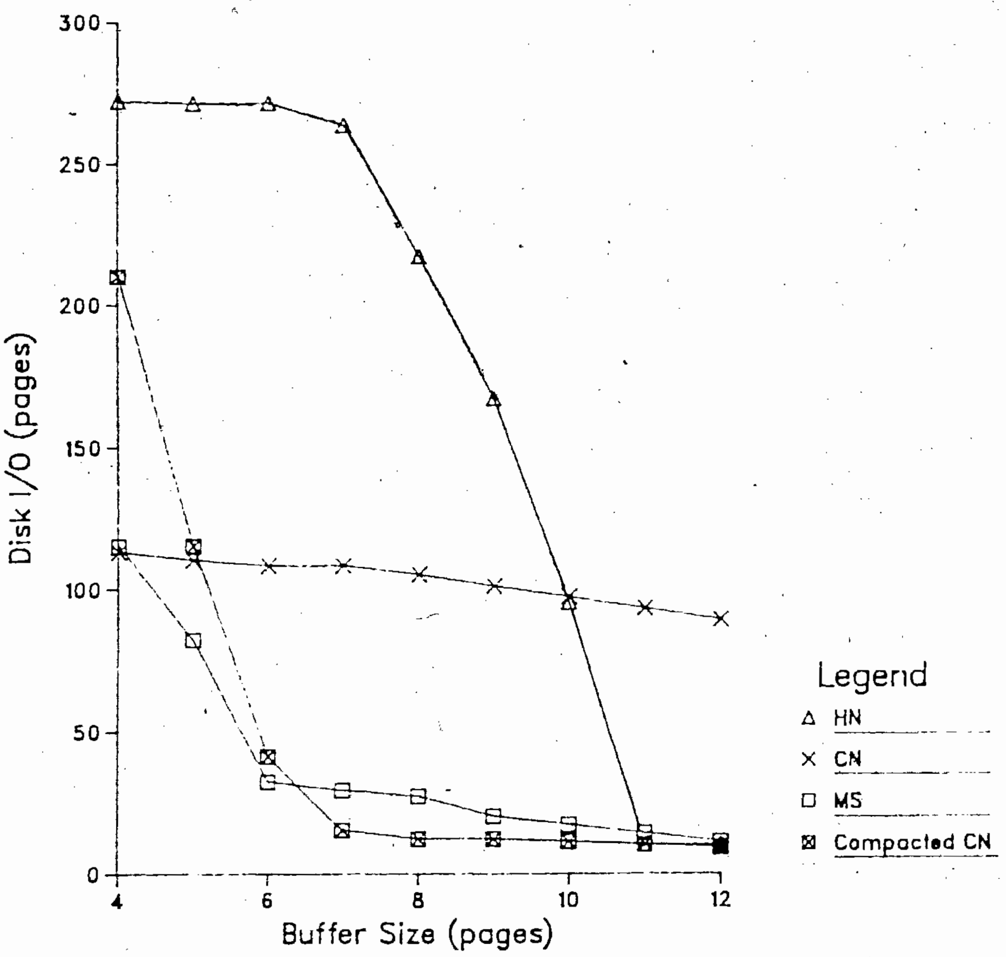


Fig. 4-13 Disk I/O performance on Sample Database II

CHAPTER 5

IMPLEMENTATIONAL ISSUES

Since this research concerns measurement of low-level disk I/Os, we must rely on specific implementations to produce performance data. It is important to consider how our specific implementations affect the disk I/O measurements so that we can assess the degree of dependence of our measurements on the simulation itself. As well, an implementor of such a database may learn useful implementation techniques that will lead to better disk I/O performance. In this chapter, we will discuss, retrospectively, what we have learned from the implementation of the simulation, and alternative methods are suggested for future work. Four general issues will be addressed: the costs of strategies of constructing and accessing index tables of data relations, the impacts of internal page fragmentation, the management of the buffer pages, and the sequencing of relational operations.

5.1 Constructing and Accessing Index Tables

Most performance studies of relational query processing assume the availability of index tables of the relations to be joined, and ignore the costs of building the index tables. This assumption may not be justified in recursive query processing for two reasons. First, several relations must typically be joined in processing a recursive query, and the costs of building the index tables for those relations can be a major part of the final I/O cost. Second, some of the index tables are for intermediate relations which are generated during processing of the query, and it is not reasonable to assume that they already exist. In our implementation, the index tables are not assumed to be readily available, and the costs of building them are included in the measurement of the final I/O costs.

To construct an index table a certain number of page accesses are required to read the data pages of the relation on which the index table is built. In general, it is difficult to predict how many of those page accesses will generate disk retrievals without considering the size of the buffer. However, in the worst case (the data relation is not in the buffer), the number of disk retrievals would be the number of I/O operations required to bring all the data pages of the relation into the buffer. In the algorithms of CN and HN, since index tables are required only for the three base relations *up, flat,* and *down,* the extra disk I/Os incurred, in the worst case, are the number of disk retrievals required to bring all the data pages of the three relations into the buffer. Notice that in our implementation we have not implemented the reading/writing of more than one page in one disk I/O operation directly. Rather, the size of a page is varied to simulate the effect of different volumes of data to be transferred between the buffer and the disk. As discussed in Section 4.2, this particular aspect of the system does not affect the disk I/O performance of the three algorithms significantly.

Constructing index tables with the MS algorithm, however, is more complex, because index tables are required for intermediate relations which are generated at various points during the execution, as well as for the base relations. The fact that some of the intermediate relations may reside in the buffer while some of them may be swapped out and be residing on the disk makes it even more difficult to predict the number of disk retrievals required to construct index tables for the MS algorithm. We can expect, in general, that if the intermediate relations are small, (i.e. JS and SS are low), they will remain in the buffer and the required number of disk retrievals, in the worst case, will be close to the total number of data pages of the three base relations, as in the cases of HN and CN. If the intermediate relations are large or numerous, however, chances are that some of the intermediate relations will be swapped out from the buffer, making it difficult to estimate the required number of disk retrievals.

Another reason for the difficulty of estimating the disk I/O activities needed for the MS algorithm comes from the fact that our implementation adopted the conventional method of always building an index table for the larger of two relations to be joined, creating an unpredictable pattern of index table constructions. As described in Section 4.4.2, one of the two intermediate relations to be joined at each iteration varies in size. If it is the larger one that changes, a different index table may have to be built at each iteration. This can trigger a considerable number of disk I/Os when the intermediate relations consume a large portion of the buffer. Choosing to construct an index table on the intermediate relation which does not vary in size at each iteration would avoid the problem of creating different index tables from iteration to iteration. This was not done because this particular effect of index construction for the MS algorithm was not realized until the implementation of the simulation has been completed. The problem of estimating the costs of constructing the index tables for the MS algorithm is beyond the scope of this thesis and future work is required. However, in the model of relation reference pattern described in Section 4.4.2, it is assumed that the index table is constructed on the particular intermediate relation which is stable in size.

Another interesting observation is that there is a noticeable difference between the numbers of page I/Os and disk I/Os during the creation and accessing of the index tables. For every tuple of the relation, an index entry is created and inserted into the correct page of the index table. The process of insertion, which involves the traversal of the partially built index tables and the writing of the new entry to a particular index page, can trigger a large number of page I/Os, because several PageRead operations and a PageWrite operation are required. Even if an entry happens to be inserted in the same index page as the previous one, another PageRead is required because that particular index page was changed by the previous insertion. On the other hand, while the page I/Os incurred in this process can be very high, the disk I/Os are comparatively low because the process of inserting entries into the index table is localized, and the chances are that most, if not all, of the index pages are still in the buffer. This observation further confirms the belief that there is indeed a difference between the cost metric of page I/Os and the cost metric of disk I/Os.

In the current implementation, when a set of tuples having the same key is required, as when retrieving matching tuples for a join operation, the index entry of each individual tuple is retrieved from the index table, and the data page which is pointed to by the index entry is then fetched. This method has the drawback that two tuples which are on the same data page but whose index entries were retrieved separately will require two separate data page fetches. A feasible solution to this, as briefly mentioned in [12] is to retrieve all the necessary index entries, group them together and sort them in the order of the data page accesses. This will ensure that each required data page will be fetched only once.

5.2 Internal Page Fragmentation

Internal page fragmentation occurs when a data page is occupied by a small number of tuples and most of the space on that page is wasted. This will cause extra disk I/Os if the partially filled pages have to be transferred back and forth between the buffer and the disk. The effect of fragmentation was discussed with respect to the CN algorithm in Section 4.6. Fragmentation occurs in this algorithm when only a small number of tuples is produced at each level and the number of iterations is high. As described in the same Section of Chapter 4, the problem can be rectified by compacting all the data on the partially filled pages into fewer pages. Then all the pages except the last one are full, and the number of pages that have to be transferred back and forth is reduced. However, compaction as a solution is not without extra problems. First, the level number of each tuple must be kept somewhere, which will result in more storage and more complicated processing. Second, compaction itself is costly. Third, it is difficult to predict in advance whether compaction cost is justified because it is not possible to predict the seriousness of the internal fragmentation problem.

Internal page fragmentation can also occur in the implementation of the Union operation. In the current implementation, the union of two relations, A and B, is done by appending all the tuples of relation B to relation A. This method has the drawback that each tuple in relation B has to be read and written once and there may be duplicate copies of these tuples in the buffer or on the disk. One way to get around this is simply to establish a logical link between the two relations A and B, without actually physically joining the tables. This method eliminates the extra read and write operations and seems to be quite elegant, but it also suffers from the problem of internal page fragmentation, which occurs when the union of a large number of relations is required, as when building the magic set in the MS algorithm. If each of the relations contains only a small number of tuples, then simply creating logical links between the relations will cause a problem similar to that described above for the CN algorithm. There must be a trade-off between the two methods of implementing the Union operation, and the relative merits of each method deserve further investigation.

5.3 <u>Buffer Management</u>

It is shown in [36] that it is important for the DBMS and the operating system to communicate because the DBMS knows which data pages are required and which can be destroyed, whereas the operating system uses only a general replacement scheme. In the current implementation, we have adopted a very simple form of communication between the recursive algorithms and the buffer manager: the *FreeSpace* command allows the query algorithms to inform the buffer manager whenever a data relation is no longer needed. The buffer manager can then proceed to free up the space occupied by that relation. While this is a very simple form of communication, some of the preliminary test runs have shown that this command does lead to improved disk I/O performance with various algorithms. This is mainly because when the space occupied by a useless relation is released, not only does that relation not have to be written back to disk, but it also provides space in the buffer for new incoming pages. This technique of allowing the algorithms to have direct control over releasing useless relations has facilitated the derivation of the working set of the algorithms in Section 4.4.1.

Another related issue regarding the management of the buffer pages is the efficiency of the replacement policy used by the buffer manager. The current implementation uses the LRU (Least Recently used) strategy. However, as pointed out in [36], the LRU method does not always provide good buffer management in a database environment. For example, in the case of sequential accesses to pages which will be cyclically referenced, such as a reference pattern of page numbers 1,2,3...,n,1,2,3...n, the LRU strategy is clearly the worst possible replacement algorithm if the buffer is not large enough to hold all *n* pages. Several researchers are actively working to formulate better buffer management techniques (e.g. [9]) in a database environment. A good replacement policy will not only provide better disk I/O performance, but will also reduce the necessary buffer space of a query processing algorithm.

5.4 Sequencing of Relational Operations

For the sake of simplicity we have not paid much attention to improving the disk I/Os by arranging the execution of relational operations in an optimal sequence. To optimize the effects of locality of reference, relational operations which access the same relation should be grouped together. Let us consider as an example the first level of execution of the HN algorithm, found in Section 4.4.2. The current sequence of relational operations is a selection on the relation *up*, followed by a join with the relation *flat*, a join with the relation *down*, and finally the join with the relation *up*. In this sequence, if the buffer size is not large enough to hold all three of the relations, the final join will require the swapping in of the relation *up* after it was already swapped out. This problem can be avoided simply by performing the join operation with *up* right after the selection on it, so that even if the buffer is not large enough to hold all the pages of the three relations, *up* will still be in the buffer, and the join operation with it will not trigger extra disk I/Os. This usage of relation reference ordering is recommended for practical implementation of recursive query processing algorithms.

5.5 Summary of Implementational Impacts

Having discussed the advantages and disadvantages of various implementation techniques used in this study, we now consider their impacts on the overall results of this research. For all three algorithms analysed, indices are built for the three base relations at various points in time. Thus our implementation decision of creation of indices does not affect, in general, the comparative performance of the three algorithms. Of course, in a subtle way, the disk I/O performance of an individual algorithm may be affected because its relation reference pattern may be altered due to the creation of indices. In fact, this is one of the reasons that the performance of MS has a greater variance than those of CN and HN. Index creation and accessing do, however, generate a large number of page I/Os that do not lead to much disk I/O activity. This is one reason why we segregate indices from base relations in calculating the page I/O and disk I/O costs. In fact, for high data volumes, different methods of performing join operations, e.g. *sort-merge*, would probably prove to be more appropriate than indexed join, and further investigation is necessary.

The issue of internal page fragmentation is an example of the importance of properly storing the intermediate relations. This phenomenon, however, only occurs in rare instances and, therefore, it does not affect significantly the comparative performances of the algorithms. For the buffer management issue, it is clear from the discussion in Section 3 above, that LRU does not provide the best buffer management for the index-nested loop join method. In fact, the primitive scheme of *First-In-First-Out* can do as well as the LRU scheme in those situations. However, as all three algorithms are executed using the same join method and the same buffer replacement policy, it may be concluded that the overall relative performance of the three algorithms would not be changed substantially if a different buffer management policy is used.

Finally, on the issue of optimal sequencing of relational operations, it seems that it is rather algorithmic specific. It is difficult to generalize this idea to all the algorithms. Thus this issue should not be a concern of the designers of the algorithm, but rather a system-fine-tuning concern of the implementors.

CHAPTER 6
CONCLUSIONS

This chapter contains a summary of the results of this study, and presents some suggestions for future research. Recall that this research presented two objectives: to study the relationships between page I/O and disk I/O using some linear recursive query processing algorithms as examples, and to compare the performances of some promising linear recursive query processing algorithms at the disk I/O level.

6.1 Page I/O and Disk I/O

In this research, we have examined the similarities and differences of the two I/O cost metrics in relational database query processing: page I/O and disk I/O. The relationships of these two metrics are listed as follows.

1. While the number of page I/O requests generally corresponds to the amount of disk traffic, this correspondence becomes less when the buffer size increases. When large buffer space is available, it is important to include the disk I/O as an additional parameter in the cost formula of a query processing strategy. Algorithms that behave very similarly when measured by their page I/O activities may have very different disk I/O performance as shown in Section 4.1.
2. It is important to estimate the maximum buffer requirement (MBR) of a query processing algorithm in order to allocate the proper amount of buffer space to ensure good disk I/O performance of the algorithm. This also ensures that buffer space required for other simultaneous activities is not wasted.
3. A simple methodology is developed to estimate the maximum buffer requirement of a query processing algorithm. This methodology has at least in this case study proven to be useful.

6.2 Linear Recursive Query Processing Algorithms

The efficiency of three recursive query processing strategies were evaluated with a set of linear rules from the perspective of their relative disk I/O performance. A number of interesting characteristics of the three algorithms were discovered in this study.

1. There is a fundamental difference in processing between CN and HN. CN proceeds by stages (Up, Flat and Down) and, for each stage, only one base relation is accessed and all necessary relations are derived and passed on to the next stage. In contrast, HN proceeds by iterations (5 iterations altogether in this study) and for each iteration, all three base relations (*up, flat* and *down*) are processed.[11] MS combines aspects of both approaches.
2. Largely because of the above differences, CN has much superior disk I/O performance because it has a much smaller maximum buffer requirement (MBR) compared to the other two algorithms. However, if the relations involved are relatively small and the number of iterations is relatively large, it may have the worst performance. For HN and MS, the disk I/O performance of HN is more predictable than that of MS in the sense that the MBR of HN does not depend on the data volume as much as the MS algorithm does.
3. If the size of the query constant (as a vector) is small, any of the three algorithms has similar disk and page I/O performances (in absolute terms since the charts can be deceiving as the Y-axis representing the performance does not always starts from 0). This is true for all buffer sizes and join selectivities.
4. If the size of the query constant is large, not only is it important to choose the right recursive query algorithm, but it is equally essential to choose the right strategy for implementing low-level operations such as relational database operations.
5. To decide on the buffer allocation scheme, one has to have a good estimate of the size of intermediate relations. This has shown to be a difficult task. Also, most of the relevant works in the literature are concerned with estimation of the size of a single join of two relations. However, it was found that in recursive query processing, the estimation of the sizes of intermediate relations resulting from successive joins of relations is equally important.

[11] A close analogy exists in the way a 2-dimensional array is internally stored in a linear array. Fortran compilers adopt a column-first approach while PL/1 compilers adopt a row-first approach.

6.3 Suggestions for Future Research

1. In this research we presented the importance of the disk I/O performance in query processing. The ultimate goal is to develop a new I/O cost analysis model in which not only the page I/O is considered, but disk I/O and buffer management schemes are also evaluated. Along the same line, developing a better communication interface between the DBMS and the operating system is also an area which should be studied.

2. As was pointed out in this study, the creation and accessing of index tables can be problematic in the index-nested loop join method. Other join methods such as the *sort-merge* and *hashed-based* join method should also be examined from the perspective of disk I/Os.

3. For recursive query processing strategies, although the CN algorithm has the best disk I/O performance, its application domain is limited to linear recursive rules and non-cyclic database. Other similar strategies such as the *Level-Cycle Merging* algorithms [16] have been developed to overcome this drawback of the CN algorithm. These algorithms should be tested to determine whether they preserve the elegance of the CN in accessing the base relations while the application domain is expanded. For the MS algorithm, the great variance in its disk I/O performance is partly due to the underlying basic method - the Semi-Naive method. Other methods should be tested with the MS algorithm to determine if the disk I/O performance of it would improve.

4. In order to predict the MBR of a recursive query processing algorithm more accurately, analytical methods should be derived to calculate the size of the intermediate relations produced from successive join operations rather than simply estimating the size of the result of one single join operation. Finally, the disk I/O performance of non-linear or more complex recursive query algorithms should be investigated.

REFERENCES

1. Bancilhon. F., "Naive Evaluation of Recursively Defined Relations" *On knowledge Base Management Systems*, Brodies and Myplopoulos. Eds., Springer-Verlag, Berlin, New York 1986.

2. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proceedings of the 5th ACM Sysmposium on Principal of Database Systems*", ACM-SIGACTSIGMOD, March, 1986.

3. Bancilhon, F., and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies", *Proceedings of SIGMOD 1986 International Conference on Management of Data*, ACM, May, 1986.

4. Bayer, R., and McCreight, E., "Organization and Management of Large Ordered Indexes", *Acta Informatica*, Volume 1, Number 3, 1972, pp. 173-189.

5. Brodie, M. L., and Mylopoulos, J., *On knowledge Base Management Systems - Integrating Database and AI Systems*, Springer-Verlag, Berlin and New York, 1986.

6. Bitton, D., "The Effect of Large Main Memory on Database Systems", *Proceedings of SIGMOD 1986 International Conference on Management of Data*, ACM, May, 1986.

7. Chakravarthy, U. S., Minker, J., and Tran, D., "Interfacing Predicate Logic Languages and Relational Databases", *Proceddings of the 1st conference on Logic Programing*, Marseille, France, September, 1982, pp. 91-88.

8. Chang, C., "On the Evaluation of Queries Containing Derived Relations in a Relational Data Base", *Advances in Data Base Theory*, Volume 1, H. Gallaire, J. Minker, and J. Nicolas, Eds., Plenum Press, New York, 1981.

9. Chou, H., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Proceedings of the 11th International Conference of Very Large Data Bases*, Stockholm, August, 1985, pp. 127-141.

10. Gallaire, H., and Minker, J., *Logic and Databases*, Plenum Press, New York, 1978.

11. Gallaire, H., Minker, J., and Nicolas, J., "Logic and Database : A Deductive Approach", *ACM Computing Survey*, Volume 16, Number 2, 1984, pp. 153-195.

12. Hagmann, R. B., "An Observation on Database Buffering Performance Metrics", *Proceedings of the 12th International conference on Very Large Data Bases*, Kyoto, August, 1986, pp. 289-293.

13. Han, J., "Pattern-Based and Knowledge Directed Query Compilation in Recursive Data Bases", *Technical Report No. 629*, Computer Science Department, University of Wisonsin-Madison, January, 1986.

14. Han, J., and Henschen , L. J., "Stack-Directed Compilation of Complex Recursive Rules in Deductive Databases", *Technical Report 86-04-AI-01*, Northwestern University, Department of EE/CS, April, 1986.

15. Han, J., and Henschen, L. J., "Handling Redundancy in the Processing of Recursive Database Queries", *Proceedings of 1987 ACM-SIGMOD Conference on Management of Data*, May, 1987.

16. Han, J., and Henschen, L. J., "Processing Linear Recursive Database Queries by Level and Cycle Merging", *Technical Report 87-05-DBM-01*, Northwestern University, Department of EE/CS, May, 1987.

17. Han, J., and Lu, H., "Some Performance Results on Recursive Query Processing", *Proc. International Conference on Data Engineering*, IEEE, February, 1986.

18. Henschen, L. J., and Naqvi, S., "On compiling Queries in Recursive First-Order Databases", *Journal of ACM*, Volume 31, Number 1, 1984.

19. Kaplan, J., "Buffer Management Polices in a Database Environment", *Master Report*, UC Berkeley, 1980.

20. Lang, T., Wood, C., and Fernandez, E. B., "Database Buffer Paging Virtual Storage Systems", *ACM Transactions on Database Systems*, Volume 2, Number 4, December, 1977.

21. Mckusick, M., Joy, W., Leffer, S., and Fabry, R., "A Fast File System for UNIX", *ACM Transactions on Coumputer System*, Volume 2, Number 3, August, 1984, pp. 181-197.

22. Minker, J., "Performing Inferences Over Relational Database", *ACM SIGMOD International Conference on Management of Data*, San Jose, Calif., May, 1975, pp. 79-91.

23. Minker, J., "Set Operations and Inferences Over Relational Database", *Proceedings of the 4th Texas Conference on Computing Systems*, Austin, Texas, November, 1975, pp. 5A1.1-5A1.10.

24. Minker, J., "Search Strategy and Selection Function for an Inferential Relational System", *ACM Transactions on Database Systems*, Volume 3, Number 1, 1978.

25. Minker, J., "An Experimental Relational Database System based on Logic", *Logic and Database*, H. Gallaire and J. Minker, Eds., Plenum, New York, 1978, pp. 107-147.

26. Minker, J., Nicolas, J. M., "On Recursive Axioms in Deductive Databases", *Informaton System*, volume 8, Number 1, January, 1982, pp. 1-13.

27. Nyberg, C., "Disk Scheduling and Cache Replacement for a Database System", *Master Report*, UC Berkeley, 1984

28. Reiter, A., "A Study of Buffer Management Polices for Data Management Systems", *Technical Summay Report No. 1619*, University of Wisconsin-Madison, Mathematics Research Center, March, 1976.

29. Reiter, R., "Deductive Question-Answering on Relational Databases", *Logic and Data Bases*, H. Gallaire and J. Minker, Eds., Plenum, New York, 1978, pp. 149-178.

30. Reiter, R., "On Closed World Database", *Logic and Data Bases*, H. Gallaire and J. Minker, Eds., Plenum, New York, 1978, pp. 56-76.

31. Reiter, R., "Towards a Logical Reconstruction of Relational Database Theory", *On Conceptual Modelling*, M. Brodie, J. Mylopoulos, and J. W. Schmidt, Eds., Springer-Verlag, Berlin and New York, 1984.

32. Sacco, G., and Schkolnick, M., "A Mechanism for Management the Buffer Pool in a Relational Database System Using the Hot Set Model", *Proceedings of the 8th International Conference of Very Large Data Base*, Mexico City, September, 1982, pp. 257-262.

33. Sagiv, Y., and Ullman, J. D., "Complexity of a Top-Down Capture Rule", *Technical Report STAN-CS-84-1009*, Standford University, Department of Computer Science, 1984.

34. Selinger, P., Astranhan, M., Chamberlin, D., Lorie, R., and Pice, T., "Access Path Selection in a Relational Database Management System", *Tecnnical Report No. RJ2429*, IBM, San Jose, 1979.

35. Shapiro, S. E., and McKay, D. P., "Inference with Recursive Rules", *Proceedings of the 1st Annual National Conference on Artifical Intelligence*, AAAI, Palo Alto, Calif., 1980.

36. Stonebraker, M., "Operating System Support for Database Management", *Communication of the ACM*, Volume 24, Number 7, July, 1981, pp. 412–418.

37. Ullman, J. D., "Implementation of Logical Query Languages for Databases", *ACM Transactions on Database Systems*, Volume 10, Number 3, 1985.

38. Ullman, J. D., "Logic and Database System", *On Knowledge Base Management Systems - Integrating Databases and AI Systems*, M. Brodie and J. Mylopoulos, Eds., Springer–Verlag, Berlin and New York, 1986.

39. Yu, C. T., and Zhang, W., "Efficient Recursive Query Processing Using Wavefront Methods", *Proceedings of the 3rd International Conference on Data Engineering*, IEEE, February, 1987.

APPENDIX A

This appendix contains the pseudocode of the three algorithms: Henschen-Naqvi, Magic Set, and Counting. The pseudocode of each algorithm is written in terms of relational operations. These operations are *join*, *select*, *project*, *union*, and *diff*. The notation used to denote these operations are described as follows.

1. A * B -> join(A, B),
2. σA -> select(A),
3. πA -> project(A),
4. A + B -> union(A,B),
5. A - B -> diff(A,B).

where A and B are relations to be operated on. For each algorithm there is a particular set of intermediate relations which will be specified with the description of the algorithm. In general, a concatenation of two relation names, AB for example, denotes an intermediate relation which is the result of the join of the two relations A and B. A relation name with a superscript, R^i means the result of joining the relation R with itself $i-1$ times. The result relation of any of the operations is a relation. Therefore, it can be an operand of another operation. For example, $\pi(\sigma A)$ represents the select operation on the relation A followed by a project operation on the result. The parenthesis indicate the imposed precedence of the operations. The number of iterations in each algorithm is fixed at 5.

Notation:

1. W_i – the wavefront relation at iteration i. (i.e. σup^i)
2. R_i – an intermediate relation at iteration i obtained by joining the W_i relation with the *flat* relation. (i.e. $(\sigma up^i)flat$)
3. $T_{i,j}$ – an intermediate relation at iteration i obtained by joining the R_i relation with the *down* relation j times. (i.e. $\sigma up^i flat down^j$)
4. A – the relation contains the final answer to the query, it is initially an empty relation.

The algorithm

```
1)  Set $W_0 = \pi(\sigma up)$.
2)  For i=1, 2, ... ,5 do
     begin
3)        $R_i = \pi(W_{i-1} \bullet flat)$
4)        Set $T_{i,0} = R_i$
5)        For j= 1, 2, ... ,i do
          begin
6)            $T_{i,j} = \pi(T_{i,j-1} \bullet down)$
          end
7)        Set $A = A + T_{i,i}$
8)        Set $W_i = \pi(W_{i-1} \bullet up)$
     end.
```

Magic Set

Notation:

1. magic – the relation *magic set.*
2. M_i – the intermediate relation contains a partial magic set at iteration *i.*
3. magicup – the intermediate relation obtained by joining *magic* and *up.*
4. magicflat – the intermediate relation obtained by joining *magic* and *flat.*
5. R_i – the derived intermediate relation at iteration *i.*
6. A – the relation contains the final answer to the query, it is initially an empty relation.

The algorithm

```
1)  Set magic = $M_0$ = $\pi(\sigma up)$.
2)  For i = 2, 3, ... ,5 do
    begin
3)      $M_i = (M_{i-1} \bullet up) - M_{i-1}$.
4)      magic = magic + $M_i$.
    end.
5)  Set magicup = $\pi((magic \bullet up) + \sigma up)$.
6)  Set magicflat = $\pi(magic \bullet flat)$.
7)  Set $R_0$ = magicflat;
8)  For i = 1, 2, ... ,5 do
    begin
9)      $R_i = \pi(((magicup \bullet R_{i-1}) \bullet down) - R_{i-1})$
10)     A = A + $R_i$.
    end.
11) Set A = $\sigma$A.
```

Counting

Notation:

1. C_i – the ***counting set*** at iteration i.
2. F_i – the intermediate relation at iteration i obtained by joining C_i and *flat*. (i.e. C_i*flat*)
3. D_i – the intermediate relation at iteration i obtained by joining the relation D_{i+1} and the relation ***down*** then unioned with F_i (i.e. $(D_{i+1} down) + F_i$)
4. A – the relation contains the final answer to the query, it is initially an empty relation.

The algorithm

```
1)  Set $C_1 = \pi(\sigma up)$.
2)  For i = 2, 3, ... ,5 do
      begin
3)        $C_i = \pi(C_{i-1} \bullet up)$.
      end.
4)  For i = 5, 4, ... ,1 do
      begin
5)        $F_i = \pi(C_i * flat)$.
      end.
6)  Set $D_5 = F_5$.
7)  For i = 4, 3, 2, 1 do
      begin
8)        $D_i = \pi((D_{i+1} \bullet down) + F_i)$.
      end.
9)  Set $A = (D_1 * down)$.
```

APPENDIX B

This appendix derives the expected values of the *join selectivity* (JS) of two relations R and S, and the expected values of the *selection selectivity* (SS) of a relation R. Let us first consider the JS of R and S, and the join attributes are A and B of R and S respectively. Recall that JS is defined as the ratio of the size of the result relation after the join operation to the product of the size of the two original relations. The following assumptions are made for the derivation.

a The size of the relation R (S) is $|R|$ ($|S|$). (i.e. there is $|R|$ ($|S|$) tuples in the relation R (S).)

b The values of the attributes A and B of R and S respectively are chosen randomly from a domain D of positive integers and the size of the domain is $|D|$. (i.e. if D contains the integers 1, 2, ..., 1000, then its size is 1000.)

c The values of A (B) are uniformly distributed over the $|R|$ ($|S|$) tuples in R (S).

The derivation of the expected JS is as follows.

1. For each tuple r of R, the probability that the attribute A of r (denoted as r.A) is equal to a random value a from D (i.e. $r.A = a$) is $1/|D|$.
   Therefore, the number of tuples in R such that $r.A = a$ is $|R|/|D|$.
2. For a tuple s in S and $s.B = a$, the number of matching tuples in R (i.e. $r.A = s.B$) is equal to $|R|/|D|$.
   Therefore, the total number of tuples obtained from R and S such that $r.A = s.B$ is equal to $(|S|*|R|)/|D|$. In other words, the size of the result relation of joining R and S is $(|S|*|R|)/|D|$.
3. Based on (1) and (2), the expected JS is then equal to $((|S|*|R|)/|D|)/(|S|*|R|)$ which is simplified to $1/|D|$.
   Therefore, the expected JS of joining two relations which satisfy the assumptions (a), (b), and (c) is equal to the reciprocal of the domain from which the values of the two join attributes are chosen from.

For the three JS values used in this research, 0.001, 0.0005, and 0.0001, the size of the domain of the join attributes are 1000, 2000, and 10000, respectively.

The derivation of the SS, defined as the ratio of the size of the result relation after the selection to the size of the original relation, of a relation *R* is similar to the derivation of the JS as shown above. Using the same assumptions, the expected value of SS of a relation *R* can be derived as follows:

For each tuple *r* of *R*, the probability that the attribute *A* of *r* is ***less than or equal to*** a random value *a* from D is $a/|D|$. The number of tuples in R such that $r.A <= a$ is $a/|D|$. Therefore, the SS of the relation *R* on the attribute *A* is $((a*|R|)/|D|)/|R|$ which is simplified to $a/|D|$.

Therefore, the SS of a relation R is equal to the reciprocal of the size of the domain of the selection attribute multiplied by a selected value from the same domain. In other words, the SS of a relation can be controlled by choosing a domain of certain size and a particular value from this domain. For example, if the domain size is 1000, choosing the value of 100 yields a SS of 0.1. For this SS value, it means that all the tuples in *R* of which the value of attribute *A* is less than or equal to 100 will be selected.