The Design and Performance Study of Binary Transitive Closure Algorithms

by

Paul L. C. Wu

B.Sc, Simon Fraser University, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
December 1988

Approval

Name:	Paul L. C. Wu
Degree:	Master of Science
Title of Thesis:	The Design and Performance Study Of Binary Transitive Closure Algorithms

Examine Commitee:

Chairman: Dr. Binay K. Bhattacharya

Dr. Woshun Luk,
Senior Supervisor

Dr. JiaWei Han
Supervisory Committee Member

Dr. Tiko Kameda,
External Committee Member

July 15, 1988
Date of Approval

Title of Thesis/Project/Extended Essay

The Design and Performance Study of Binary Transitive Closure Algorithms.

Author:

(signature)

Loong Cheong Paul WU

(name)

December 14, 1988

(date)

Abstract

Transitive closure operation is one of the most useful new operations in deductive database systems. When it is added to conventional relational database systems, most practical problems with recursion can be coped with. Therefore, efficient processing of transitive closure is an important task in deductive database systems.

Transitive closure operation can be divided into **total closure** and **query closure** operations. To implement the operations, we can have **unary** or **binary** algorithms. A lot of work has been done on the efficient derivation of total closures and unary query closures, but not much work has been done on binary query closures. A binary query closure algorithm derives the transitive closure relevant to a set of query constants, associated with pairs of the initial query constants and their driven elements. The binary query closure operation is more frequently used than the unary one.

In this research, 6 algorithms are developed for binary query transitive closure processing, namely Binary Wavefront, Unary Wavefront with Frontier-Edges, Unary Wavefront with Implied-Edges-Closure, Unary Wavefront Preprocessing with Total-Frontier-Edges, level-relaxed Binary Wavefront, and level-relaxed Unary Wavefront Preprocessing with Total-Frontier-Edges. These algorithms are analyzed and their relative performances are compared on their I/O behavior and other processing costs. More importantly, the analysis is done on different characteristics of data and on different buffer sizes.

Our analysis and performance study show that reference locality and data clustering play an important role in the performance of the algorithms. The ordering of the set of relational operations is also important in determining the I/O performance of the algorithms. Our research also demonstrates that the rate at which the disk I/O of an algorithm decreases

with the increase in buffer size is affected by the Maximum Buffer Requirement of the algorithm. Among the algorithms without level relaxation, Binary Wavefront outperforms the others in a wide range of data sets when the buffer size is small. Unary Wavefront Preprocessing with Total-Frontier-Edges Algorithm performs better when the number of query constants is small and the buffer size is large. A base relation that requires a lot of iterative processing and generates a large volume of answers to the transitive query is best processed by Unary Wavefront with Implied-Edges-Closure Algorithm. The analysis of the I/O behavior of Unary Wavefront with Frontier-Edges Algorithm helps to develop a better algorithm, like the Unary Wavefront Preprocessing with Total-Frontier-Edges Algorithm. The level relaxed versions of the algorithms are best for clustered data.

Our research provides insight into binary query closure processing. We hope that this will, in turn, stimulate further research on the processing of more complex recursions.

Acknowledgements

I owe a great debt of gratitude to Dr. Woshun Luk and Dr. Jiawei Han who introduced me into this area. Their support, both financial and academic, and their thoughtful guidance are deeply appreciated. I would also like to thank Dr. Tiko Kameda for his constructive comments and suggestions, and for helping with the revision of the thesis.

Thanks also to Ada Fu and T. Pattabhiraman for their long hours devoted to editing this thesis. I am very grateful to Frank Tong, Mimi Kao and Steven Yap for their useful ideas and implementation techiques. I owe thanks to the faculty, staff, and graduate students of the School of Computing Science for providing me with an enjoyable environment to work in. My sincere thanks also to Carolina Chan and Ponch Poon for their general support.

Finally, I thank my very dear grandparents and parents who have unfailingly supported my studies. I wish to dedicate this work to them.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Deductive database (DDB) is an important research area to integrate relational database systems and logic programming. Relational database systems are inadequate in handling recursion and making inferences. On the other hand, logic programming languages like Prolog are inefficient in large database management. The general database system features like integrity constraints, concurrency control and recovery are not well-addressed by Prolog. Deductive database systems incorporate both relational operations and inference rules. As pointed out in [18], the objective of DDB is to enrich the query language of the relational database system, and to handle missing or unknown data.

Transitive closure operation is the most important operation in DDBs (eg. [7], [9], [11] and [12]). Most other recursions may include transitive closure computation as an operation or suboperation[1]. Therefore, most practical applications involving recursion will need the evaluation of transitive closure. Hence, efficient processing of transitive closure is an important issue in DDB research.

There are two types of transitive closures: **Total Closures** and **Query Closures**. To find the total closure of a binary base relation, one needs to compute the transitive closure of all the attribute values in the base relation. On the other hand, query closure computation requires one to find the transitive closure of some attribute values of the base relation only. A transitive closure algorithm can be either **unary** or **binary**. The output of a unary algorithm is a unary relation, while that of a binary one is a binary relation. The definitions of these terms will be treated more formally in section 1.2. The problem of computing

[1] In [7], the relationship of various complicated recursive clusters to transitive closure is discussed. This is beyond the scope of this research; interested readers should refer to [7].

total closure (eg. [1], [9], [19], [20] and [21]) has received much more attention than the query closure (eg. [7], [10], [16] and [8]). We feel that it is more important to study the query closure computation of transitive closures. This is because query closure enables us to find only those portions of interest of the closure in the database. In practice, we are seldom interested in the whole portion in the database, and hence, query closure computation will be more frequently encountered in the real world. For example, it is more common to ask questions like "Find the ancestors of the individuals in Group A of the database", which is a query closure question, than questions like "Find the ancestors of all the individuals in the database", which is a total closure query question. The study of query closure is thus more practically relevant. Moreover, we also feel that it is more important to study binary algorithms than unary ones. In the example above, it is more likely that we want to find out the ancestors for each individual, and attach this relationship to that particular individual. In other words, it is more interesting to know the ancestor-descendant pairs (binary) than just a list of ancestors or a list of descendants (unary).

To our best knowledge, there has not been any in-depth study of binary query closure algorithms. In this thesis, we will introduce several algorithms for computing binary query closure. The efficiency of the algorithms is measured in terms of its disk I/O. The disk I/O of these algorithms is measured in simulation against both randomly generated and specially constructed databases, using various I/O buffer sizes. This performance study will provide insight into the algorithms; not only does it identify the strengths and weaknesses of these algorithms, but it also suggests ways to produce improved algorithms.

In Chapter 2, we will introduce and present some algorithms used to compute binary query closure. The correctness and the termination problem of the algorithms will also be discussed. In Chapter 3, some typical data sets are given in order to show the strengths and weaknesses of the algorithms presented in Chapter 2. Then in Chapter 4, our database model will be described. The analytical tools will be presented in Chapter 5. The general performance of the algorithms on randomly generated data is studied and analyzed in Chapter 6. In Chapter 7, a further refinement of the algorithms, the level-relaxation on the algorithms, is proposed, and the new versions are presented and discussed. Finally, in Chapter 8, we will draw our conclusions, and discuss issues for further research.

1.1. Deductive Database

The goals of research into DDB systems are:

- to enhance relational database system with the use of deductive logic, and
- to increase the expressive power of relational database system to cope with more complicated queries.

As defined in [4], a DDB has three major components:

1. elementary facts.
2. deductive rules.
3. integrity constraints.

The elementary facts are the sets of relations which are composed of tuples stored in the database. These sets of tuples form the **extensional database (EDB)**, and the relations are known as base relations. Deductive rules and integrity constraints are collectively known as the **intensional database (IDB)**. In the extreme sense, integrity constraints can be considered as deductive rules. In this research, this view will be adopted, and IDB will be considered to consist of merely deductive rules.

DDB can be viewed as a theorem proving system, [4], in which queries are represented as theorems. Query processing is then considered to be a theorem proving activity. Recursive query processing in DDB is the resolution of query using not only the explicit information stored in EDB, but also implicit facts that can be derived by using the rules in IDB. Thus, the evaluation of recursive queries requires iterative access to the information in the EDB as well as the deductive rules in the IDB.

In current DDB literature, there are two common approaches to resolving recursive queries. They are the **interpretive approach** and **compilation approach**. Using the interpretive approach, the theorem prover of DDB interleaves to both the IDB and EDB. The merit of this approach is that it can dynamically retrieve relevant facts towards the answer at run time. However, this approach is very expensive when accessing large databases because of the nature of interleaving EDB and IDB. On the other hand, the compilation approach requires the theorem prover to preprocess the deductive rules into a set of relational operations, and then search for facts in the EDB. Thus it is easier for this approach to obtain global optimization of database accesses by using the techniques developed in relational database systems. For the advantages and disadvantages of these approaches, interested readers are referred to more formal studies in [2], [14] and [17]. In this research, we will consider recursive query processing using the compilation approach.

1.2. Transitive Closure

Transitive closure is the simplest kind of recursive query processing. Formally, the transitive closure of a binary base relation *A* is defined by the following Horn clauses expressed in notation very similar to Prolog, but we use capital letters to denote predicate head, small letters to denote variables:

R(x,y) :- A(x,y).
R(x,y) :- R(x,z), A(z,y).

The first implication rule is often referred to as the exit rule for the predicate R. The second rule is the recursive rule for predicate R. The symbol ":-" separates the conclusion (on the left of the symbol) and the antecedent (on the right of the symbol) of the implication rule, and the symbol "," is "logical and". Thus, the recursive rule reads like "R(x,y) is true when R(x,z) and A(z,y) are both true". After compilation, a sequence of expanded formulas will be formed:

$$
\begin{aligned}
&A^1(x,y) = A(x,y)\\
&A^2(x,y) = A(x,y_1), A(y_1,y)\\
&A^3(x,y) = A(x,y_1), A(y_1,y_2), A(y_2,y)\\
&\quad\cdots\\
&A^i(x,y) = A(x,y_1), \cdots, A(y_{i-1},y)\\
&\quad\cdots\\
&A^n(x,y) = A(x,y_1), \cdots, A(y_{n-1},y)
\end{aligned}
$$

The number i in $A^i(x,y)$ denotes the **chain distance** between x and y. The compiled formula for transitive closure is then defined as

$$A^{+}(x,y) = \bigcup_{i=1}^{n} A^{i}(x,y),$$

where n is the smallest m such that

$$\bigcup_{i=1}^{m} A^{i}(x,y) = \bigcup_{i=1}^{m+1} A^{i}(x,y).$$

We call the number n the **longest chain distance** between x and y.

In the theorem-prover view of DDB, the truth of $A^{i}(x,y)$ implies that there exist constants $y_1, y_2, \cdots, y_{i-1}$ such that $A(x,y_1)$, $A(y_1,y_2)$, and so on to $A(y_{i-1},y)$ are true. $A^{+}(x,y)$ is true, if $A^{i}(x,y)$ is true for some i, $1 \leq i \leq n$. On the other hand, in a relational database system, the expanded formulae are viewed as a series of join operations. Besides, we will determine whether the set $A^{i}(x,y)$ is empty or not by searching for the tuple (x,y) in the result of joining the relation *A* with itself i-1 times. In the rest of this section, we will adopt the theorem-prover view of DDB to define terms, but the readers should be aware that these terms can be defined equally well by using the relational database terminology.

A transitive closure can be either a total closure or a query closure. We define two different closures using the above Horn clauses rules as follows:

Definition 1-1: Total Closure of A is the set { (x,y): $A^{+}(x,y)$ is true }.

Definition 1-2: Query Closure of *C* in A is the set { (c,y): $c \in C$ and $A^{+}(c,y)$ is true }, where *C* is a set of constants, called the **query constants.**

The **depth of total closure** of *A* is the maximum over all x and y of the set

{ n_{xy} : n_{xy} is the longest chain distance between x and y; $A^{+}(x,y)$ is true }

The **depth of query closure** of *C* in *A* is defined in a similar fashion. That is, it is the maximum over all c and y of the set

{ n_{cy} : n_{cy} is the longest chain distance between c and y; $A^{+}(c,y)$ is true and $c \in C$}

When it is clear whether a query closure or a total closure evaluation is needed, we will use the term **depth of transitive closure** instead of either "the depth of query closure" or "the depth of total closure".

A base relation may be represented as a digraph, in which each node in the graph represents an attribute value, and each arc represents a tuple with the attribute values on its two ends. In the rest of this thesis, we use attributes and nodes, tuples and edges, and, a relation and a digraph interchangeably. For example, for a base relation A with the tuples:

$\{(a,c), (b,d), (a,d), (c,e), (d,f), (d,g)\}$

the digraph will be the one shown in Figure 1-1. The set of initial query constants will be some nodes in the digraph. This set of nodes which corresponds to the set of query constants is referred as **starting nodes** in the graphical representation.

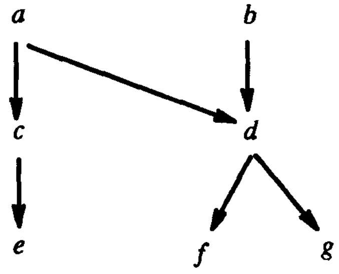


Figure 1-1: The digraph of relation A

The total closure of a digraph is defined as the set of all possible node pairs where the first element is any node in the digraph and the second element is a node reachable from the first. The query closure of some given nodes in a given digraph is the set of all possible node pairs where the first element is a starting node, and the second element is a node reachable from the first. For example, in the above base relation, assuming the starting nodes to be a and b, we have:

total closure: $\{(a,c),(a,d),(a,e),(a,f),(a,g),(b,d),(b,f),(b,g),(c,e),(d,f),(d,g)\}$
query closure: $\{(a,c),(a,d),(a,e),(a,f),(a,g),(b,d),(b,f),(b,g)\}$

When computing transitive closure, some tuples which are not in the set of the original edges of the digraph will be added to the final closure (total or query closure). A tuple, say (x,z), is produced when we have edges (x,y_1), (y_1,y_2), $\cdots$, (y_k,z). We call the tuple (x,z) **implied edge**. Take the above example, (a,e) is an implied edge which is formed because of the existence of the edges (a,c) and (c,e).

In the process of computing transitive closure, if we compute using a relation A(x,y), starting from a set of x values to find a set of y values, we call the set of x **drivers**, and the set of y **driven elements**. For the above example, driver *a* will lead to driven element *c* because we have the tuple *(a,c).*

As stated before, a query transitive closure algorithm can be classified as **unary** or **binary**. Their formal definitions are given as follows:

Definition 1-3: The unary query closure of a set of initial query constants with respect to a relation is the set of driven elements that are derived from the initial drivers[2]. (The unary query closure can also be defined as the set of all the nodes that are accessible from the set of starting nodes.)

Definition 1-4: A unary query closure algorithm is an algorithm that computes unary query closures.

Definition 1-5: The binary query closure of a set of initial query constants with respect to a relation is the set of tuples, associating the driven elements with their corresponding initial drivers. (The binary query closure consists of node pairs of the starting nodes and the nodes accessible from them.)

$$Cl(C,\mathrm{A}) = \{ (c,y) : c \in C \text{ and } \mathrm{A}^i(c,y) \text{ is true for ssome positive integer } i \leq n \}$$

where C is the set of the initial drivers.

Definition 1-6: A binary query closure algorithm is one that computes binary query closures.

If we want to find the descendants of each individual within a group, a binary query closure algorithm is needed. This is because we have to keep information on the kinship of the individual and his/her descendants. However, if we just want to know who are the descendants of a group of individuals, all we need is a unary algorithm, because the information on the relationship of two individuals is not needed. It should be noted that applying a unary algorithm to total closure evaluation will generate the set of all the nodes with in-degree greater than 0. Take the base relation *A* in Figure 1-1 as an example. If we

[2] The term "initial query constants" is used interchangeably with the term "initial drivers" in this thesis.

use a unary algorithm to find the query closure for the starting nodes *a* and *b* then all we will get is the following set of nodes:

$N = \{ c, d, e, f, g \}$.

In contrast, if a binary algorithm is used, we will get

$Cl(\{a,b\},\mathrm{A}) = \{(a,c),(a,d),(a,e),(a,f),(a,g),(b,d),(b,f),(b,g)\}$.

1.3. Unary Query Closure Algorithms

In the current literature, δ-Wavefront is a popular unary algorithm for the evaluation of the transitive closure. Its efficiency can be further improved by an algorithm called level-relaxed δ-Wavefront in [6]. δ-Wavefront uses two intermediate sets: the frontier *F* to store the "active" drivers, and closure *Cl* to collect the results. These sets are initialized to the set of query constants *C*. At each iteration, a new driver set *F* (frontier) will be derived by joining the drivers generated from the last iteration to the base relation *A*. The occurrence of each of the new drivers in *Cl* will be checked. Those drivers which exist in *Cl* are discarded from the driver set. The remaining drivers will then be added to *Cl*. The loop terminates when no more drivers are available.

The δ-Wavefront algorithm is as follows:

```
Cl := C
F := C
while ( $F \neq \emptyset$ )
    F := F Δ A - Cl
    Cl := Cl $\cup_c$ F
end while
```

Δ is the join operator joining a unary relation with a binary relation. Thus, the result of the join operation is the set of y values in *A*(x,y), where (x,y) is a tuple of *A*, and x is an attribute value in *F*. Unlike the standard union operation, $\cup_c$ concatenates its two operands without removing any duplicates. The result of $\cup_c$ is a unary relation which is a set of driven elements from the drivers in *F*. Readers interested in the proof of correctness and termination of the algorithm are referred to [5].

Chapter 2
Binary Query Closure Algorithms

In Section 1.3, the δ-Wavefront algorithm was described. This unary algorithm is very efficient in computing the query closure of a relation. However, δ-Wavefront does not keep track of the starting nodes from which each of the elements in the final closure is derived. To find such relationships using the unary algorithm, one needs to separately apply the algorithm to each of the starting nodes, instead of applying to the set of query constants (starting nodes) collectively. Nevertheless, this strategy is not efficient in terms of disk I/O. Several drivers from several starting nodes may be in the same data page of the base relation. By using the above strategy, the number of times the data page where the drivers reside is accessed at least equals the number of the drivers. Hence, what we need is some strategies that enable us to compute the transitive closures of the starting nodes simultaneously, so that the number of data pages accesses is reduced. Such global optimization is possible also with binary algorithms. In the following sections, we will present some binary algorithms for query closure evaluation, and we will also prove their correctness and termination.

2.1. Binary Wavefront (BWFT)

Binary Wavefront is the simplest and most straightforward algorithm derived from the unary δ-Wavefront algorithm. Being a binary algorithm, BWFT determines the pairs of initial drivers and their driven elements. The major characteristic of BWFT which distinguishes it from the δ-Wavefront algorithm is the use of binary relation *F* to store the driver-driven elements pairs, and binary relation *Cl* to accumulate the final answers. We call each tuple (*x*,*y*) in *F* a **binary driver**, *y* an **active driver**, and *F* the **binary frontier**. In BWFT, both *F* and *Cl* are initialized to the set of tuples resulting from the selection of *A* on the query constants. As in the δ-Wavefront algorithm, at each iteration of BWFT, new binary drivers will be derived. These newly generated binary drivers will be checked against *Cl* for occurrence. Existing binary drivers will then be deleted from the set of binary drivers and the closure *Cl* will be updated by adding to it the remaining drivers. The basic algorithm for BWFT is presented as follows:

```
Cl := $\sigma A$
F := Cl
while ( $F \neq \emptyset$ )
   F := $F \Delta A - Cl$
   Cl := $Cl \cup_c F$
end while
```

σ represents the selection operator. The first step of the algorithm is to select all the tuples in the relation *A* with the query constants on its first attribute column. These tuples are assigned to the binary relation *Cl.* The final closure will be stored in this relation. Δ here represents the join operation on two binary relations *F* and *A*. The joining attributes are the second attribute in *F* and the first attribute in *A*. The result of the join operation will consist of the first attribute of *F* and the second attribute of *A*. The generated tuples are stored in some temporary relation. $\cup_c$ concatenates its two operands without duplication removal. The following theorem shows that BWFT is an effective and correct algorithm.

Theorem 2-1: BWFT terminates and generates the correct query closure in *Cl.*

Sketched Proof :

Inside the loop, the new frontier (*F*) is found. The new frontier is the difference of the result of joining the previous frontier with the base relation *A*, and the closure *Cl.* The size of *Cl* can never exceed n^2 (n = size of the base relation *A*) because the size of total closure is bounded by n^2 and query closure is a subset of total closure. As the size of *Cl* grows iteratively, the new frontier will become empty eventually. Hence, the loop must terminate.

The correctness of BWFT follows directly from the δ-Wavefront algorithm. Initially, the frontier equals the closure *Cl* which is the result of selection of *A* on the query constants. Hence we get the paths of length 1 from the starting nodes to their immediate descendants. These descendant nodes are associated with their initial drivers in tuples which are stored in *Cl.* Then inside the loop, the nodes which are at distance 2, 3, ..., and so on to the longest path length from the set of starting nodes, will be found. Cycles are avoided by performing the difference operation with *Cl.* Therefore, in the graphical representation, each arc that is accessible from a given starting node will be visited once and only once during the derivation of the closure for the given starting node. □

2.1.1. Example of Using BWFT

Consider the digraph in Figure 2-1 with nodes *a* and *b* as the starting nodes. The closure *Cl* and frontier *F* are initialized to be the tuples corresponding to the edges emanating from nodes *a* and *b*. That is, we have (*a*,*c*) and (*b*,*c*) in *Cl* and *F* initially. At the first iteration, joining the frontier *F* with the base relation *A* will get the tuples (implied edges) (*a*,*d*) and (*b*,*d*), which do not yet belong to *Cl*. These tuples are then added to *Cl*. So *Cl* now contains *(a,c), (b,c), (a,d)* and *(b,d)*. In the second iteration, new frontier *F* with the tuples (*a*,*e*) and (*b*,*e*) will be produced. These tuples are then used in updating *Cl*. As there is no more arc emanating from node *e*, the new frontier formed in the third iteration is empty. Hence, no updating is needed for the closure *Cl*, and the loop terminates. The computation by BWFT is also shown in Figure 2-2.

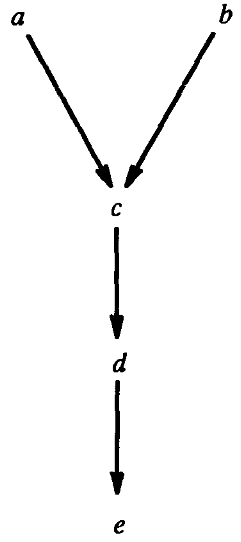


Figure 2-1: The digraph of a simple base relation

Iteration	Old-Frontier	New-Frontier	Closure
1	*(a,c),(b,c)*	*(a,d),(b,d)*	*(a,c),(b,c),(a,d),(b,d)*
2	*(a,d),(b,d)*	*(a,e),(b,e)*	*(a,c),(b,c),(a,d),(b,d),(a,e),(b,e)*
3	*(a,e),(b,e)*		*(a,c),(b,c),(a,d),(b,d),(a,e),(b,e)*

Figure 2-2: Derivation process of BWFT

2.1.2. Discussion on BWFT

If we take a closer look at BWFT algorithm, we will notice certain redundancies. Let us re-consider the graph in Figure 2-1. The drivers *c*, *d* and *e* for each of the iterations are common to the nodes *a* and *b*. BWFT ignores the fact that nodes *a* and *b* have a common descendant *c*. Therefore in each iteration, BWFT must consider two tuples. However, if we operate on only their common descendant *c*, then for each iteration, we need to consider one tuple. After that, we just attach the nodes reachable from *c* to nodes *a* and *b*.

Consider another more illustrative example of redundant processing inherent in BWFT. In Figure 2-3, nodes a_1 and a_2 are the starting nodes. It can be shown that in 3 iterations, BWFT can produce the implied edge (a_2,b_1) into *Cl* (the relation to store the final closure). Then BWFT needs two more iterations to get (a_2,d_1) into *Cl*. In total, BWFT requires 5 iterations to compute the query closure. However, the process from the third iteration to the fifth iteration is actually redundant. This is because the edges from b_1 to c_1 and from c_1 to d_1 have been accessed during the computation of the transitive closure for node a_1 in the first and second iterations. Therefore, if the information that b_1 can lead to c_1 and d_1 was kept in the first and second iterations, when b_1 is discovered to be a descendant of a_2, we can know that c_1 and d_1 are also descendants of a_2. Thus, in the third iteration, we should be able to include the tuples (a_2,c_1) and (a_2,d_1) along with (a_2,b_1) into *Cl*.

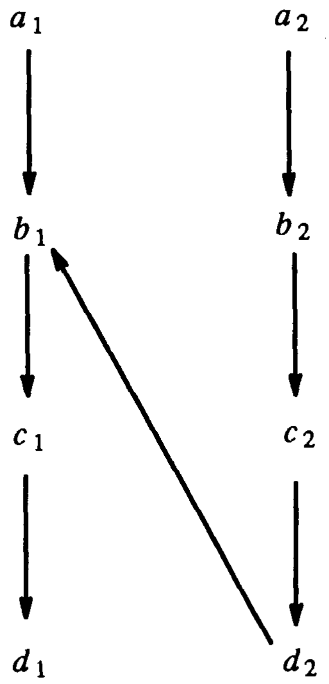


Figure 2-3: BWFT will retraverse the path from b_1 to d_1

There are two observations from the above examples that will serve as guidelines for the design of alternative binary query closure algorithms. First, we can use a unary frontier rather than a binary one. Initially, the frontier consists of the starting nodes. At each iteration, the frontier is joined with the base relation to produce new nodes which are reachable from the starting nodes. With duplicates removed, these unique new nodes will form the frontier for the next iteration of processing, until no more new nodes are produced. Thus, at the beginning of the i-th iteration, the frontier will consist of all the nodes which are at distance i from at least one of the starting nodes. We call this set of nodes **i-th-frontier**, i≥0 (when i=0, it is the set of starting nodes). For example, 0th-frontier in the example shown in Figure 2-3 will contain a_1 and a_2, 1st-frontier will contain b_1 and b_2, 2nd-frontier will contain c_1 and c_2 and 3rd-frontier will contain d_1 and d_2.

The second observation concerns the processing during each iteration. The processing can produce new edges (tuples) that will be part of the final answer (the query closure), or it can produce nothing of that sort, in which case the algorithm is only a preprocessing algorithm. We will discuss the former case first. We define **i-th-edges** to be the set of direct edges leading from the nodes in i-th-frontier and **i-frontier-edges** to be the union of all j-th-edges, $0 \leq j \leq i$. We also define the *i-closure* to be the query closure which is derivable from i-frontier-edges with the same starting nodes. When the algorithm terminates, the i-closure will be the final answer. This is because at the end of the algorithm, say at the m-th iteration, all the nodes accessible from the starting nodes must be included in k-th-frontier, where $0 \leq k \leq m$. Thus, k-frontier-edges must be all the edges with which the starting nodes can access their descendants. Hence, the closure derived from these k-frontier-edges must be the desired query closure.

There are two ways to compute the i-closure. The i-closure at each iteration can be obtained using the BWFT method. Alternatively, the total closure of i-frontier-edges can be computed, which is called the **i-implied-edges-closure** here. At the end, say after the n-th iteration, the query closure can be computed by performing a selection on the n-implied-edges-closure. We call the n-implied-edges-closures the **total-implied-edges-closure**, and it is the total closure of the set n-frontier-edges which is defined as **total-frontier-edges**. The chief benefit of finding the i-implied-edges-closure in each iteration is that it requires fewer iterations of computation than that of the i-closure computation because of more implied edges being stored. Consider the example in Fig. 2-3 again. If the 3-closure is computed, the implied edge (a_2,d_1) will be established (deduced) in 2 iterations because (a_2,c_1) has to be first established. However, if we have in the i-implied-edges-closure the edge (b_1,d_1), then the implied edge (a_2,d_1) can be established in one iteration. Of course, the total closure computation is more intensive and worst of all, the i-implied-edges-closure can be huge in size.

As a preprocessing strategy, one can make use of a unary frontier algorithm to identify all edges (tuples) that may be needed to compute the query closure. In doing so, all redundant tuples in the base relation may be removed before applying a standard query closure algorithm, i.e., BWFT.

In the next few sections, we shall describe three different unary frontier algorithms based on the above discussions. They are: the Unary Wavefront with Implied-Edges-Closure algorithm, which computes the i-implied-edges-closure at each iteration, the Unary Wavefront with Frontier-Edges algorithm, which computes the i-closure at each iteration, and the Unary Wavefront Preprocessing with Total-Frontier-Edges algorithm which is a preprocessing algorithm.

2.2. Unary Wavefront with Implied-Edges-Closure (UIEC)

In the previous section, we have discussed the use of unary frontier in each iteration. In this section, we will present and study an algorithm, Unary Wavefront with Implied-Edges-Closure (UIEC). UIEC will compute the i-implied-edges-closure at each iteration. In fact, the i-implied-edges-closure provides a complete information about the derivation path of each of the drivers encountered so far. In particular, there are (implied) edges associating a node with each of its descendants found in the i-implied-edges-closure. Thus, when a driver *a* reaches a node *b*, the descendants of *b* can be simply passed to *a* instead of deriving from scratch.

In our implementation, we have a relation *SubCl* to store the i-implied-edges-closure. The major objective in each iteration of UIEC is to find the i-implied-edges-closure which is the total closure of i-frontier-edges. Direct application of total closure algorithm to i-frontier-edges is costly, so we attempt to reduce the cost by making use of the (i-1)-implied-edges-closure from the last iteration.

Each iteration m begins by first exploring the nodes in m-th-frontier. This results in finding new edges, which are stored in *newSub*, different from (m-1)-frontier-edges. This set of newly discovered edges is actually equivalent to m-th-edges. From the nodes in m-th-frontier, the edges in *newSub* can lead to driven elements which fall into one of the three categories: the set of current driver (m-th-frontier) c_i, the set of unexplored drivers ((m+1)-th-frontier) d_j, and the set of previously explored drivers e_k. This can be visualized as in Fig. 2-4. If we can find all the implied edges leading from the set c_i to all the visited nodes

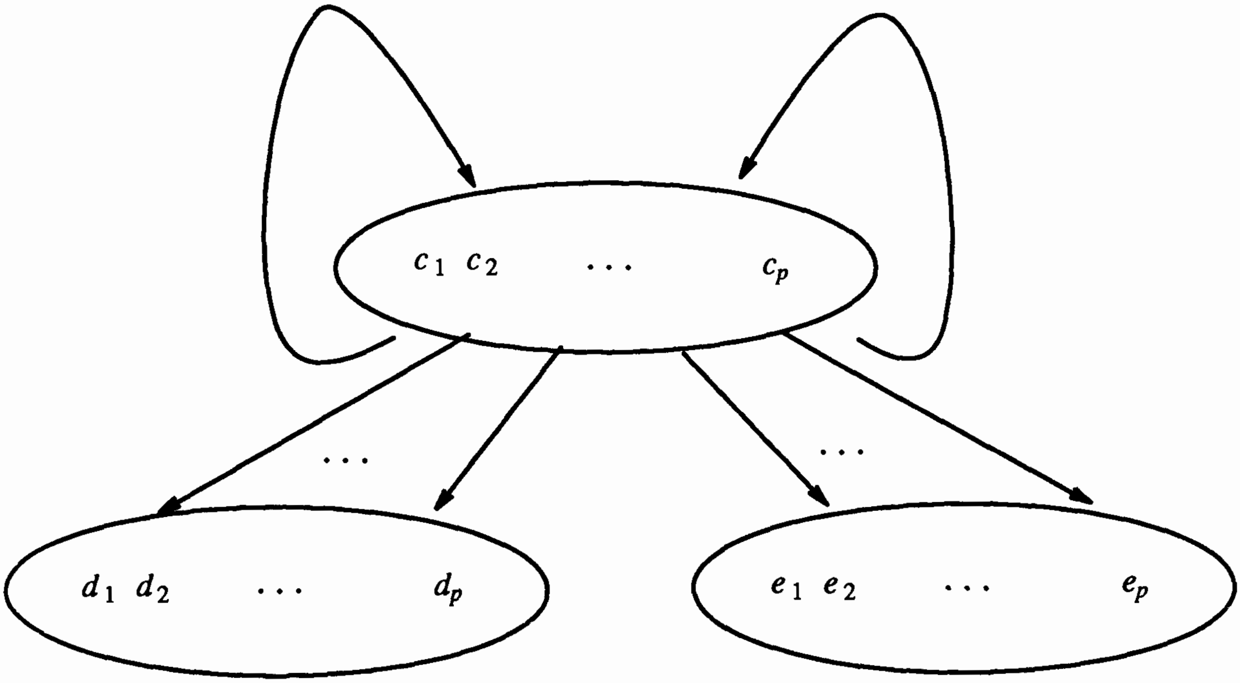


Figure 2-4: Digraph with edges incident from current drivers

(i.e., to the nodes in the union of l-th-frontier's, where $0 \leq l \leq m+1$), the job that remains for finding the m-implied-edges-closure is to compute the implied edges leading from the nodes in l-th-frontier's, where $0 \leq l \leq m-1$, to all the visited nodes using nodes in m-th-frontier as intermediates nodes. Now, let us look at the pseudo-code of UIEC:

```
SubCl := $\sigma$A                    /* 0-frontier-edges */
F := $\Pi_2$(SubCl) - $\Pi_1$(SubCl)   /* 1st-frontier */


    /* Loop 1 :  Compute 0-frontier-edges-closure */
newSub := SubCl $\Delta$ SubCl - SubCl
while ( newSub $\neq \emptyset$ )
   SubCl := SubCl $\cup_c$ newSub
   newSub := newSub $\Delta$ SubCl - SubCl
end while

    /* Loop 2 */
while ( F $\neq \emptyset$ )
   newSub := F $\Delta$ A
   if ( newSub = $\emptyset$ ) then exit the loop
   newSub2 := newSub $\Delta$ SubCl - newSub
   newSub := newSub $\cup_c$ newSub2

    /* Loop 3 :  Compute the implied edges leading from the */
    /*           current drivers to the visited nodes.      */
   newSub2 := newSub $\Delta$ newSub - newSub
   while ( newSub2 $\neq \emptyset$ )
       newSub := newSub $\cup_c$ newSub2
       newSub2 := newSub2 $\Delta$ newSub - newSub
   end while

   newSub3 := SubCl $\Delta$ newSub - SubCl /*find implied edges from  */
                                      /*the starting nodes to all*/
                                      /*visited nodes using the  */
                                      /*current drivers as       */
                                      /*intermediate nodes.      */
   SubCl := SubCl $\cup_c$ newSub      /* compute the i-implied- */
   SubCl := SubCl $\cup_c$ newSub3     /* edges-closure.         */

   F := $\Pi_2$(newSub) - $\Pi_1$(SubCl)   /* (i+1)th-frontier  */
end while


Cl := $\sigma$SubCl
```

σ as before represents the selection of the base relation on the query constants. Two selections are done in this algorithm. The first one chooses the tuples with the starting nodes in the first attribute of the base relation *A*, and the second one chooses from the relation *SubCl*. Π_i(*rel*) is the operator to do projection on the i-th attribute of relation *rel*. In the second line of the code, the difference from the set of the first attribute and the set of second attributes of *SubCl* is computed. The result is the unary relation *F*. This step ensures that the frontier (driver set) *F* contains drivers never processed before. Inside loop 1, the operands of the join operator Δ are both binary (*SubCl* and *newSub* are binary), like those in BWFT. The joining attributes are the second attribute of the left operand and first attribute the right operand. The result of the join operation is a binary relation whose first attribute is the first attribute of the left operand, and second attribute is the second attribute of the right operand. The join operator functions differently in loop 2, where *F* is unary but *A* is binary. The joining attributes are from *F* and the first attribute of *A*. The result is binary, and its attributes come from the attributes of *A*. Join operation in loop 3 is the same as in loop 1.

Lemma 2-2: *SubCl* correctly stores the i-implied-edges-closure at the end of the i-th iteration.

Sketched Proof :

We must note that loop 1 and loop 3 perform similar function. In fact, it can be proved that the total closure of the relations *SubCl* and *newSub* will be found after the termination of loop 1 and loop 3, respectively.

Initially, the drivers are all the starting nodes, i.e., 0th-frontier. After the first selection, *SubCl* will contain all the edges from the starting nodes to their direct descendants. Thus, *SubCl* is equivalent to 0-frontier-edges. Since loop 1 compute the total closure of *SubCl*, therefore, after the completion of loop 1, the 0-implied-edges-closure must be stored in *SubCl.*

At the beginning of each iteration i of loop 2 of UIEC, we assume that the lemma holds. That is to say, *SubCl* contains the (i-1)-implied-edges-closure. Finding i-implied-edges using the (i-1)-implied-edges-closure, we need to compute: i) all the implied edges leading from the nodes in i-th-frontier; ii) the implied edges using nodes in i-th-frontier as intermediate nodes.

Joining the frontier *F* to the base relation *A*, we will get all the edges from the current drivers to their descendants in *newSub*, and hence, we get i-th-edges. Performing join operation from *newSub* to *SubCl* will result in finding the set of implied edges

$$\{(c_j,y) : \text{there is an arc from } c_j \text{ to } e_k;\ c_j \in \text{i-th-frontier},\ e_k \in l\text{th-frontier}, \text{ where } 0 \leq l \leq \text{i-1, and } (e_k,y) \in SubCl\}.$$

The difference operation performed after the join operation is mainly to remove duplicates. The implied edges found are unioned to *newSub*. It should be noted that *newSub* now contains i-th-edges and some implied edges from the nodes in i-th-frontier to all the visited nodes. By using loop 3, the total closure of *newSub* will be found. Hence, all the implied edges leading from the current drivers to all the visited nodes can be found.

After the completion of loop 3, *SubCl* is joined to *newSub*. Thus, the implied edges leading from the nodes in *l*th-frontier, where $0 \le l \le i-1$, to i-th-frontier and to all the visited nodes are found. Then, the join operation is followed by two union operations, so that all the implied edges found are included in *SubCl*. Hence, *SubCl* will contain the i-implied-edges-closure at the end of loop 2.

Furthermore, the lemma is valid after loop 1, therefore, by induction on the iteration i, the lemma must hold. □

Theorem 2-3: UIEC terminates and correctly computes the query closure in *Cl*.

Sketched Proof :

There are n tuples in *A*, and hence, the n-th-frontier set is empty. Thus, in at most n iterations, all the nodes in i-th-frontier ($0 \le i \le n$) must have been explored. Therefore, loop 2 must terminate.

Since at the termination of loop 2, all the drivers have been considered, and hence their driven elements must also been considered. Therefore, all the descendants from the starting nodes must be considered, too. By Lemma 2-2, *SubCl* must contain the i-implied-edges-closure which is a superset of the i-closure. Thus, at the completion of loop 2, query closure will be a subset of the i-implied-edges-closure. Therefore, by performing a selection of *SubCl* on the initial nodes, we can get the query closure for the starting nodes. Hence, upon termination of UIEC, *Cl* contains the query closure. □

2.2.1. Example of Using UIEC

Let us consider Figure 2-1. The derivation process of UIEC is shown in Figure 2-5. *SubCl* is initialized to {*(a,c),(b,c)*}, and *F* contains node *c* only. Since nodes *a* and *b* are not directly linked to each other, loop 1 will not be entered. At the first iteration of loop 2, *newSub* will contain tuple (*c,d*). Node *d* will be the only element in i-th-frontier for the next iteration. At the time when loop 3 is being entered, *newSub* contains the tuple (*c,d*). As node *d* has not been explored, loop 3 will not be entered. Now, *newSub3* will contain tuples *(a,d), (b,d)*. Then updating *SubCl* will set it to { *(a,c), (b,c), (a,d), (b,d), (b,c)* }. Similarly, at the second iteration of loop 2, (*d,e*) will be derived and stored in *newSub*. Loop 3 will not be entered. After the update, tuples *(a,e), (b,e)* and (*d,e*) will be added to *SubCl*. Since node *e* has no outgoing arc, *SubCl* need not be updated, and loop 2 terminates. The last step is to perform selection on *SubCl* and to take the tuples starting with *a* and *b* into *Cl*.

Iteration	Old *F*	New *F*	Old *SubCl*	*SubCl*	*Cl*
1	*c*	*d*	(*a*,*c*),(*b*,*c*)	(*a*,*c*),(*b*,*c*),(*a*,*d*),(*b*,*d*),(*c*,*d*)	
2	*d*	*e*	(*a*,*c*),(*b*,*c*),(*a*,*d*),(*b*,*d*),(*c*,*d*)	(*a*,*c*),(*b*,*c*),(*a*,*d*),(*b*,*d*),(*c*,*d*), (*a*,*e*),(*b*,*e*),(*c*,*e*),(*d*,*e*)	
3	*e*		(*a*,*c*),(*b*,*c*),(*a*,*d*),(*b*,*d*),(*c*,*d*), (*a*,*e*),(*b*,*e*),(*c*,*e*),(*d*,*e*)	(*a*,*c*),(*b*,*c*),(*a*,*d*),(*b*,*d*),(*c*,*d*), (*a*,*e*),(*b*,*e*),(*c*,*e*),(*d*,*e*)	(*a*,*c*),(*b*,*c*),(*a*,*d*), (*b*,*d*),(*a*,*e*),(*b*,*e*)

Figure 2-5: The derivation process of UIEC

2.2.2. Discussion on UIEC

As pointed out earlier, loop 1 and loop 3 of UIEC are actually implementations of a total closure algorithm. Loop 1 finds the total closure of 0th-frontier-edges, and loop 3 computes the total closure of *newSub*, i-th-edges. Much research has been done on total closure evaluation, involving good algorithms like the Logarithmic Algorithm [9], Warshall's Algorithm [21] and Warren's Algorithm [20], etc. These algorithms are proved to be very efficient in total closure processing. In this section, we will modify the algorithm UIEC by employing the Logarithmic Algorithm in its loop 1 and loop 3. The disk I/O performance of the original and the improved version are then compared. The new version for UIEC is as follows:

```
SubCl := $\sigma$A
F := $\Pi_2$(SubCl) - $\Pi_1$(SubCl)

SubCl := TotalCl( SubCl )

while ( F $\neq \varnothing$ )
   newSub := F $\Delta$ A
   if ( newSub = $\varnothing$ ) exit the loop
   newSub2 := newSub $\Delta$ SubCl - newSub
   newSub := newSub $\cup_c$ newSub2

   newSub := TotalCl( newSub )

   newSub3 := SubCl $\Delta$ newSub - SubCl
   SubCl := SubCl $\cup_c$ newSub
   SubCl := SubCl $\cup_c$ newSub3

   F := $\Pi_2$(newSub) - $\Pi_1$(SubCl)
end while


Cl := $\sigma$SubCl


/* procedure TotalCl */
procedure TotalCl( rel )
    TCl := rel
    LW := rel
    sizeTCl := $|TCl|$
    loop
       LW := LW $\Delta$ LW
       if ( LW = $\varnothing$ ) then exit loop

       W := TCl $\Delta$ LW
       if ( W = $\varnothing$ ) then exit loop

       TCl := TCl $\cup$ LW

       TCl := TCl $\cup$ W

       if ( sizeTCl = $|TCl|$ ) then exit loop

       sizeTCl := $|TCl|$
    end loop

    return( TCl )
end TotalCl
```

For the comparison purpose, we perform experiments on the original and improved versions of UIEC using special data (Sawtooth data) that is favorable to UIEC. This kind of data will be discussed in Section 3.2. Different parameter values and different relation sizes are used, and the result is shown in Table 2-1. The second column of Table 2-1 corresponds to the number of starting nodes in Section 3.2 (in Figure 3-2). The third and the forth columns of Table 2-1 show the disk I/O[3] performance of the original and the improved versions of UIEC, respectively.

rel. size	#selected	original	improved
100	20	12046	1439
200	20	12058	1451
500	20	12092	1485
1000	20	12155	1549
1500	20	12213	1607
100	40	1075773	19983
200	40	1075785	19995
500	40	1075820	20030
1000	40	1075881	20091
1500	40	1075939	20149

Table 2-1: The disk I/O of UIEC is improved

The result shows that the disk I/O performance of the improved version is better than that of the original one. In fact, the number of times loop 1 and loop 3 of the original UIEC presented in Section 2.2 are executed is also logarithmic in the depth of the transitive closure. Take loop 1 as an example. The loop finds the total closure for $SubCl_0$. Joining $SubCl_0$ with itself yields $SubCl_0^2$. After the union operation, $SubCl$ contains $SubCl_0$ and

[3] Disk I/O is the number of pages swapped between disk and main memory

$SubCl_0^2$, and, *newSub* contains $SubCl_0^3$ and $SubCl_0^4$. In the second iteration, *SubCl* becomes the union of $SubCl_0$, $SubCl_0^2$, $SubCl_0^3$ and $SubCl_0^4$. Thus, when we join *newSub* with *SubCl*, we will get $SubCl_0^4 \cup SubCl_0^5 \cup \cdots \cup SubCl_0^8$. However, a large number of duplicates are produced. The result of joining $SubCl_0^3$ in *newSub* to $SubCl_0^i$ in *SubCl* is $SubCl_0^{i+3}$ which can also be produced by joining $SubCl_0^4$ to $SubCl_0^{i-1}$ in *SubCl*. Then in the third iteration, we can get the union of $SubCl_0$, $SubCl_0^2, \cdots, SubCl_0^{16}$ and so on. Hence, in logarithmic time of the depth of the transitive closure, we will get the total closure for $SubCl_0$. The implication of the results from Table 2-1 is twofolds:

1. The duplication produced at each iteration is an important factor in the disk I/O performance. The duplicates require extra storage. This implies that more buffer memory or else incurs high disk I/O will be required. Moreover, we can see that duplication removal is a very costly operation.

2. The selection of an efficient algorithm for total closure processing affects the disk I/O performance of UIEC. In other words, if an efficient total closure algorithm is used, then the performance of UIEC will be improved.

As the improved version of UIEC performs very well, in our performance studies, we will embed the Logarithmic Algorithm into UIEC, and do the analysis based on this improved version.

2.3. Unary Wavefront with Frontier-Edges (UWFE)

To reduce the redundancy inherited by BWFT algorithm, we devise UWFE. As in UIEC, we proceed by using a unary driver set (unary frontier). However, in each iteration, instead of computing the i-implied-edges-closure as in UIEC, UWFE will compute the i-closure. The computation is made possible by storing i-frontier-edges. UWFE has three important intermediate relations which are: i) the unary frontier *F*; ii) *Cl* which stores the i-closure, and iii) *PB* which stores i-frontier-edges. Both *Cl* and *PB* are initialized to be the set of those edges originating from the starting nodes, i.e., 0th-edges. *F* is initialized to be 1st-frontier. Note that, we use the same method as the one in UIEC to find the frontier. In each iteration of PB, *newPB*, which is i-th-edges, will be found and unioned to (i-1)-frontier-edges to obtain i-frontier-edges. Then the i-closure will be computed by making use of the (i-1)-closure. The algorithm of UWFE is stated as follows:

```
Cl := $\sigma$A
PB := Cl
F := $\Pi_2$(PB) - $\Pi_1$(PB)          /* compute the 1st-frontier */

      /*   Loop 1 : compute the 0-closure */
newCl := Cl $\Delta$ PB - Cl
while ( newCl $\neq \varnothing$ )
   Cl := Cl $\cup_c$ newCl
   newCl := newCl $\Delta$ PB - Cl
end while

while ( F $\neq \varnothing$ )
   newPB := F $\Delta$ A
   if ( newPB = $\varnothing$ ) exit the loop

   PB := PB $\cup_c$ newPB

      /*   Loop 2 : compute the i-closure from (i-1)-closure */
   newCl := Cl $\Delta$ newPB - Cl
   while ( newCl $\neq \varnothing$ )
       Cl := Cl $\cup_c$ newCl
       newCl := newCl $\Delta$ PB - Cl
   end while

   F := $\Pi_2$(PB) - $\Pi_1$(PB)      /* compute the (i+1)th-frontier */
end while
```

As in UIEC, the join operator Δ functions differently depending on whether its left operand is unary or binary.

Lemma 2-4: UWFE correctly stores the i-closure in *Cl* at the end of the i-th iteration.

Sketched Proof :

After the initialization, *PB* and *Cl* contain all the edges emanating from the starting nodes. Thus, *PB* and *Cl* are equivalent to 0-frontier-edges. As loop 1 is an implementation of BWFT, therefore, after the completion of loop 1, *Cl* must contain the 0-closure.

In each iteration of loop 2 of UWFE, i-th-edges are found using the join operation and are stored in *newPB*. By induction, *PB* contains the union of k-th-edges ($0 \leq k \leq i-1$), and so with *newPB* unioned to *PB*, we can obtain i-frontier-edges in *PB*.

Now, assume that *Cl* contains the (i-1)-closure. By joining *Cl* to *newPB*, we can get all the implied edges from the starting nodes to the driven elements of the current drivers (i-th-frontier) using nodes in i-th-frontier as the intermediate nodes. With loop 3 that follows, instead of joining *Cl* to *newPB*, if we join *Cl* to *PB*, we will have an implementation of BWFT on i-frontier-edges. However, using *PB* will not produce more implied edges to be added to the i-closure than using *newPB*. Let us assume to the contrary that one more implied edge, say (*x*,*z*), is found when using *PB*. This means that we have a tuple (*x*,*y*) in *Cl* and a tuple (*y*,*z*) in *PB*, for some *y*. The node *y* must be in k-th-frontier (0≤k≤i-1), because if it is in i-th-frontier, we must have (*y*,*z*) in *newPB* (i-th-edges), and this implies that (*x*,*z*) should have been found using *newPB*. This contradicts our assumption. Therefore, node *y* is in k-th-frontier and this implies that (*y*,*z*) is in k-frontier-edges. By definition, the implied edge (*x*,*z*) should be contained in the (i-1)-closure. Therefore, (*x*,*z*) cannot be distinct from edges in the (i-1)-closure, and thus, using *newPB* is the same as using *PB*. Now, loop 3 can be seen to be an implementation of BWFT, and therefore, the query closure of i-implied-edges (i.e., i-closure) will be found at the end of loop 3. Thus, *Cl* contains the i-closure at the end of iteration i. □

Theorem 2-5: The algorithm UWFE terminates and produces the correct closure in Cl.

Sketched Proof :

As UWFE has the same i-th-frontier as that of UIEC for each iteration, UWFE must terminate by Theorem 2-3.

As all the drivers must have been considered at the end of loop 2, *PB* must contain all the edges accessible from the starting nodes. By Lemma 2-4, *Cl* contains the i-closure which is the query closure of i-frontier-edges. Therefore, when UWFE terminates, *Cl* must be the final query closure. □

2.3.1. Example of Using UWFE

Consider Figure 2-1. As before, we take nodes *a* and *b* as the starting nodes. After the selection operation, we derive tuples (*a*,*c*) and (*b*,*c*) in *PB* and *Cl*. As there are no edges between nodes *a* and *b*, then the complete 0-closure will be stored in *Cl*, and so we can proceed to the second loop. At the first iteration, *F* contains the driver *c*. With the driver, after the join operation, we get (*c*,*d*) which is then stored in *newPB*. Then this new tuple is added to *PB*. Computing 1-closure, we get {*(a,c), (b,c), (a,d), (b,d)*}. The only driven element is node *d*, which has not been explored, so it is the only element in 2nd-frontier. At the second iteration, (*d*,*e*) is the only element in *newPB* after the join operation and it is added to *PB*. The new implied edges (*a*,*e*) and (*b*,*e*) are formed by joining *Cl* and *newPB* in the computation of the 2-closure. The set of these tuples is then unioned with *Cl*. Then node *e* will be found as the only element in 3rd-frontier. Since node *e* does not have any outgoing arc, and so the loop terminates. The derivation process of UWFE is also shown in Figure 2-6.

Iteration	Old *F*	New *F*	Old *PB*	*PB*	*Cl*
1	*c*	*d*	(*a*,*c*),(*b*,*c*)	(*a*,*c*),(*b*,*c*),(*c*,*d*)	(*a*,*c*),(*b*,*c*),(*a*,*d*),(*b*,*d*)
2	*d*	*e*	(*a*,*c*),(*b*,*c*),(*c*,*d*)	(*a*,*c*),(*b*,*c*),(*c*,*d*),(*d*,*e*)	(*a*,*c*),(*b*,*c*),(*a*,*d*),(*b*,*d*) (*a*,*e*),(*b*,*e*)
3	*e*		(*a*,*c*),(*b*,*c*),(*c*,*d*),(*d*,*e*)	(*a*,*c*),(*b*,*c*),(*c*,*d*),(*d*,*e*)	(*a*,*c*),(*b*,*c*),(*a*,*d*),(*b*,*d*), (*a*,*e*),(*b*,*e*)

Figure 2-6: The derivation process of UWFE

2.3.2. Discussion on UWFE

UWFE and UIEC use the same driver set for each iteration. However they use different methods of derivation. While UIEC uses the i-implied-edges-closure to record the derivation path, UWFE uses i-frontier-edges to record only edges from one driver to other driver(s). The use of the i-implied-edges-closure will provide complete information of the derivation process, in the sense that whenever a driver *b* is revisited while a node *a* is being explored, all the reachable nodes leading from *b* can be immediately recognized as successors of *a*. This means that by one join operation, we can get all the implied edges

$\{(a,x) : (b,x)$ is an implied edge leading from $b\}$.

In contrast, the use of i-frontier-edges only gives partial information. We only know the direct edges leading from j-th-frontier $(0 \leq j \leq i)$, i.e., j-th-edges. Thus every time when a node *b* is revisited, the successors of node *b* are not explicitly stored, and we have to use node *b* as the root and traverse through the edges in *PB* to find the successors of *b*.

Therefore, when many such nodes are revisited throughout the derivation stage, UWFE has to find the successors of these nodes repeatedly. This implies that more I/O is needed for UWFE. On the other hand, in order to provide such complete information on the derivation stage, UIEC needs extra effort to update the i-implied-edges-closure. If there are not many nodes being revisited, then the effort of updating the implied edges for each of the encountered drivers will become a substantial overhead. Thus, there is a tradeoff between providing the full derivation information and the extra effort of updating the i-implied-edges-closure. In the performance studies presented in Chapter 6, we will demonstrate such tradeoffs.

2.4. Unary Wavefront Preprocessing with Total-Frontier-Edges (UPFE)

UPFE is a preprocessing algorithm. It first finds total-frontier-edges and then finds the query closure of total-frontier-edges. To accomplish the task, UPFE computes the query closure by the direct application of BWFT on total-frontier-edges. UPFE employs intermediate relations *PB* to record the i-frontier-edges set, *F* to store i-th-frontier (current drivers), and *Cl* to collect the i-closure. As UWFE, *PB* and *Cl* are initialized to be the set of edges leading from the starting nodes. The frontier *F* is found by using the same method as that in UIEC and UWFE. There are two stages in UPFE. The first stage is to find i-frontier-edges (the preprocessing stage), and the second stage is to find the i-closure (the propagation stage). The preprocessing stage is the iterative process of finding a new driver buffer by exploring the current drivers. The new drivers are all the unexplored direct successors of the current drivers. Edges leading from the current drivers will be added to *PB* as in UWFE. Each iteration of the propagation stage will require i-th-edges to be found and stored in an intermediate relation *newCl* which will then be unioned to *Cl*. The UPFE algorithm is stated as follows:

```
Cl := $\sigma$A
PB := Cl
F := $\Pi_2$(PB) - $\Pi_1$(PB)

/*   preprocessing stage   */

while ( F $\neq \emptyset$ )
   newPB := F $\Delta$ A
   if ( newPB = $\emptyset$ ) exit the loop

   PB := PB $\cup_c$ newPB

   F := $\Pi_2$(newPB) - $\Pi_1$(PB)
end while

/*   propagation stage   */

newCl := $\Pi$( Cl $\Delta$ PB - Cl )
while ( newCl $\neq \emptyset$ )
   Cl := Cl $\cup_c$ newCl
   newCl := newCl $\Delta$ PB - Cl
end while
```

Theorem 2-6: UPFE terminates and correctly computes the query closure upon termination.

Sketched Proof :

The preprocessing stage terminates because there are at most 2n distinct drivers (n = size of the base relation *A*), and also only i-th-frontier are considered for each iteration. Besides, at each iteration, we record i-th-edges in *PB*. In other words, we explore the current drivers, and record the direct edges leading from these current drivers. Hence, the active driver set must be exhausted at some iteration and the loop terminates. Moreover, all the drivers driven from the starting nodes must be considered. Thus we must have recorded all the driving-driven pairs (total-frontier-edges) in *PB* upon termination of the preprocessing stage.

All the edges needed for the derivation process are recorded after the preprocessing stage, and using these edges is sufficient to derive the query closure. That is, we have eliminated those useless edges (edges that are not incident on the successors of the starting nodes) from the base relation *A*. In the propagation stage, we find the query closure on the smaller relation *PB* instead of the larger relation *A*. As we use BWFT for this propagation process, upon termination, *Cl* must collect the query closure properly, by Theorem 2-1. □

2.4.1. Example of Using UPFE

Let us consider Figure 2-1 again. The preprocessing stage is started by initializing *PB* and *Cl* to be (*a,c*) and (*b,c*). The only driver found is node *c*. With driver *c* in 1st-frontier, we find *newPB* to be (*c,d*) which is then added to *PB*. Then in the next iteration, the driver *d* in 2nd-frontier has only one edge (*d,e*) leading from it, and so *newPB* will contain (*d,e*) only. With *newPB* unioned to *PB*, we now have in the *PB* (i-frontier-edges) the tuples {*(a,c), (b,c), (c,d), (d,e)*}. Since node *e* is not drivable, the preprocessing stage is completed. The first iteration of propagation stage will add the tuples (*a,d*) and (*b,d*) to *Cl*. Then in the next iteration, the two tuples (*a,e*) and (*b,e*) will be added. Now, *newCl* is empty, and the loop terminates. The query closure found is {*(a,c), (b,c), (a,c), (b,d), (a,e), (b,e)*}. The derivation process of UPFE is also shown in Figure 2-7.

Preprocessing Stage of UPFE

Iteration	Old *F*	New *F*	Old *PB*	*PB*
1	*c*	*d*	*(a,c),(b,c)*	*(a,c),(b,c),(c,d)*
2	*d*	*e*	*(a,c),(b,c),(c,d)*	*(a,c),(b,c),(c,d),d,e)*
3	*e*		*(a,c),(b,c),(c,d),(d,e)*	*(a,c),(b,c),(c,d),(d,e)*

Propagation Stage of UPFE

Iteration	Old *Cl*	*newCl*	*Cl*
1	*(a,c),(b,c)*	*(a,d),(b,d)*	*(a,c),(b,c),(a,d),(b,d)*
2	*(a,c),(b,c),(a,d),(b,d)*	*(a,e),(b,e)*	*(a,c),(b,c),(a,d),(b,d),* *(a,e),(b,e)*

Figure 2-7: The derivation process of UPFE

2.4.2. Discussion on UPFE

Both UPFE and UWFE find and record i-frontier-edges, and stores these direct edges emanating from the drivers in *PB*. The algorithms will also compute the i-closure. However, the computation of i-closure in UWFE is done within the loop of deriving the driver sets, but that in UPFE is done after the drivers derivation process is completed. Thus, they are different in the order of the relational database operations. It is this ordering that makes the two algorithms different in their relation reference pattern (RRP) and values of Least Maximum Buffer Requirement (LMBR), which will be treated more formally in Chapter 5. Moreover, reference localities of the two algorithms are different. Independence of driver derivation and i-closure update processing enables UPFE to concentrate on fewer relations in each stage. Thus, the chance that these relations are in the main memory will be higher. In particular, the data pages of the base relation *A* needed during the join operation, which is a very costly operation, in the preprocessing stage of UPFE will be more likely in the main memory than those data pages needed during the corresponding join operation in UWFE. On the other hand, updating i-closure inside the driver derivation loop has its advantage. The merit of the action lies in the data clustering property of the relations during the updating process. To be precise, each iteration of the updating process of UWFE will encounter much smaller relations than that of UPFE. Hence, there will be a higher chance for those relations to be found in the main memory, and thus there will be less disk traffic. All of these factors contribute to the difference in disk I/O performance of UPFE and UWFE. In Chapter 5, these factors will be considered, and the difference in disk I/O performance of the two algorithms will be discussed in more detail in Chapter 6.

Chapter 3
Algorithm Analysis On Some Typical Data Sets

In this chapter we will consider some special data sets in order to get a deeper understanding of the algorithms. These special data sets are arranged such that one of the algorithms will be seen to outperform all others. For each data set, simulation experiments are carried out to compare the disk I/O performance of the algorithms. A small buffer size (10 pages) is used in each experiments, and each of the pages is assumed to hold 20 binary tuples[4].

3.1. Parallel-Chains Data

When the base relation contains only independent chains of data like that in Figure 3-1, we refer the kind of data as **Parallel-Chains Data**. The base relation may contain complex relationships among the nodes, but from the starting nodes chosen to their descendants, the outdegree and in-degree of the nodes must be equal to or less than 1.

In the discussion of BWFT, we pointed out that extra processing occurs when some of the drivers have common descendants, and the path leading from those descendants to their descendants may be revisited. However, if no such case occurs, that is, if the base relation has nodes with indegree ≤ 1, then each of these nodes cannot be revisited by any search mechanism. Parallel-Chains Data has this property. This means that BWFT satisfies the condition that no path is revisited as other algorithms, but without the overhead. The only difference is the arity of the driver (binary vs. unary). Thus BWFT is expected to outperform the others for this kind of data set. In fact, the number of iterations that BWFT

[4] The buffer size and page size will be discussed in Section 6.1.

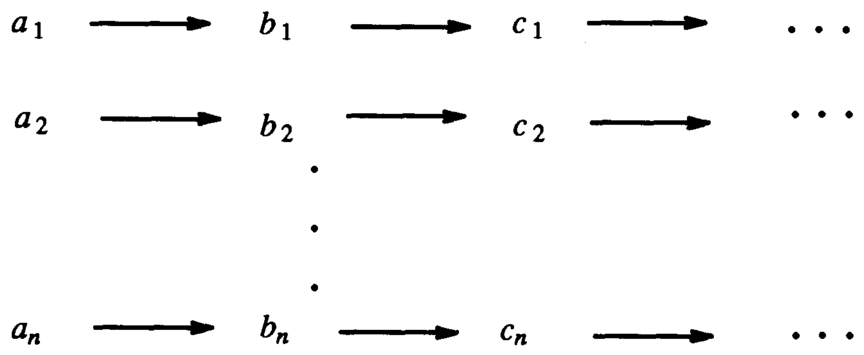


Figure 3-1: Parallel-Chains Data

takes is exactly the same as those for UIEC and UWFE, thus eliminating the motivation of UIEC and UWFE, as stated in Chapter 2.

We have generated this kind of data with 2 parameters and compared the results of the algorithms. The parameters are: the number of starting nodes and the depth of each chain (all chains have equal depth). The results are shown in Table 3-1. From Table 3-1, we can see that the experimental results agree with the discussion we have made above.

3.2. Sawtooth Data

We refer to those relations that are composed of a long single chain with each odd node as a starting node as the **Sawtooth Data**. Figure 3-2 shows this kind of data.

In Figure 3-2, $a_1, \cdots, a_n$ are all the starting nodes, and the closure for the relation is the set of tuples such that $Cl = \{ (a_i,a_j) : j < i \} \cup \{ (a_i,b_j) : j \leq i \}$. Since this data set has a long chain, BWFT will iterate the loop for 2n-2 times in order to find the closure. Both UWFE and UIEC will exit from its second loop after one iteration. As a preprocessing algorithm, UPFE does not perform better than BWFT because all the edges needed for

rel. size	length	BWFT	UWFE	UPFE	UIEC
100	20	34	268	55	725
200	20	54	279	72	735
500	20	81	315	101	769
1000	20	187	428	197	877
1500	20	256	489	267	937
100	40	226	885	589	3719
200	40	260	920	623	3756
500	40	420	1075	819	3912
1000	40	480	1133	878	3967
1500	40	536	1190	934	4026

Table 3-1: Disk I/O performance of the algorithms on Parallel-Chains Data

propagation can be found in one iteration. Sawtooth data requires UPFE to put much effort on its propagation stage which is just the implementation of BWFT. Even though UWFE and UIEC exit from their respective loops after the same number of steps, the processing inside the loop makes a difference in their disk I/O. The computation of the i-closure inside loop 2 of UWFE is an implementation of BWFT; hence loop 3 of UWFE (loop that carries out the i-closure computation) will iterate 2n - 2 times in its loop 2. However, finding the i-implied-edges-closure inside loop 2 of UIEC requires only ln(2n-2) steps. (Recall that a logarithmic algorithm is used for the total closure evaluation.) This makes UIEC outperform the other algorithms for this kind of data.

The experimental results are shown in Table 3-2. The data set generated has two

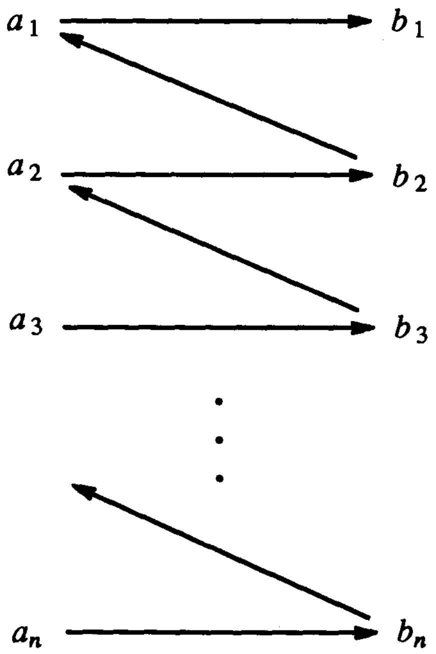


Figure 3-2: Sawtooth Data

parameters, i.e., the number of starting nodes, and the size of the base relation. The first and second columns of Table 3-2 correspond respectively to the relation size and the number of starting nodes. The third to sixth columns correspond to the disk I/O of BWFT, UWFE, UPFE and UIEC respectively. The result in Table 3-2 shows that UIEC has the best disk I/O performance. This confirms with the above discussion.

rel. size	#selected	BWFT	UWFE	UPFE	UIEC
100	20	2578	6450	6076	1439
200	20	2590	6462	6088	1451
500	20	2624	6496	6122	1485
1000	20	2726	6558	6186	1549
1500	20	2784	6616	6244	1607
100	40	25394	55879	53705	19983
200	40	25471	55891	53717	19995
500	40	25541	55926	53752	20030
1000	40	25661	55986	53812	20091
1500	40	25727	56044	53870	20149

Table 3-2: Disk I/O performance of the algorithms on Sawtooth Data

3.3. One-Cycle Data

When the base relation contains one large cycle and there is a direct edge from each starting node to a node on the cycle, we refer this kind of data as **One-Cycle Data**. In Figure 3-3 we have a One-Cycle Data set with two starting nodes a_1 and a_2. In our experiment with this kind of data, there are two parameters: the cycle length and the relation size. The results of the experiment are tabulated in Table 3-3.

The results indicate that UPFE outperforms the others when the cycle length is of moderate size, such as 20. When the length of the cycle increases, BWFT becomes the

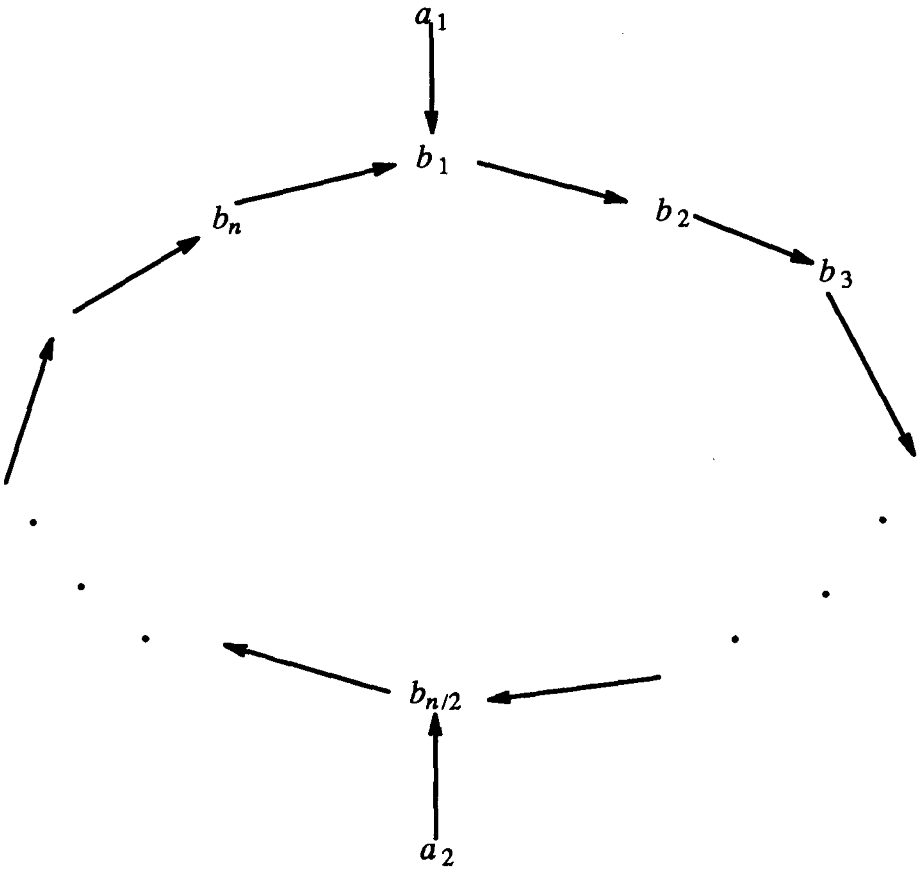


Figure 3-3: One-Cycle Data

best. Generally speaking, the performance gap between UPFE and BWFT narrows as the size of the relation increases. UIEC is the worst in this kind of data. This is because the effort to update the i-implied-edges-closure is a waste. For this kind of data, only the implied edges leading from b_1 and $b_{n/2}$ (see Figure 3-3) are useful. Hence, UIEC has a higher overhead than the other algorithms and its disk I/O performance becomes the worst. When we examine the data more closely, we will find that it exhibits the property of Parallel-Chains Data. For example, a_1 to $b_{n/2}$ is a long chain, and $b_{n/2}$ to b_1 is also a long chain. When the length of the cycle increases, the property of Parallel-Chains Data becomes more apparent. However, for moderate cycle length, UPFE has the least disk I/O. The reason is that it preprocesses the base relation to a smaller relation (total-frontier-edges *PB*), and then finds the closure by just focusing on *PB*. Since the relation *PB* is smaller than the original base relation, the page swapping during the propagation stage is reduced, and hence, the disk I/O is small. UWFE stores i-frontier-edges and computes the i-closure at each iteration. In a small memory buffer environment, this will make more frequent disk swapping of these intermediate relations (*PB* and *Cl*, etc.) for processing, and hence, UWFE will have higher disk I/O than that of UPFE. Therefore, UPFE will outperform the others on moderate cycle lengths.

rel. size	length	BWFT	UWFE	UPFE	UIEC
100	20	30	118	28	298
200	20	59	127	48	309
500	20	121	161	101	348
1000	20	208	276	196	441
1500	20	281	339	263	505
100	40	42	308	60	1946
200	40	66	319	81	1960
500	40	145	348	148	1997
1000	40	306	492	273	2117
1500	40	369	550	331	2175
100	60	57	500	163	5715
200	60	80	511	181	5725
500	60	149	533	243	5753
1000	60	392	708	432	5910
1500	60	450	766	491	5968

Table 3-3: Disk I/O performance of the algorithms on One-Cycle Data

Chapter 4
Database Model

In this chapter, we will describe the database model we use. The introduction of our database model is to provide the readers with information about the environment in which the simulation experiments will take place (Chapter 6). The buffer requirements of the algorithms are analyzed in Chapter 5.

4.1. Architecture of the Database System

As described in logic and database literature, a deductive database system is divided into an intensional database (IDB) system and an extensional database (EDB) system. The intensional database consists of deductive rules and integrity constraints whereas the extensional database is the set of data stored in the database. Normally, IDB and EDB reside in the disk, and are brought to the main memory when needed. Since we use the compilation approach (see Chapter 1) to query processing, the rules in IDB will be compiled first into one of the query processing algorithms we have. The query processing algorithms are embedded in our system software. Query processing requires the system to access the EDB in the disk, and to manage the I/O between the main memory and the disk.

4.2. General View of the System Software

Our System Software Model is divided into Relational Data System (RDS) and Data Storage System (DSS). The input to RDS is the compiled query which consists of a set of relational database operations. In RDS, these operations will be transformed into sequences of read and write requests, and these operations are the input to the DDS module. Each of these read/write operations is at the tuple level, i.e., the interface of RDS and DDS.

DDS is further divided into two modules. The two software modules are File Structure System (FSS) and Buffer Management System (BMS). The tuple read/write operations from RDS are directed toward FSS. In fact, the operations can be sequential read/write or indexed read. A sequential operation is one that reads from and writes to a relation sequentially. For indexed read, a B+ tree structured indexing is built for the relation which has not been indexed. A tuple is accessed directly through the use of the index. FSS is the module responsible for these direct and sequential read and write operations at the tuple level. In order to carry out its job, FSS will convert the tuple read/write operations into page read/write operations which are then forwarded to BMS.

Within BMS, the buffer manager is responsible for managing a buffer pool of pages. Basically, the task of the buffer manager is to: 1) serve the page requests passed from FSS, 2) communicate with the disk manager in order to do read/write operation from/to disk, and 3) decide which page in the buffer is to be replaced when the buffer is full. BMS also acts as the interface to the disk manager.

In our simulation model, each of the above layers is implemented as a module. The communication between the modules is by subroutine invocation. The devices such as the main memory (buffer pool of pages) or the disk are simulated by data structures in our package.

4.3. Relational Database Operations

The transitive closure algorithms will be transformed into a sequence of relational database operations serving as input to RDS. Within RDS, the modules *Assign, Join, Select, Project, Diff* and *Union* which correspond to each of the six relational operations, **Assign, Join, Selection, Projection, Difference,** and **Union** will perform a series of read and write operations either sequentially or directly using indexing. The relational database operators are implemented as follows:

Assign(A,C) - This is perhaps the simplest operation. The function of this operation is to copy all the tuples of relation A to relation C. Hence a loop is needed to do a sequential read on relation A, and write on relation C.

Join(*j_attb,r_attb,A,B,C*) -

j_attb and *r_attb* are bit vectors representing the join attributes of the operands and the resulting attributes of the result relation, respectively. Using the bit vectors, we can arbitrarily choose any of the attributes of the operands to be the join attributes or resulting attributes.

In this research, we adopt the nested-loop indexing strategy to implement the join operation. The outer loop will do sequential read on one of the two relations. Using the joining attribute of the retrieved tuple as a key, we will find all the matching tuples of the other relation in the inner loop by direct read operations on that key. Then a resulting tuple will be formed for each pair of the matching tuples by examining the bit vector *r_attb*. The relation to be chosen in the outer loop is always the smaller relation. In Appendix A, we describe our experiments on two different choosing strategies for the inner and outer relations, and justify our choice.

Select(*attb,constant,A,C*) -

The parameter *attb* designates the selection attribute column. Since we are considering only unary or binary relations, *attb* can be either 1 or 2. A value of 1 denotes that the selection is done on the first attribute, and a value of 2 means the second attribute is the selection attribute. The result relation *C* can be unary or binary, depending on whether *A* is unary or binary. *constant* is an array which is the set of constants by which the selection is applied. If indices have already been built on the selection attributes, then direct read operation will be issued using the list of constants as keys; otherwise, relation *A* will be accessed sequentially to find all the tuples that have one of the values in the constant list in its selection attribute column.

Project(*attb,A,C*) - This operation not only performs projection, but also duplication removal. If relation *A* is unary, then the only job that can be done is removing all the duplicates of *A*. For binary relation *A*, projection can be done on the first or second attribute, and *attb* is used to represent which of the attributes should be applied on. Instead of having values "1" or "2", *attb* may have value "3". In that case, only duplication removal is done for the binary relation *A*.

In our implementation, we first sort the relation *A* on the projection attribute. Then the sorted relation will be scanned sequentially to find distinct values or tuples. Hence, the cost of this operation is the sorting cost, and the cost to read the sorted relation.

Diff(*A,B,C*) - The relations *A*, *B*, and *C* can all be unary or all be binary. All the tuples in *A* that are not in *B* will be written to *C*. Like join operation, this operation also requires a pair of nested loops. Sequential read operations will be done on relation *A* in the outer loop. For unary relation *A*, the attribute of the retrieved tuples will be used as the key to do an index read to relation *B* in the inner loop. If no such key can be found for relation *B*, then the key will be written to relation *C*. In the case of binary relation *A*, the first attribute of the retrieved tuple of *A* will be used as the key to the index search in relation *B*. If no equivalent tuple is found in *B*, then the tuple will be written to *C*.

Union(*A*,*B*,*C*) - Relation *B* will be unioned to relation *A*. These relations can be both unary or both binary. There are two versions of Union operation. The first one is quite simple, and it just needs to read all the tuples of relation *B* and writes them to the end of relation *A*. The other version is equipped with a mechanism for duplication removal. In that version, a pair of nested loops are needed. The outer loop will read *A* sequentially, and the second loop is to use the first attribute of *A* as the key to do an index search on relation *B*. If no occurrence of the retrieved tuple of *A* is found on *B*, then the tuple will be written to the end of *A*. In addition to writting the tuples of *B* to the end of the relation *A*, the two versions also build an index to those tuples added to *A* if an index table has been built for *A* before.

4.4. Other Operations Used in The Simulation Model

We implemented three more operations to free space for a relation which will not be used later on, to sort a relation on one specific attribute column, or to build an index for a relation. The three operations are *FreeSpace*, *SortRel* and *BuildIndex*. Moreover, our algorithms sometimes require the application of the same sequence of relational database operators. For easy analysis and improved performance, we have added two operators both of which combine two relational database operations. These are the operators for **join and difference**, and **project and difference** operation. We call these operators *JoinDiff* and *ProjDiff*.

FreeSpace(*A*) - This operation will free all the storage occupied by the relation *A*. This operation proved in some of our tests to be useful because when we have only limited buffer space (main memory), releasing those unwanted pages in the main memory will allow useful pages to remain in main memory without being swapped out. Without this command, it is very hard to use any replacement algorithm to ensure that all the pages that are no longer needed will be chosen to be swapped. Therefore, using this command, the overall performance will be improved.

SortRel(*attb*,*A*) - As before, *attb* denotes the attribute the operation is applied on. We have implemented the quick sort algorithm for sorting the relation. In our simulation model, the base relation will be sorted on the first attribute before any of the binary query closure algorithms are executed. The disk I/O performance is improved by sorting the base relation first, because all the drivers with the same value will be clustered together.

BuildIndex(*attb*,*A*) -

A B+ tree indexing will be built on the attribute column designated by *attb*. This operation will be applied on the base relation at the beginning of all the algorithms because we always need the first attribute of the base relation as the key for finding driver sets.

JoinDiff(*j_attb*,*r_attb*,*A*,*B*,*C*,*D*) -

j_attb and *r_attb* are the bit vectors for choosing the join attributes and result attributes of joining relations *A* and *B*. This operation finds the difference of the result relation of joining *A* and *B* from *C*. The difference will be stored in *D*.

Like the join operator, this operator is also implemented with a pair of nested loops. In its second loop, after a matching tuple *t* is found, the tuple will be checked immediately against relation *C* for occurrence. This test is carried out by using the tuple as the key to the index table of *C*. That is to say, we also need index table for *C*. Our implementation of difference operation also requires the index table, so it is not an extra effort to build an index table for relation *C* here. If no equivalent tuple *t* is found in *C*, then *t* will be written to relation *D*. Thus, no intermediate relation is needed to store the intermediate result of joining *A* and *B*.

ProjDiff(*A*,*B*,*C*) - All the three relations are binary. This operation finds the projection of *A* on the second attribute, and the projection of *B* on the first attribute, and then stores the difference of the former projection from the latter projection into *C*.

A will be sorted on the second attribute. Then the sorted relation will be scanned sequentially to find distinct values. Each of the distinct values will be used as the key to access the index table of *B* on its first attribute. If the index entry exists for the key, then the key will not be recorded; otherwise, the key will be written to *C*. Therefore, there is no need for intermediate relations for the two projections, and hence I/O performance will be improved.

4.5. File Manager

The File Manager manages the interface between File Structure System (FSS) and Buffer Mangement System (BMS) inside Data Storage System (DDS). Upon receiving the tuple requests from Relation Data System (RDS), the File Manager will find out whether the requests are sequential or directed ones. If sequential operations are needed, then the File Manager will convert the requests to page requests and then pass the page request to the

Buffer Manager. The conversion is a trivial calculation as long as we know the number of tuples that can reside in a page. When a direct request is passed from RDS, then there are two things the File Manager needs to do. First, the File Manager needs to see whether the relation already has an index. If not, then an index table must be built on the relation. Second, the File Manager should initiate an index search. The index search requires the traversal of the B+ tree table built for the relation until a leaf node is reached. At the leaf level, the address of the data pages where the tuples with the key in question can be found. Readers interested in the B+ tree structure are referred to [3].

4.6. Buffer Manager

In addition to the buffer pool (main memory) mentioned earlier, there are some system buffers within Buffer Mangement System (BMS). These system buffers simulate cache memory. In our system, we have in total 2 buffers for reading, 1 buffer for writing, and 1 buffer for indexing.[5] All these buffers (the main memory and the cache memory) are managed by the Buffer Manager. When a page request arrives, the Buffer Manager will first find out whether the page is in one of the system buffers. In fact, a read request will entail a search in the read buffers, and a write request will entail a search in the write buffer. If the page is found in one of those buffers, then it will be read or written according to the request. When the page is not in the system buffers, then it will try to search in the main memory, and finally to search in the disk when the page is not in main memory either.

4.7. Disk Manager

There are 3 main tasks which the Disk Manager is responsible for:

1. To load the base relation into the disk storage when the system is first initialized.

2. To read and write some tuples to a page in the disk.

[5] A buffer in our context means memory large enough for a page of tuples.

3. To communicate with the Buffer Manager to copy a page from/to in the main memory to/from the corresponding page in the disk.

Chapter 5
Buffer Requirement Analysis

In this chapter, the analytical tools which are the basis for the analysis in the performance studies we will carry out in the following chapters are described. This chapter will then be proceeded by applying the analysis tools to each of the algorithms described in Chapter 2. A general comment of these algorithms based on the analysis will then be followed.

5.1. The Reference Pattern and the Least Maximum Buffer Requirement

In this section, we will introduce the analytical tools we will use in our performance studies. Now let us first define the **Least Maximum Buffer Requirement** (LMBR) of an algorithm as follows:

> **Definition 5-1:** The LMBR is the least buffer size required by an algorithm so that all the base relations will be read only once into the main memory and the intermediate relations will reside in the main memory whenever needed without generating any disk I/O. When an algorithm reaches its LMBR, increase in the buffer size will not improve the disk I/O performance of the algorithm.

When the buffer size is large enough, then the base relation or intermediate relations need not be swapped between the disk and main memory, and thus minimum disk traffic is incurred. However, in most situations, the amount of memory available for allocation as buffer space is rather limited and the LMBR of the algorithm will be a useful indicator for the amount of memory in the buffer required to ensure good disk I/O performance. If a number of algorithms are compared to determine their disk I/O performance, the algorithm with the smallest LMBR will be a good choice, especially when main memory is considered as a resource that need to be conserved. Obviously, one cannot guarantee that LMBR is always **well-behaved**, i.e. the I/O performance of an algorithm will smoothly converge to its minimum as the buffer size approaches its LMBR. However, our experience with LMBR here as well as the results reported in [15] convince us that the LMBR is a very well-behaved indicator. With LMBR, we can compare the disk I/O performance of alternative algorithms (in this thesis, the algorithms for comparison were described in Chapter 2). To estimate the LMBR of an algorithm, we need to know how the relations are accessed. The **Relation Reference Pattern** (RRP), which records the accessing pattern of the relation, is defined as follows:

> **Definition 5-2:** An RRP of an algorithm is the chronological order of relations (base and intermediate) by which the algorithm accesses the relations.

In order to show the relation reference pattern of some sequence of operations, we have the notation: A → B. This means that the access to the relation *A* is followed by the access to relation *B*. If "→" is used repeatedly, then the sequence of relations will be made explicitly. For example, if we have:

$$R_1 \rightarrow R_2 \rightarrow \quad \cdots \quad \rightarrow R_n$$

then we know that the operation requires access to relation R_1 first, then to relation R_2, and so on, to R_n.

A relational operation will usually cause reference to its operands and the result relation. However, sometimes, a relational operation will access other relations, such as the index table[6] of an operand. For example, the join operation as described in section 4.3 requires the access to one of its operand *A* to create the index table for *A*, the other operand *B* to access a tuple *t*. Let k be the join attribute of tuple *t*. Then the join operation accesses the index table of *A* to find the address of a matching tuple with key k. Finally, it accesses the relation *A* to find the matching tuples, and places the result tuple into relation *C*. Thus, the RRP for the join operation on *A*, *B* to get *C* will be :

$$A \rightarrow B \rightarrow A_{index} \rightarrow A \rightarrow C$$

where A_{index} is the index for *A*.

Other relational operators like difference, select, project will also access the index

[6] The index table itself is also represented as a relation.

relation. Since the RRP of an algorithm is generally quite complicated, we will make some assumptions to simplify the process of deriving RRP. In the sections that follow, we will consider only data pages. The index relation will not be considered. The size of the index table of a relation is always proportional to that of the relation. Normally, accesses to the index of a relation will also be followed by accesses to the relation itself. Thus, if we have to access a large relation, we expect its large index table also to be accessed. Therefore, considering only the data pages will also provide an idea of the whole picture of relation-accessing including index.

5.2. RRP and LMBR for BWFT

As stated before, we will not consider index reference here. Before presenting the RRP of BWFT, we present the algorithm of BWFT, described in Chapter 2, for easy reference:

```
$Cl := \sigma A$
$F := Cl$
while ( $F \neq \emptyset$ )
   $F := F \Delta A - Cl$
   $Cl := Cl \cup_c F$
end while
```

We use the notation F_i and Cl_i to denote the binary frontier and the derived closure at the iteration i. As BWFT must terminate, it is possible to find some integer M_b such that BWFT stops at that iteration. The relation reference pattern (RRP) for BWFT looks like the following:

$$
\begin{array}{lrl}
\text{Stage} & 0: & A \rightarrow Cl_0 \rightarrow Cl_0 \rightarrow \\
 & 1: & F_0 \rightarrow A \rightarrow Cl_0 \rightarrow F_1 \rightarrow Cl_0 \rightarrow Cl_1 \rightarrow \\
 & 2: & F_1 \rightarrow A \rightarrow Cl_1 \rightarrow F_2 \rightarrow Cl_1 \rightarrow Cl_2 \rightarrow \\
 & & \cdots \\
 & i: & F_{i-1} \rightarrow A \rightarrow Cl_{i-1} \rightarrow F_i \rightarrow Cl_i \rightarrow Cl_i \rightarrow \\
 & & \cdots \\
 & M_b: & F_{M_b-1} \rightarrow A \rightarrow Cl_{M_b-1} \rightarrow F_{M_b} \rightarrow Cl_{M_b-1} \rightarrow Cl_{M_b}
\end{array}
$$

Stage 0 : Selection is done on A to get the closure Cl_0. Then Cl_0 is accessed again for the assign operation to get the frontier F_0.

Stage 1 : The first iteration of the loop.

. . .

Stage i : The frontier F_{i-1} at the previous iteration and the base relation A are joined. F_i is then obtained as the difference of the result relation and the closure Cl_{i-1}. The reader should note that we do not reference the result relation from the join operation. This is the consequence of the use of our new operator join and difference as described in section 4.4. After F_i is found, Cl_{i-1} will be accessed in the union operation with F_i to produce Cl_i.

. . .

Stage M_b : The last iteration of the loop. The final query closure Cl will be produced at this iteration.

It should be noted that in finding the Least Maximum Buffer Requirement for BWFT, after the derivation of new driver set F, the old one will be deleted. But the deletion can only be brought about after the formation of the new set. Hence, we must have a buffer large enough to hold the old and new sets for LMBR. However, the formation of new closure will not cause deletion of the old one. The union operation causes attachment of the new frontier set to the old closure to produce the new closure at that iteration. Now, the LMBR of BWFT is calculated to be:

$$\frac{|A| + |Cl| + 2|F|}{pagesize}$$

where *pagesize* is the number of tuples a system page can hold,
$|Cl|$ is the size of the final results (the closure),
$|F|$ is the size of the largest frontier.

5.3. RRP and LMBR for UIEC

As pointed out in Chapter 2, UIEC requires implementation of the total closure algorithm. Hence, the relation reference pattern (RRP) of UIEC depends on the implementation method. In order to show such dependency, we will use $\mathrm{RRP}_{TC}(rel)$ to represent the RRP of total closure algorithm to the relation *rel*. Before we show the RRP of UIEC, let us present UIEC algorithm again:

```
SubCl := $\sigma$A
F := $\Pi_2$(SubCl) - $\Pi_1$(SubCl)

SubCl := TotalCl( SubCl )

while ( F $\neq \emptyset$ )
   newSub := F $\Delta$ A
   if ( newSub = $\emptyset$ ) exit the loop
   newSub2 := newSub $\Delta$ SubCl - newSub
   newSub := newSub $\cup_c$ newSub2

   newSub := TotalCl( newSub )

   newSub3 := SubCl $\Delta$ newSub - SubCl
   SubCl := SubCl $\cup_c$ newSub
   SubCl := SubCl $\cup_c$ newSub3

   F := $\Pi_2$(newSub) - $\Pi_1$(SubCl)
end while

Cl := $\sigma$SubCl
```

We use F_i to denote i-th-frontier, and $SubCl_{i,j}$ to indicate the j-th step in getting the i-implied-edge-closure $SubCl_i$. Since UIEC must terminate, we should be able to find the last iteration M_u where UIEC complete its job[7]. The relation reference pattern for UIEC is then described as follows:

[7] We use a subscript "u" here instead of "b" used in BWFT before in order to distinguish that UIEC uses unary frontier, whereas BWFT uses a binary one.

$$
\begin{array}{rcl}
\mathtt{Stage\ 0} & : & \mathtt{A} \rightarrow \mathtt{SubCl}_{0,0} \rightarrow \mathtt{SubCl}_{0,0} \rightarrow \mathtt{F}_{1} \rightarrow \mathtt{RRP}_{\mathtt{TC}}(\mathtt{SubCl}_{0,0}) \rightarrow \\
 & & \mathtt{SubCl}_{0} \rightarrow \\
\mathtt{1} & : & \mathtt{F}_{1} \rightarrow \mathtt{A} \rightarrow \mathtt{newSub}_{1,0} \rightarrow \mathtt{SubCl}_{0} \rightarrow \mathtt{newSub}_{1,0} \rightarrow \\
 & & \mathtt{newSub2}_{1} \rightarrow \mathtt{newSub}_{1,0} \rightarrow \mathtt{newSub}_{1,1} \rightarrow \mathtt{RRP}_{\mathtt{TC}}(\mathtt{newSub}_{1,1}) \rightarrow \\
 & & \mathtt{newSub}_{1} \rightarrow \mathtt{SubCl}_{0} \rightarrow \mathtt{newSub}_{1} \rightarrow \mathtt{SubCl}_{0} \rightarrow \\
 & & \mathtt{newSub3}_{1} \rightarrow \mathtt{SubCl}_{0} \rightarrow \mathtt{newSub}_{1} \rightarrow \mathtt{SubCl}_{1,0} \rightarrow \\
 & & \mathtt{newSub3}_{1} \rightarrow \mathtt{SubCl}_{1} \rightarrow \mathtt{newSub}_{1} \rightarrow \mathtt{SubCl}_{1} \rightarrow \\
 & & \quad \ldots \\
\mathtt{i} & : & \mathtt{F}_{i} \rightarrow \mathtt{A} \rightarrow \mathtt{newSub}_{i,0} \rightarrow \mathtt{SubCl}_{i-1} \rightarrow \mathtt{newSub}_{i,0} \rightarrow \\
 & & \mathtt{newSub2}_{i} \rightarrow \mathtt{newSub}_{i,0} \rightarrow \mathtt{newSub}_{i,1} \rightarrow \mathtt{RRP}_{\mathtt{TC}}(\mathtt{newSub}_{i,1}) \rightarrow \\
 & & \mathtt{newSub}_{i} \rightarrow \mathtt{SubCl}_{i-1} \rightarrow \mathtt{newSub}_{i} \rightarrow \mathtt{SubCl}_{i-1} \rightarrow \\
 & & \mathtt{newSub3}_{3} \rightarrow \mathtt{SubCl}_{i-1} \rightarrow \mathtt{newSub}_{i} \rightarrow \mathtt{SubCl}_{i,0} \rightarrow \\
 & & \mathtt{newSub3}_{i} \rightarrow \mathtt{SubCl}_{i} \rightarrow \mathtt{newSub}_{i} \rightarrow \mathtt{SubCl}_{i} \rightarrow \\
 & & \quad \ldots \\
\mathtt{M}_{u} & : & \mathtt{F}_{M_u} \rightarrow \mathtt{A} \rightarrow \mathtt{newSub}_{M_u,0} \rightarrow \mathtt{SubCl}_{M_u-1} \rightarrow \mathtt{newSub}_{M_u,0} \rightarrow \\
 & & \mathtt{newSub2}_{M_u} \rightarrow \mathtt{newSub}_{M_u,0} \rightarrow \mathtt{newSub}_{M_u,1} \rightarrow \mathtt{RRP}_{\mathtt{TC}}(\mathtt{newSub}_{M_u,1}) \rightarrow \\
 & & \mathtt{newSub}_{M_u} \rightarrow \mathtt{SubCl}_{M_u-1} \rightarrow \mathtt{newSub}_{M_u} \rightarrow \mathtt{SubCl}_{M_u-1} \rightarrow \\
 & & \mathtt{newSub3}_{M_u} \rightarrow \mathtt{SubCl}_{M_u-1} \rightarrow \mathtt{newSub}_{M_u} \rightarrow \mathtt{SubCl}_{M_u,0} \rightarrow \\
 & & \mathtt{newSub3}_{M_u} \rightarrow \mathtt{SubCl}_{M_u} \rightarrow \mathtt{newSub}_{M_u} \rightarrow \mathtt{SubCl}_{M_u} \rightarrow \\
\mathtt{M}_{u}\mathtt{+1} & : & \mathtt{SubCl}_{M_u} \rightarrow \mathtt{Cl}
\end{array}
$$

Stage 0 : Selection on base relation A to get $SubCl_{0,0}$. The 1st-frontier F_1 is obtained next as the difference of the second attribute and the first attribute of $SubCl_{0,0}$. Then total closure is to be found for $SubCl_{0,0}$ to get the 0-implied-edges-closure $SubCl_0$.

Stage 1 : The loop starts.

. . .

Stage i : The i-th-frontier F_i and base relation A are accessed in a join operation to get $newSub_{i,0}$. Together with $SubCl_0$, $newSub_{i,0}$ is referenced again in the join and difference operation to produce $newSub2_i$. $newSub_{i,1}$ is produced after the union operation of $newSub_{i,0}$ and $newSub2_i$. Total closure operation is then performed on $newSub_{i,1}$ to get $newSub_i$. The (i-1)-implied-edges-closure $SubCl_{i-1}$ and $newSub_i$ will then be referenced in another join and difference operation. The result of the operation is stored in $newSub3_i$. The i-implied-edges-closure $SubCl_i$ is obtained by performing the union of $SubCl_{i-1}$ first with $newSub_i$, and then with $newSub3_i$. Finally, $newSub_i$ and $SubCl_i$ are involved in another projection and difference operation to get the (i+1)-th-frontier F_{i+1}.

. . .

Stage M_u : The loop terminates. The M_u-implied-edges-closure will be found and stored in $SubCl_{M_u}$.

Stage M_u+1 : Selection on $SubCl_{M_u}$ to get the closure Cl.

Before we can calculate the LMBR for UIEC, we should know the RRP of the total closure we used. Here, we will use the Logarithmic Algorithm which has been presented in Section 2.2.2. For easy reference, we present the procedure here again:

```
procedure TotalCl( rel )
    TCl := rel
    LW := rel
    sizeTCl := $|TCl|$
    loop
       LW := LW $\Delta$ LW
       if ( LW = $\emptyset$ ) then exit loop

       W := TCl $\Delta$ LW
       if ( W = $\emptyset$ ) then exit loop

       TCl := TCl $\cup$ LW

       TCl := TCl $\cup$ W

       if ( sizeTCl = $|TCl|$ ) then exit loop

       sizeTCl := $|TCl|$
    end loop

    return( TCl )
end TotalCl
```

We denote the partially computed total closure at iteration i by TCl_i, and the j-th step in evaluating TCl_i by $TCl_{i,j}$. LW_i and W_i are the intermediate relations LW and W respectively computed at the i-th iteration. We assume that the invokation to the Logarithmic Algorithm will terminate in some iteration, say the L-th iteration. The RRP of the total closure algorithm on *rel* is as follows:

```
Stage 0  :  rel → $TCl_0$ → rel → $LW_0$ →
      1  :  $LW_0$ → $LW_1$ → $TCl_0$ → $TCl_{1,0}$ →
            $TCl_{1,0}$ → $LW_1$ → $W_1$ → $TCl_{1,0}$ → $TCl_1$ →
                          ...
      L  :  $LW_{L-1}$ → $LW_L$ → $TCl_{L-1}$ → $TCl_{L,0}$ →
            $TCl_{L,0}$ → $LW_L$ → $W_L$ → $TCl_{L,0}$ → $TCl_L$
```

The stage 0 of the Logarithmic Algorithm references to *rel* in two assign operations. Then inside the loop, *LW* will be referenced in a join operation. The updated *LW* is involved in a union and a join operation. *W* is then accessed in a union operation to produce the total closure TCl_i at that iteration.

In the Logarithmic Algorithm, the old *LW* at each iteration will be deleted after the formation of new *LW*, so we should have a buffer large enough for the new and old *LW* for LMBR. Moreover, the new *LW* is used to find *W* at each iteration, so *W* will be formed after the old *LW* is deleted. Thus, LMBR of the algorithm is at least:

$$|LW| + \max(|LW|, |W|)$$

where $|LW|$ is the maximum size of all the LW_i found,
$|W|$ is the maximum size of all W_i,
max{x,y} is the function that computes the maximum of x and y.

Our implementation of total closure algorithm, TotalCl(*rel*) procedure in 2.2.2, will build the total closure by directly updating the relation *rel*. The relation *TCl* takes over the space occupied by *rel*, hence we saved some memory. Thus the LMBR for the total closure algorithm is:

$$\frac{|TCl| + |LW| + max(|LW|, |W|)}{pagesize}$$

where *TCl* is the final closure.

At each iteration in UIEC, the relation *A*, the implied edges from the current drivers to all their visited descendants in $newSub_i$ the intermediate relations for deriving the total closure (i.e., *LW* and *W*), intermediate relations for deriving the i-implied-edges-closure (i.e., $newSub2_i$ and $newSub3_i$), and the i-th-frontier F_i are needed. Thus, the iterative process requires the buffer size of:

$$\frac{|A| + |newSub| + |LW| + |W| + |newSub2| + |newSub3| + |F| + |SubCl|}{pagesize}$$

where *newSub2* and *newSub3* are the largest of
all the $newSub2_i$ and $newSub3_i$ respectively, and,
F is the largest driver set.

Once we get the total-implied-edges-closure *SubCl* (i.e. $SubCl_{M_u}$), we can release the storage of other relations including the base relation. Thus, the storage requirement for the last step is:

$$\frac{|SubCl| + |Cl|}{pagesize}$$

Therefore, the LMBR of UIEC is:

$$\frac{max\{(|A| + |newSub| + |LW| + |W| + |newSub2| + |newSub3| + |F|, |Cl|\} + |SubCl|}{pagesize}$$

5.4. RRP and LMBR for UWFE

As stated before, loop 1 and loop 3 of UWFE compute the i-closure by making use of the arcs directing from the nodes in i-th-frontier, and/or the (i-1)-closure from the previous iteration. A call to propagation procedure *propagate* needs 3 parameters: the closure to be updated (*Cl*), the arcs from directing from the current driver (*newPB*), and i-frontier-edges (*PB*). We present UWFE algorithm with the invocation to this procedure:

```
Cl := $\sigma$A
PB := Cl
F := $\Pi_2$(PB) - $\Pi_1$(PB)

Cl := Propagate(Cl,PB,PB)

while ( F $\neq \emptyset$ )
   newPB := F $\Delta$ A
   if ( newPB = $\emptyset$ ) exit the loop

   PB := PB $\cup_c$ newPB

   Cl := Propagate(Cl,newPB,PB)

   F := $\Pi_2$(PB) - $\Pi_1$(PB)
end while
```

The propagation is done using BWFT algorithm. We use Cl_0 to denote the initial closure (i.e. the (i-1)-closure), and Cl_i to signify the result of the modification of the closure from the previous iteration. We assume that the procedure will terminate at some iteration P. Thus, Cl_P will become the i-closure. The RRP of propagate(*Cl,newPB,PB*) is as follows:

```
Stage 0  :  $Cl_0 \rightarrow newPB \rightarrow Cl_0 \rightarrow newCl_0 \rightarrow$
      1  :  $Cl_0 \rightarrow Cl_1 \rightarrow newCl_0 \rightarrow PB \rightarrow Cl_1 \rightarrow newCl_1 \rightarrow$
                                  ...
      i  :  $Cl_{i-1} \rightarrow Cl_i \rightarrow newCl_{i-1} \rightarrow PB \rightarrow Cl_i \rightarrow newCl_i \rightarrow$
                                  ...
      P  :  $Cl_{P-1} \rightarrow Cl_P \rightarrow newCl_{P-1} \rightarrow PB \rightarrow Cl_P$
```

At stage 0, the join and difference operation on Cl_0 and *newPB* yields *newCl*$_0$. Then for each iteration i from stage 1 to P, a union operation is done on closure Cl_{i-1} and *newCl*$_{i-1}$ to get the new closure Cl_i. This is then followed by another join and difference operation on $newCl_{i-1}$, PB and Cl_i to get $newCl_i$ for the next iteration. The LMBR for this procedure is:

$$\frac{|Cl| + |newPB| + 2|newCl| + |PB|}{pagesize}$$

where $|newCl|$ is the maximum size of all $newCl$'s derived.

For the RRP of UWFE, F_i, Cl_i and PB_i are used to denote i-th-frontier, i-closure and i-frontier-edges respectively at the i-th iteration. As we use the same sets of i-th-frontier as that of UIEC in each iteration, UWFE will terminate at same iteration of UIEC, i.e. at M_u iteration. Now the RRP of UWFE looks like the following:

$$\begin{array}{rl}
\mathtt{Stage\ 0\ :} & \mathtt{A} \rightarrow \mathtt{Cl}_{0,0} \rightarrow \mathtt{PB}_0 \rightarrow \mathtt{PB}_0 \rightarrow \mathtt{F}_1 \rightarrow \\
 & \mathtt{Propagate(Cl}_{0,0}\mathtt{,PB}_0\mathtt{,PB}_0\mathtt{)} \rightarrow \mathtt{Cl}_0 \rightarrow \\
\mathtt{1\ :} & \mathtt{F}_1 \rightarrow \mathtt{A} \rightarrow \mathtt{newPB}_1 \rightarrow \mathtt{PB}_0 \rightarrow \mathtt{PB}_1 \rightarrow \\
 & \mathtt{Propagate(Cl}_0\mathtt{,newPB}_1\mathtt{,PB}_1\mathtt{)} \rightarrow \mathtt{Cl}_1 \rightarrow \mathtt{newPB}_1 \rightarrow \mathtt{PB}_1 \rightarrow \\
 & \quad\quad\quad\quad \cdots \\
\mathtt{i\ :} & \mathtt{F}_i \rightarrow \mathtt{A} \rightarrow \mathtt{newPB}_i \rightarrow \mathtt{PB}_{i-1} \rightarrow \mathtt{PB}_i \rightarrow \\
 & \mathtt{Propagate(Cl}_{i-1}\mathtt{,newPB}_i\mathtt{,PB}_i\mathtt{)} \rightarrow \mathtt{Cl}_i \rightarrow \mathtt{newPB}_i \rightarrow \mathtt{PB}_i \rightarrow \\
 & \quad\quad\quad\quad \cdots \\
\mathtt{M}_u\mathtt{\ :} & \mathtt{F}_{M_u} \rightarrow \mathtt{A} \rightarrow \mathtt{newPB}_{M_u} \rightarrow \mathtt{PB}_{M_u-1} \rightarrow \mathtt{PB}_{M_u} \rightarrow \\
 & \mathtt{Propagate(Cl}_{M_u-1}\mathtt{,newPB}_{M_u}\mathtt{,PB}_{M_u}\mathtt{)} \rightarrow \mathtt{Cl}_{M_u} \rightarrow \mathtt{newPB}_{M_u} \rightarrow \mathtt{PB}_{M_u}
\end{array}$$

Stage 0: Selection on A to get $Cl_{0,0}$; assignment of $Cl_{0,0}$ to get PB_0; projection and difference on PB_0 to get the 1st-frontier F_1; computation of the i-closure on $Cl_{0,0}$ to derive the 0-closure Cl_0.

Stage 1 : Beginning of the loop.

. . .

Stage i : Join operation on the i-th-frontier F_i and A to get $newPB_i$, all the arcs emanating from the current drivers. $newPB$ then involves in a union operation with the (i-1)-frontier-edges (PB_{i-1}) to produce the i-frontier-edges PB_i; using the (i-1)-closure Cl_{i-1} and $newPB$ to compute the i-closure Cl_i; projection and difference on $newPB_i$ and PB_i to derive the (i+1)-th-frontier F_{i+1}.

. . .

Stage M_u : The driver set is exhausted, and the final closure Cl_{M_u} is produced. The loop terminates.

We can see that each iteration requires buffers for F, A, $newPB$, PB, $newCl$ and Cl for LMBR. Hence the LMBR for UWFE is:

$$\frac{|A| + |Cl| + |newPB| + |PB| + |F| + 2|newCl|}{pagesize}$$

where Cl is the final closure,
$newPB$ is the largest $newPB_i$,
PB is total-frontier-edges,
F is the largest driver set encountered,
$newCl$ is the largest $newCl_i$ found.

5.5. RRP and LMBR for UPFE

UPFE algorithm can be divided into two stages: the preprocessing stage and the propagation stage. The pseudocode of UPFE (described in Chapter 2) is as follows:

```
Cl := $\sigma A$
PB := Cl
F := $\Pi_2$(PB) - $\Pi_1$(PB)


/*   preprocessing stage    */

while ( F $\neq \emptyset$ )
   newPB := F $\Delta$ A
   if ( newPB = $\emptyset$ ) exit the loop

   PB := PB $\cup_c$ newPB

   F := $\Pi_2$(newPB) - $\Pi_1$(PB)
end while


/*   propagation stage    */

newCl := $\Pi$( Cl $\Delta$ PB - Cl )
while ( newCl $\neq \emptyset$ )
   Cl := Cl $\cup_c$ newCl
   newCl := newCl $\Delta$ PB - Cl
end while
```

Cl_0 denotes the set of all arcs leading from the starting nodes (i.e., 0th-edges). F_i is i-th-frontier, $newPB_i$ is i-th-edges, and PB_i is i-frontier-edges. The preprocessing stage of UPFE will terminate at the same iteration, M_u, as that of the other unary frontier algorithms (UIEC and UWFE), as it has the same sets of i-th-frontier. The RRP of the preprocessing stage of UPFE is:

$$\begin{array}{rcl}
\textbf{Stage } 0 & : & \mathtt{A} \rightarrow \mathtt{Cl}_0 \rightarrow \mathtt{PB}_0 \rightarrow \mathtt{PB}_0 \rightarrow \\
1 & : & \mathtt{F}_1 \rightarrow \mathtt{A} \rightarrow \mathtt{newPB}_1 \rightarrow \mathtt{PB}_0 \rightarrow \mathtt{PB}_1 \rightarrow \mathtt{newPB}_1 \rightarrow \mathtt{PB}_1 \rightarrow \\
 & & \dots \\
i & : & \mathtt{F}_i \rightarrow \mathtt{A} \rightarrow \mathtt{newPB}_i \rightarrow \mathtt{PB}_{i-1} \rightarrow \mathtt{PB}_i \rightarrow \mathtt{newPB}_i \rightarrow \mathtt{PB}_i \rightarrow \\
 & & \dots \\
M_u & : & \mathtt{F}_{M_u} \rightarrow \mathtt{A} \rightarrow \mathtt{newPB}_{M_u} \rightarrow \mathtt{PB}_{M_u-1} \rightarrow \mathtt{PB}_{M_u} \rightarrow \mathtt{newPB}_{M_u} \rightarrow \mathtt{PB}_{M_u}
\end{array}$$

Stage 0 : Before getting into the loop, the preprocessing stage of UPFE will perform: selection on A, assignment of Cl_0 to produce PB_0 (0-frontier-edges), and projection and difference on PB_0 to get the 1st-frontier F_1.

Stage 1 : The beginning of the loop of preprocessing stage.

. . .

Stage i : Join operation on i-th-frontier F_i with A; union of the set of arcs leading from the current drivers, $newPB_i$, with (i-1)-frontier-edges PB_{i-1} to form i-frontier-edges; projection and difference operation on $newPB_i$ and PB_i to get a new driver set, the (i+1)-th-frontier F_{i+1}.

. . .

Stage M_u : The end of preprocessing stage. The total-frontier-edges PB_{M_u} (i.e. PB, all the arcs leading from the starting nodes and their descendants) is found. $newCl_i$ is the set of all the newly found implied edges leading from the starting nodes in the i-th iteration.

In the RRP of the propagation stage of UPFE that will be formulated below, we use Cl_i (i>0) to denote the partially derived query closure at the iteration i. Cl_0 as before will be 0th-edges. Since the propagation stage is an implementation of BWFT, so it will terminate at the M_b iteration as that in BWFT. The RRP for the propagation stage of UPFE looks like:

$$\begin{array}{rcl}
\textbf{Stage } 0 & : & \mathtt{Cl}_0 \rightarrow \mathtt{PB} \rightarrow \mathtt{Cl}_0 \rightarrow \mathtt{newCl}_0 \rightarrow \\
1 & : & \mathtt{Cl}_0 \rightarrow \mathtt{Cl}_1 \rightarrow \mathtt{newCl}_0 \rightarrow \mathtt{PB} \rightarrow \mathtt{Cl}_1 \rightarrow \mathtt{newCl}_1 \rightarrow \\
 & & \dots \\
i & : & \mathtt{Cl}_{i-1} \rightarrow \mathtt{Cl}_i \rightarrow \mathtt{newCl}_{i-1} \rightarrow \mathtt{PB} \rightarrow \mathtt{Cl}_i \rightarrow \mathtt{newCl}_i \rightarrow \\
 & & \dots \\
M_b & : & \mathtt{Cl}_{M_b-1} \rightarrow \mathtt{Cl}_{M_b} \rightarrow \mathtt{newCl}_{M_b-1} \rightarrow \mathtt{PB} \rightarrow \mathtt{Cl}_{M_b} \rightarrow \mathtt{newCl}_{M_b}
\end{array}$$

Stage 0 : Join and difference of relations Cl_0 and PB to get $newCl_0$.

Stage 1 : The propagation stage begins.

. . .

Stage i : Union of $newCl_{i-1}$ with Cl_{i-1} (the partially derived closure at the previous iteration) to get Cl_i; projection and difference operation on the current closure Cl_i and PB to form $newCl_i$.

. . .

Stage M_b : The propagation stage ends. The final closure Cl_{M_u} (i.e. the query closure Cl) is found.

The LMBR for preprocessing stage is:

$$\frac{|A| + |Cl_0| + |PB| + |newPB| + |F|}{pagesize}$$

where Cl_0 is 0th-edges,
PB is total-frontier-edges,
$newPB$ is the largest $newPB_i$,
F is the largest driver set F_i.

The LMBR for the propagation stage is:

$$\frac{|Cl| + |PB| + 2|newCl|}{pagesize}$$

where $|newCl|$ is the size of largest $newCl$.

Combining the two stages, we get the LMBR for UPFE to be:

$$\frac{max\{(|A| + |Cl_0| + |newPB| + |F|), (|Cl| + 2|newCl|)\} + |PB|}{pagesize}$$

5.6. Discussions on the RRP and LMBR of the Algorithms

The RRP of BWFT is the simplest, and it accesses only three different relations: the base relation A, the driver set F, and the closure Cl. Besides, BWFT has a strong reference locality. It accesses F, then A and then Cl repeatedly inside the loop. This property of strong locality makes higher probability of accessing to data pages that are in the main memory. The RRP of UPFE is also simple, and it is composed of the separate sequences: i) F, A, $newPB$ and PB (in preprocessing phase); ii) Cl, $newCl$ and PB (in propagation stage). However, UPFE needs access to at least 6 binary relations. Therefore, UPFE requires more page accessing than that of BWFT. The RRP of UIEC and UWFE are more complicated and these algorithms access even more intermediate relations. More data (and/or index) accessing means that the page I/O of UPFE, UWFE and UIEC will be more than that of BWFT. It was pointed out in [13] that there is a strong correlation between the disk I/O and page I/O with small buffer size. In fact, the algorithm with the least pages I/O is expected to achieve the best disk I/O performance in small buffer size. Thus, BWFT should have the least disk I/O when buffer size is small.

From previous sections we have the following estimation for the LMBR of the 4 algorithms:

$$\text{BWFT}: \frac{|A| + |Cl| + 2|F|}{pagesize}$$

$$\text{UIEC}: \frac{max\{\ (|A| + |newSub| + |LW| + |W| + |newSub2| + |newSub3|),\ |Cl|\ \} + |SubCl|}{pagesize}$$

$$\text{UWFE}: \frac{|A| + |Cl| + |newPB| + |PB| + 2|newCl|}{pagesize}$$

$$\text{UPFE}: \frac{max\{\ (|A| + |Cl_0| + |newPB|), (|Cl| + 2|newCl|\ \} + |PB|}{pagesize}$$

Note that we have deliberately dropped out the size of the frontier in the formulae of LMBR of UIEC, UWFE and UPFE above. This is because unary frontiers are used in these algorithms, and unary relations are usually small in comparison with the other binary relations. Even so, the comparison on the LMBR of the algorithms is quite difficult. The major difficulty lies in the fact that the sizes of the intermediate relations like *newSub*, *LW*, *newPB* and *newCl* are hard to estimate. Each size depends on the characteristics of the data, and varies from one data set to another. The join and selection selectivities described later in Section 6.1 determine the volume of final results, but they are not sufficient to represent the overall characteristics of the data. However, a rough estimate will be enough for our analysis at this stage.

UIEC has the largest LMBR value. This is mainly due to the storage required for the i-implied-edges-closure *SubCl.* When the final query closure is large, *SubCl,* the superset of the query closure, will become even larger. The LMBR value of UPFE is smaller than that of UWFE, because the query closure of i-frontier-edges is computed after total-frontier-edges is formed, and at that time the base relation *A* can be released to provide space required for the intermediate relations in the propagation stage of UPFE. When the query closure Cl is small[8], the LMBR of UPFE is $\frac{|A| + |Cl_0| + |newPB| + |PB|}{pagesize}$. This also implies that PB will be small. PB can never have tuples other than those in A, and it is also smaller than the query closure Cl. As Cl_0 (0th-edges) which is a subset of Cl, will also be small, thus, the LMBR of UPFE will be smaller than that of BWFT when Cl is small. However, Cl can be very large, and hence, the size of PB can be more than half the size of A. At that time, the LMBR of UPFE is $\frac{|Cl| + 2|newCl| + |PB|}{pagesize}$. This value will be bigger than that of the LMBR of BWFT. To sum up, we state the following:

- BWFT has the least disk I/O when the buffer size is small.
- UIEC has the largest LMBR value.
- UWFE has larger LMBR value than that of UPFE.
- UPFE has a smaller LMBR value than that of BWFT when the size of the final closure Cl is small, but its value will be larger than that of the BWFT when Cl is large.

Recall that when we have limited buffer resources, the algorithm with the smallest LMBR will be a good candidate for query processing. From the above estimation, BWFT has the least LMBR except when the query closure Cl is large, in which case UPFE has the least LMBR. This implies that BWFT will be a good choice for query processing for a wide range of data as it also has a simple RRP. UPFE is good when Cl is not very large. These guidelines provided by RRP and LMBR will be reconfirmed by the simulation studies we have in Chapter 6.

[8] This will happen when selection selectivity or join selectivity is small.

Chapter 6
Simulation Studies

This chapter presents the simulation results using randomly generated data. The size of the generated base relation will be kept to 1000 tuples (binary relation) throughout our simulation. Three parameters affecting the I/O performance will be studied. They are, namely, the buffer size (main memory capacity), the join selectivity and selection selectivity. All algorithms will be run on 5 base relations, generated for each set of the parameter values. The performance of an algorithm is measured by the disk I/O generated by the algorithm, although occasionally page I/O[9] is also used. The average of these runs will be calculated. The results will then be analyzed and interpreted.

6.1. Parameters of the Simulation Model

In our performance studies, we generated some uniformly distributed data to construct the tuples for the base relation on which the binary query closure algorithms are applied. Each tuple of the base relation has two attributes, and the values are generated independently. To control the volume of the final query closure, we have two parameters: Join Selectivity and Selection Selectivity. Let A and B be two relations, J is the result relation of joining A and B, and S is the result relation after selection is done on A. Now the selectivities are defines as follows:

Definition 6-1: Selection Selectivity is defined as the ratio of the size of S to the size of A.

$$SS = \frac{|S|}{|A|}$$

[9] Readers interested in the difference between page I/O and disk I/O as cost metric are referred to [15]

Definition 6-2: Join Selectivity is defined as the ratio of the size of J to the product of the sizes of A and B.

$$JS = \frac{|J|}{|A| \cdot |B|}$$

Assume that the attribute values of both relations A and B are chosen from the same domain D[10]. Let $jattb_A$ and $jattb_B$ denote the join attributes from A and B respectively. Let $sattb_A$ be the selective attribute from A. Without loss of generality, we assume that the domain D consists of positive integers $1, 2, \cdots, |D|$. If the attributes are randomly chosen from the domain D, and c is an integer in domain D, then the probability that a tuple t from A has $sattb_A \leq$ c is:

$$Prob(\text{a tuple } t \text{ from } A \text{ has } sattb_A \leq \text{c}) = \frac{c}{|D|}$$

then the expected number of tuples in A with $sattb_A \leq$ c is:

$$E(\text{number of tuples with } sattb_A \leq \text{c}) = \frac{c}{|D|} \times |A|$$

Hence, the average size of the S, i.e., the result of selection on A with a range of values less than or equal to c is:

$$E(\,|S|\,) = \frac{c}{|D|} \times |A|$$

Therefore,

$$\begin{aligned} SS &= \frac{c\,|A|}{|D|} \;/\; |A| \\ &= \frac{c}{|D|} \end{aligned}$$

For each tuple t_A from A, let its $jattb_A = \text{j}$ for some integer j. Then the probability that a tuple t_B from B has $jattb_B = \text{j}$ is:

$$Prob(\text{a tuple } t_B \text{ from } B \text{ with } jattb_B = \text{j}) = \frac{1}{|D|}$$

This implies that the expected number of tuples in B matches with t_A is:

[10] This is not a necessary condition, but this will simplify a lot of calculations.

$$E(\text{number of matching tuples of } B \text{ with } t_A) \quad = \quad \frac{|B|}{|D|}$$

Thus, the expected number of matching tuples of relation A and B is:

$$E(\text{total number of matching tuples}) \quad = \quad \frac{|B|}{|D|} \times |A|$$

Hence, the expected size of the result relation J of joining A and B is:

$$E(\,|J|\,) \; = \; \frac{|B|}{|D|} \times |A|$$

Therefore,

$$\begin{aligned} JS &= \frac{|A|\cdot|B|}{|D|} \;/\; (|A|\cdot|B|) \\ &= \frac{1}{|D|} \end{aligned}$$

In order words, if we perform selection of A on $sattb_A$ attribute using constant values ≤ c, then we expect the size of the result relation to be $|S| = \frac{c\;|A|}{|D|}$. Besides, the mean size of J is equal to $\frac{|A|\cdot|B|}{|D|}$. In this way, we can have some control regarding the size of the result relations of the operations. Hence, if we use a small constant value c, i.e., SS is small with $|D|$ being kept as constant, then the selection which is performed on the base relation at the beginning of all the binary query algorithms (described in Chapter 2) will produce a small relation. Then the outcome of the query closure will not be large. Moreover, if JS is small, i.e., $|D|$ is large, then the query closure will not be big. On the other hand, if we want to produce large volume of tuples for the query closure, then we have to make JS and SS both large (small $|D|$ and large c). In our simulation, we do not have different join selectivity for each pair of joining relations in the algorithms, not only because it is hard to get the values right, but also because it will complicate the analysis. Instead, we use only one join selectivity and one selection selectivity for each randomly generated base relation. The values we use are:

JS :	0.0001, 0.0005, and 0.001
SS :	0.001, 0.005, 0.05, 0.1, 0.3, and 0.5

In addition to those parameters to control the data volume of the final result, we have two other parameters to control the system performance. These are the page size (the number of the binary tuples that can reside in a system page), and the buffer size (number of pages in the main memory). Since varying one of the parameters will be sufficient to control the capacity of the data in the main memory, we fix the page size parameter to 20 tuples/page throughout our experiments. The buffer size will have values 10, 25, 50, 75, 100, 150 and 200.

6.2. Observations and Interpretations of the Results

In this section, we will present our simulation results using random data. In the previous sections, we have mentioned that we have 3 different values for JS, and 6 different values for SS. For each pair of the values from the combination of the two parameters, we generated 5 sets of 1000-tuple base relation on which query closure is to be found by the 4 algorithms. Each of the test runs will further experiment through the 6 different buffer sizes. Then the average of the 5 runs will be calculated and recorded. The page I/O of the algorithms are plotted and presented in Figures 6-1 to 6-3. The significant results of the disk I/O of the algorithms are plotted and presented in Figures 6-4 to 6-6.

From the graphs in Figure 6-4, Figure 6-5 and Figure 6-6, we have the following observations and interpretation:

Fact 1: For small SS and small JS ($SS \leq 0.005$; $JS \leq 0.0005$), the curves of the 4 algorithms almost coincide with each other. Thus, when the data volume of the query closure produced is very small, there is no difference in choosing any of the 4 algorithms for processing with any buffer size.

Fact 2: The curve of BWFT is always below the other curves when the buffer size is small. When buffer size is large (B.S > 25), the curve of BWFT is not very much higher than the other curves. This implies that BWFT on the average outperforms all other algorithms. Its disk I/O performance is more steady than the others, so that varying the buffer sizes does not cause a dramatically decrease in its disk I/O. When the main memory resource is limited (buffer size is small), it makes BWFT the best candidate for binary query processing.

Fact 3: When SS and JS are not both very large ($SS \leq 0.3$ when $JS \leq 0.0005$; $SS \leq 0.05$ when $JS = 0.001$), the curve of UPFE is below all the other curves for large buffer sizes (B.S. ≥ 75 pages). For large SS (i.e., SS = 0.5), the UPFE curve is above all the other curves when the buffer size is small. This means that UPFE performs as well as BWFT when the

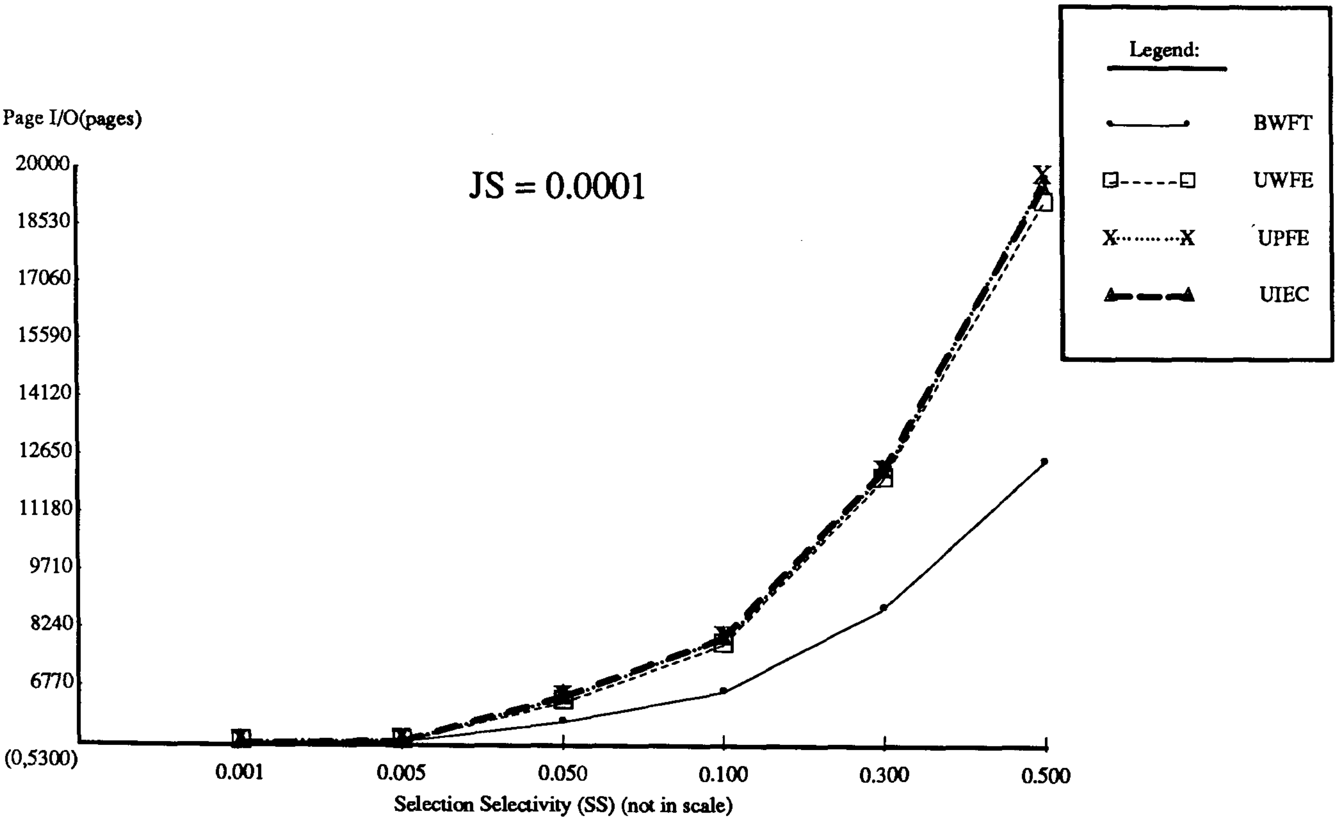


Figure 6-1: Page I/O with JS = 0.0001

buffer size is large, and when the values for SS and JS are not both large. That is when the data volume of the query closure is not large, UPFE is a good choice for query processing for large buffers. UPFE even outperforms BWFT in those situations. However, UPFE has the worst disk I/O performance when the SS value is high with limited buffer resources.

Fact 4: Curve UWFE is above curve UPFE for a large range of data, but it is below curve UPFE when SS and JS are both large (SS=0.5, JS=0.001). Hence, UWFE does not always perform worse than UPFE in its disk I/O performance. When the volume of data handled by the algorithms increases, UWFE does outperform UPFE.

Fact 5: Starting at a very high point when SS or JS is large, the curve of UIEC decreases rapidly with the increase in buffer sizes. When SS and JS are both large (SS=0.5; JS=0.001), UIEC even drops below the curve of BWFT when the buffer size is large (B.S. = 200 pages). Therefore, it seems that UIEC is very sensitive to the changes in the buffer size, especially when the final data volume is large. In the case when SS and JS are large, UIEC will have the best disk I/O performance. Therefore,

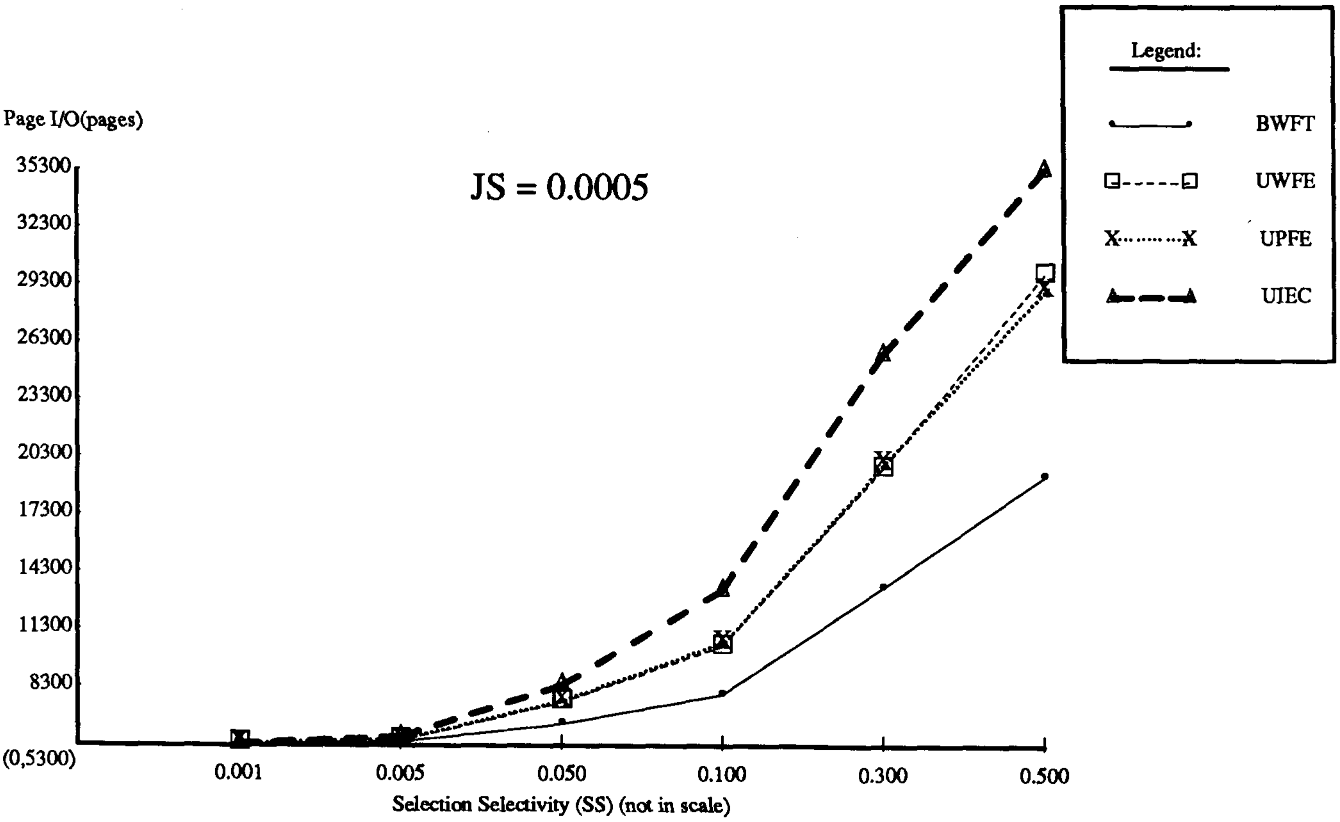


Figure 6-2: Page I/O with JS = 0.0005

UIEC can be chosen for binary query processing to handle large data volume results when the system is provided with rich main memory resource.

6.3. Simulation vs Analytical Results

In this section, we will compare the simulation results with the analysis presented in Chapter 5 which are based on the RRP and LMBR of the algorithms, and the characteristics of the data. We do so by providing an explanation for each of the simulation results stated in the previous section.

Fact 1:

When the values for SS and JS are small, then the data volume for the result closure will be small too. This means that all the algorithms will execute only a few steps before the

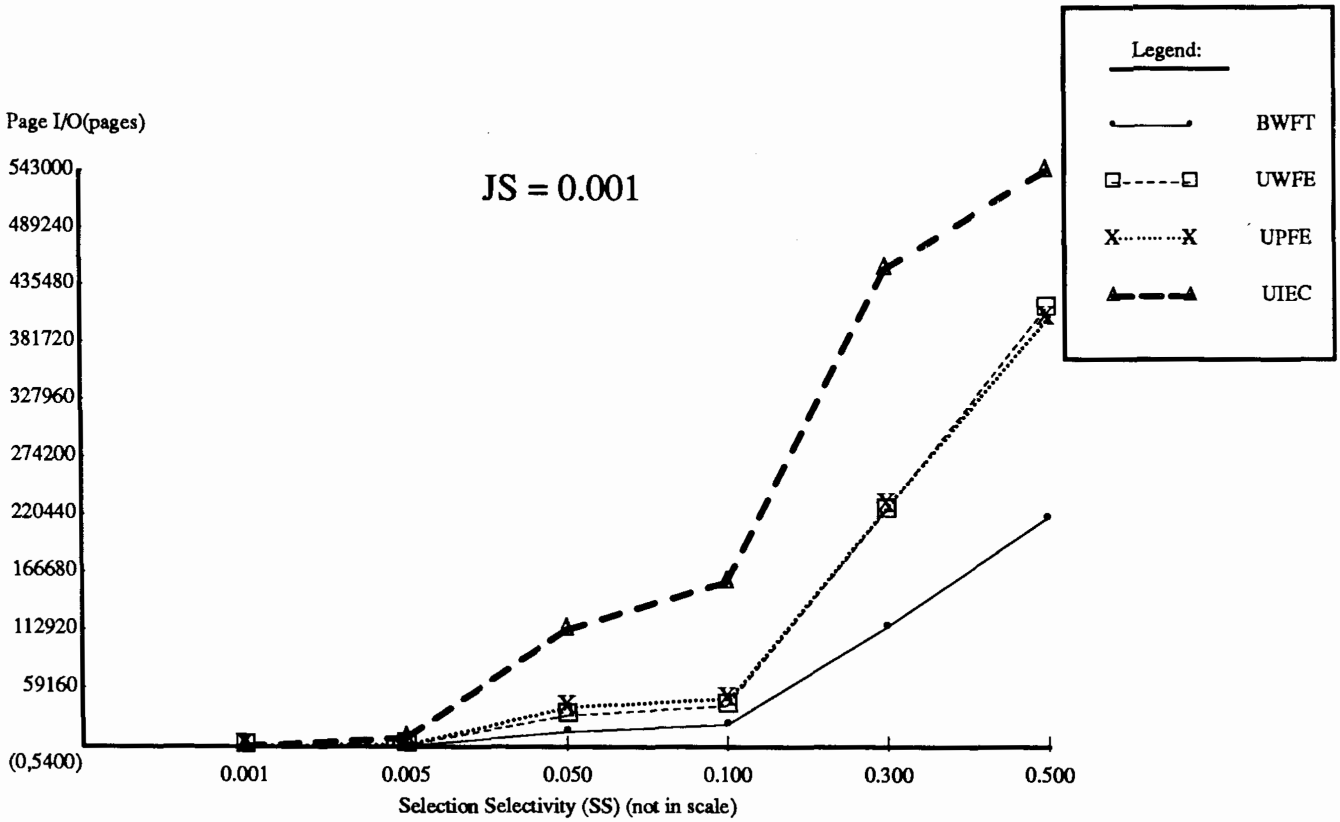


Figure 6-3: Page I/O with JS = 0.001

answer to the query closure is found. Therefore, even algorithms with bad reference locality will produce the answer without lot of disk I/O. Besides, the buffer requirement will be the least, and so the algorithms will achieve their LMBR for small buffer size.

Fact 2:

BWFT has overall the least disk I/O performance. This confirms with our conjecture that simple RRP, high degree of reference locality, and small LMBR is desirable for query processing.

Fact 3 & 4:

When the data volume handled by the algorithms is small (SS ≤ 0.3, JS ≤ 0.0005; SS ≤ 0.05, JS ≤ 0.001), UPFE will have the least LMBR. Thus, increasing the buffer size makes UPFE better chance to have least disk I/O. Therefore, when the buffer size increases to

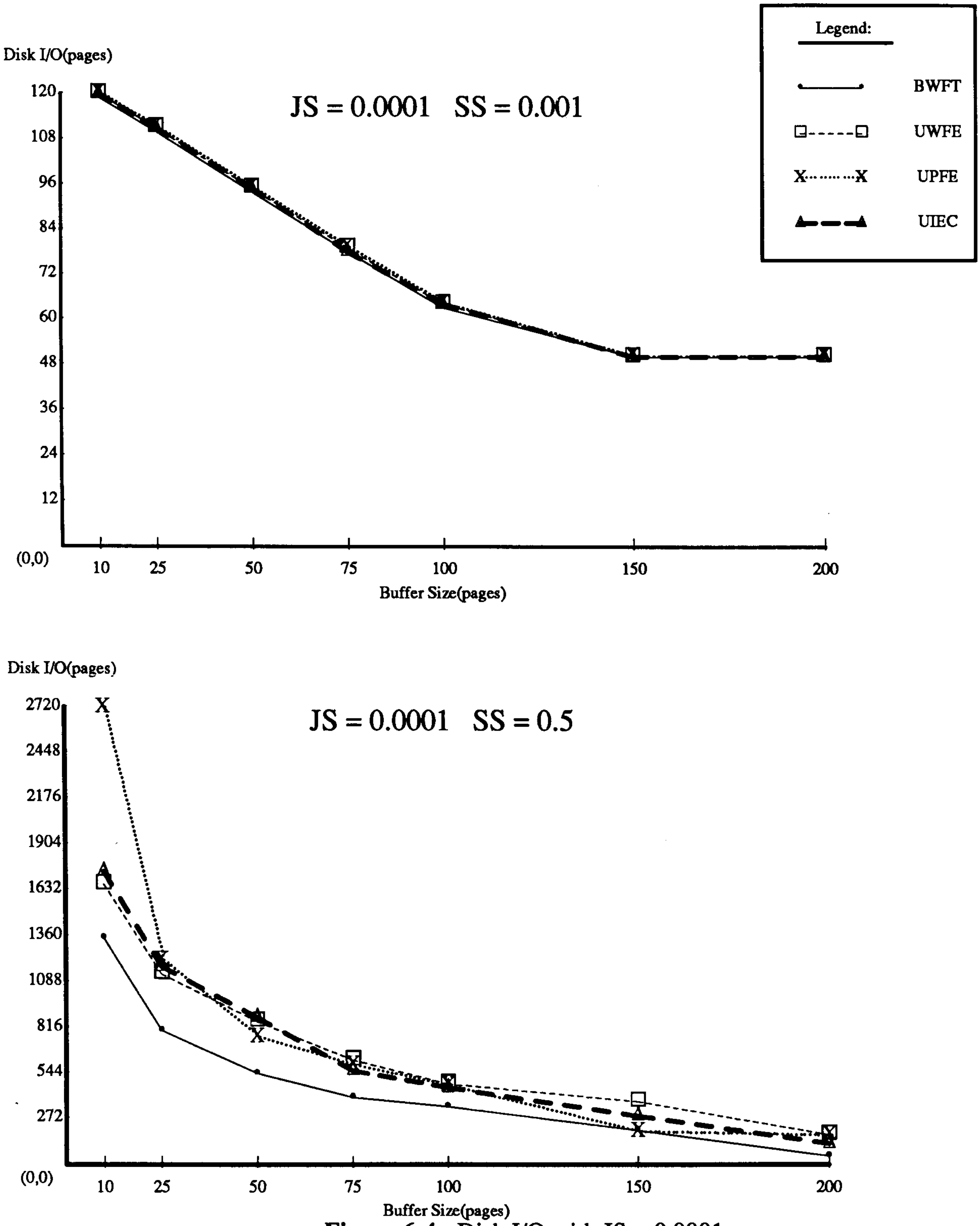


Figure 6-4: Disk I/O with JS = 0.0001

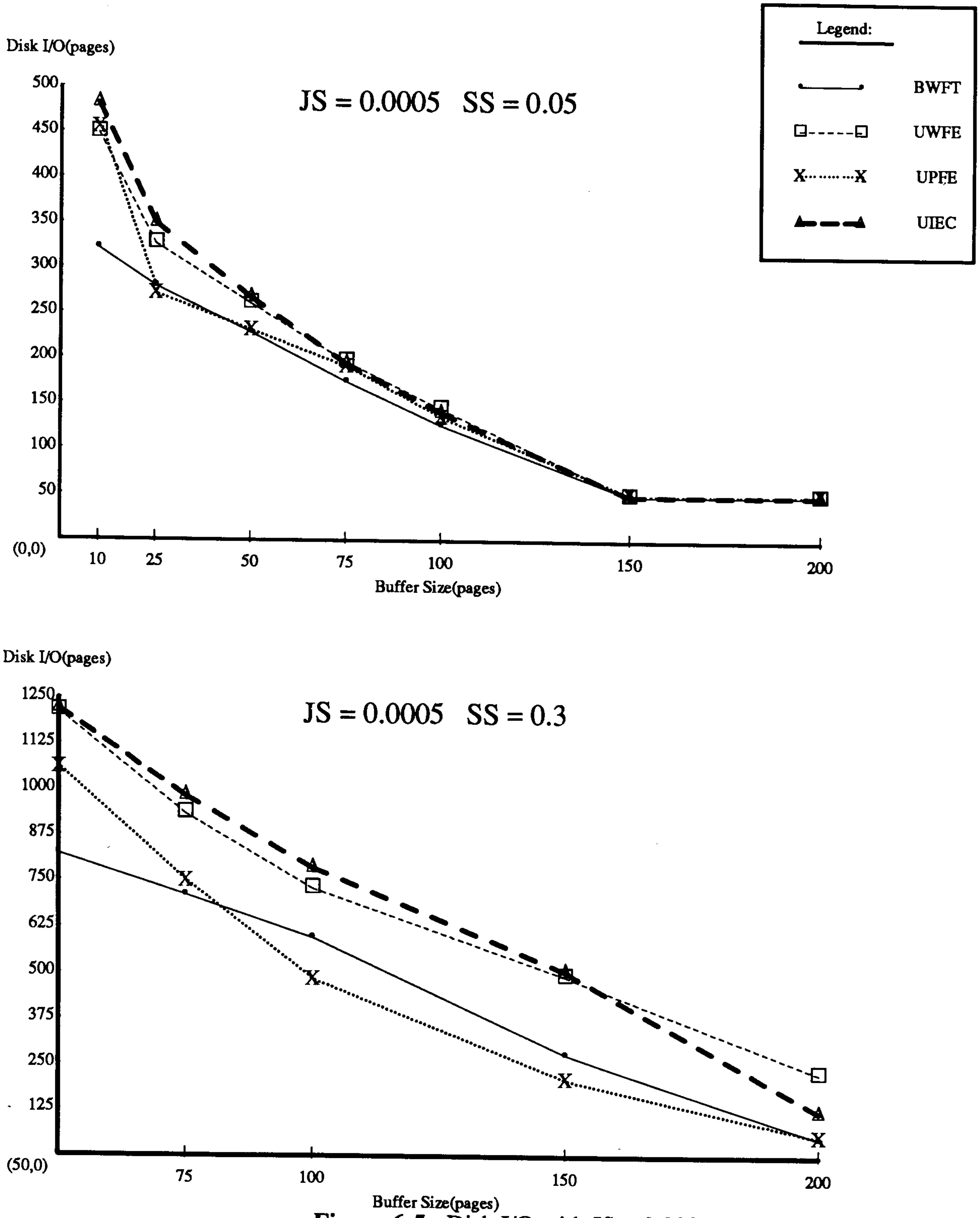
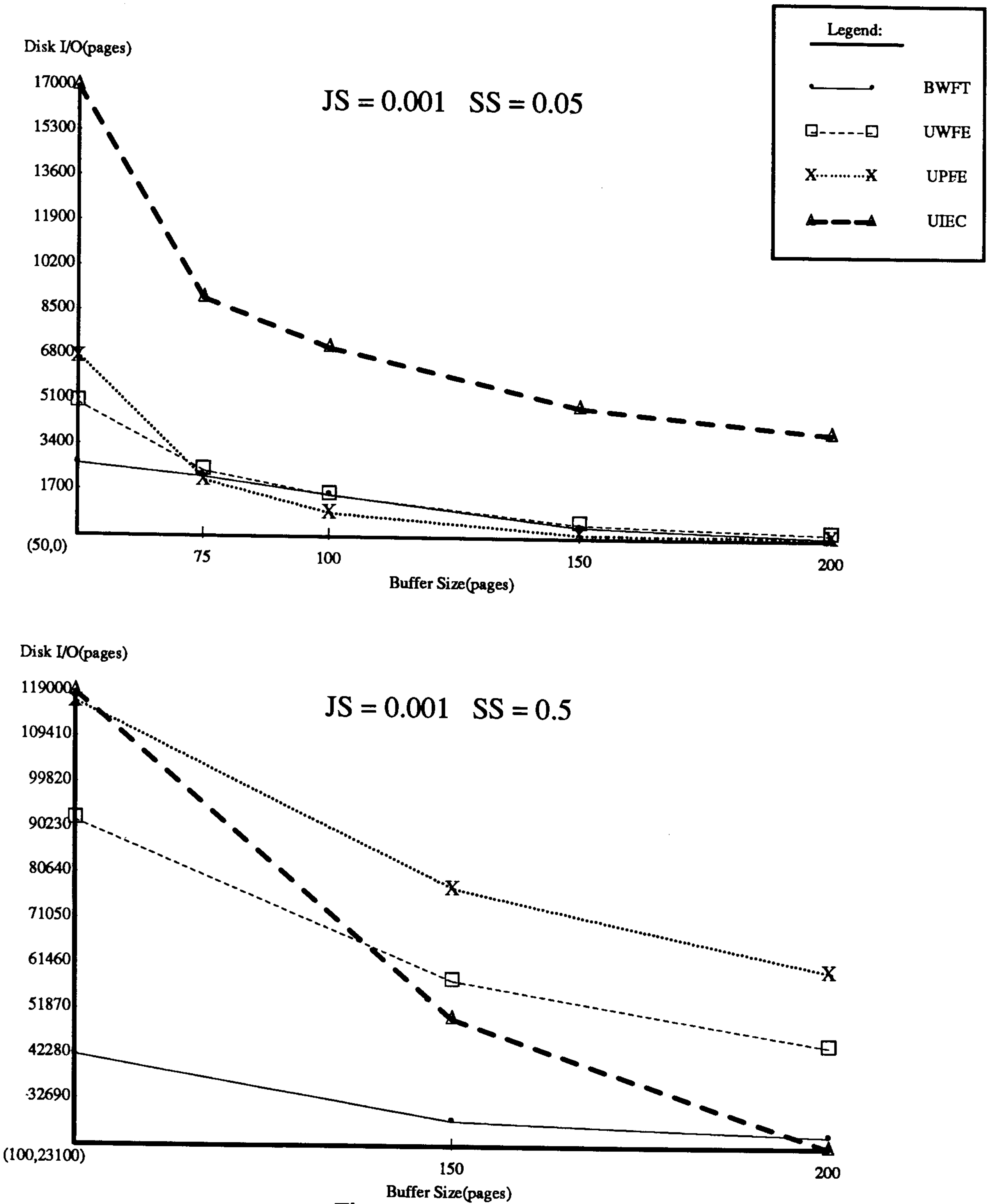


Figure 6-5: Disk I/O with JS = 0.0005



JS = 0.001 SS = 0.05

JS = 0.001 SS = 0.5

Figure 6-6: Disk I/O with JS = 0.001

moderate size (B.S. ≈ 75 pages), UPFE has the best disk I/O performance. However when SS increases (SS = 0.5), total-frontier-edges *PB* of UPFE will be very large[11] In that case, the propagation process of UPFE must start with large total-frontier-edges *PB*, while UWFE starts with a much smaller intermediate relation *newPB*. The data clustering effect with smaller intermediate relation in i-closure updating process of UWFE enhances it to perform better that UPFE.

Fact 5:

When both SS and JS are large (i.e., SS=0.5 and JS=0.001), the data generated will require more iterative processing from the algorithms. That is, the depth of the transitive closure will be large. This kind of data resembles the Sawtooth Data (described in Chapter 3) and is best processed by UIEC. UIEC performs quite poorly when the buffer is very small (B.S. < 100 pages) and improve drastically as the buffer increases to 200 pages. Our explanation for this seemingly strange disk I/O behavior of UIEC is that the page I/O of UIEC is so poor that when the buffer size is small, the page I/O becomes the dominating factor in its disk I/O performance. The Sawtooth data effect becomes apparent only when the buffer size becomes much larger (say 200 pages, which is still only a small fraction of its LMBR).

[11] $|PB|$ is greater than half of $|A|$.

Chapter 7
Level Relaxation

The level relaxation version of the δ-Wavefront Algorithm was developed in [7]. Level relaxation further eliminates redundancy in data accessing. The main feature of the level relaxation algorithm is the extraction of all the drivers at different levels of iteration from a page before the page is swapped back to disk. Therefore, as long as the tuples in the main memory can be accessed by the current drivers, the data page will not be swapped out from the memory. Some disk I/O may be saved by doing this if the main memory contains some of the derivation paths. We have implemented the level relaxation versions of the algorithms. Since UWFE and UIEC have already shown their weaknesses in their disk I/O performance, we will focus on level-relaxed BWFT and level-relaxed UPFE.

7.1. Level-relaxed BWFT (LWFT)

The level relaxation version of BWFT is very similar to the original version. The only difference is that we have provided a set of routines that enable us to achieve a higher level control of page swapping. To facilitate, we have two assumption:

A_1 : *A* is sorted so that all the tuples with the same drivers will be clustered together.

A_2 : The page fetching to the data page of *A* is done in the way that either all the pages corresponding to an active driver (data pages of *A* that contain tuple(s) having the active driver in its (their) first attribute) or none of these pages will be brought into the main memory.

We first consider the algorithm for level-relaxed BWFT. The implementation issues will be addressed later.

```
Cl := $\sigma(A)$
F := Cl
while ( F $\neq \varnothing$ )
   Fetch drivable data pages of A to main memory (MM)
   MF := drivers in F drivable in main memory
   NMF := F - MF
   F := NMF
   while ( MF $\neq \varnothing$ )
       newF := MF $\Delta_{MM}$ A - Cl
       Cl := Cl $\cup$ newF
       MF := drivers in newF are drivable in main memory
       NMF := newF - MF
       F := F $\cup$ NMF
   end while
end while
```

The statement "Fetch drivable data pages of *A* to main memory (MM)" means that we will fetch those data pages of base relation *A* which contain some drivers according to those in the driver set *F*. For example, if in *F* we have $\{(a_1,b_1),(a_2,b_2),\cdots,(a_m,b_m)\}$, the data pages of *A*, with the first attribute column having any values of b_i ($1 \leq i \leq m$), may be fetched. However, sometimes not all these pages will be fetched, since the number of pages fetched can never be more than what the main memory can hold. Thus, unless the main memory has a large buffer (or there are not many of these drivable pages), normally not all the drivable data pages are fetched. In order to optimize further, we can fetch in pages that have more active drivers. Since not all the binary drivers will be used for each page fetching, we divide the driver set *F* into two subsets, *MF* and *NMF*. *MF* (*NMF*) is the driver set which is (not) drivable with respect to the portion of data pages of *A* in MM. *NMF* must be retained in the driver set (*F*) for processing until their corresponding data pages in *A* have been fetched. The operator Δ_{MM} is used for the level-relaxed join operation. Unlike the original join operator (Δ), it will not cause page fetching of the joining relation *A*. It will only consider the portion of *A* in main memory and do ordinary join operation on that portion. New binary drivers will be found and these will also be checked of occurrence against the tuples in *Cl*, as that in the original BWFT algorithm. Then the unique new binary drivers will be divided into two subsets *MF* and *NMF* again. This process of finding new binary drivers according to the current data pages of *A* in main memory will continue until no more active drivers can be found in the data pages of *A* in main memory (i.e. *MF* is empty). At that time, new drivable data pages of *A* will be fetched in. The algorithm will terminate when *F* is empty.

Theorem 7-1: The level-relaxed BWFT will produce the complete query closure in *Cl* upon termination.

Sketched Proof :

In the trivial case when the buffer is large enough to hold all the data pages of *A*, each fetching will enable all the drivable pages to be brought into the main memory. This implies that *MF* must equal the whole set of *F* at the beginning of the loop, and equal *newF* inside the second loop. *NMF* must be empty all the time. Hence, in each iteration of the inner loop, *newF* actually get the same set of tuples as that of the binary frontier in the original BWFT algorithm. The inner loop of the level-relaxed version then resembles the original one. Thus, the same set of *Cl* will be computed. Since *F* is the union of all the *NMF*'s, and so it is empty. When the inner loop termiantes, the outer loop must also terminate. Therefore, the claim is valid.

When only a part of *A* is in main memory, the binary frontier will be divided into *MF* and *NMF* where at least one of the sets will be non-empty if the binary frontier is non-empty. With assumption A_2, if *MF* is not empty, all the binary drivers in *MF* will get the same set of implied edges as that when these binary drivers are joined with the whole relation *A*. Therefore, the drivers in *MF* will derive the same implied edges as before. When *NMF* is not empty, all its binary drivers will be retained. Page fetching of different portion of *A* will enable some elements in *NMF* to become elements in *MF*, and hence allow each of the active drivers to be fully explored. Therefore, each of the binary driver ever derived will get the same set same of implied edges as before. Thus, upon termination, level-relaxed BWFT must produce the correct closure. □

7.2. Level-relaxed UPFE (LPFE)

The algorithm for UPFE has two loops, one for the derivation of total-frontier-edges (preprocessing stage), and the other one for finding the query closure of total-frontier-edges (propagation stage). In the level relaxed version of UPFE, there are also two separate stages, and they are very similar to the original UPFE. In its preprocessing stage, level-relaxed strategy will be used to get total-frontier-edges. The propagation stage of level-relaxed UPFE is just the level-relaxed BWFT in which total-frontier-edges *PB* is used instead of the base relation *A*. The assumptions made in level-relaxed BWFT will also applied here. The derivation stage and the propagation stage of the level-relaxed UPFE are given as follows:

```
/*   Proprocessing Stage */
Cl := $\sigma$(A)
PB := Cl
F := $\Pi_2$(PB) - $\Pi_1$(PB)
while ( F $\neq \emptyset$ )
   Fetch in drivable data pages of A to main memory (MM)
   MF := drivers in F are drivable in main memory
   NMF := F - MF
   F := NMF
   while ( MF $\neq \emptyset$ )
     newPB := MF $\Delta_{MM}$ A
     PB := PB $\cup$ newPB
     newF := $\Pi_2$(newPB) - $\Pi_1$(PB)
     MF := drivers in newF are drivable in main memory
     NMF := F - MF
     F := F $\cup$ NMF
   end while
end while

/*    Propagation Stage    */
F := Cl
while ( F $\neq \emptyset$ )
   Fetch in drivable data pages of PB to main memory (MM)
   MF := drivers in F are drivable in main memory
   NMF := F - MF
   F := NMF
   while ( MCl $\neq \emptyset$ )
     newF := MF $\Delta_{MM}$ PB - Cl
     Cl := Cl $\cup$ newF
     MF := drivers in F are drivable in main memory
     NMF := newF - MF
     F := F $\cup$ NMF
   end while
end while
```

As before, the operator Δ_{MM} is the level-relaxed join which will not cause page fetching. We will not consider the propagation stage as it is just the implementation of the level-relaxed BWFT with *A* changed to *PB*. Here, we will focus on the derivation stage.

Lemma 7-2: The derivation stage of level-relaxed UPFE terminates and get total-frontier-edges *PB*.

Sketched Proof :

When the buffer is large enough to hold the whole data relation *A*, then each time all the drivable data pages can be fetched into the main memory. Thus, *MF* always equal the binary frontier of the original UPFE at each iteration. Hence, the derivation stage of level-relaxed UPFE functions just as in the original one. It will terminate because *NMF* is empty, and hence, when the inner loop terminates, the derivation stage also completed.

If only a fraction of drivable data pages of A can be brought into the main memory each time, because of our assumption in page fetching, each driver in MF will be fully explored before it is deleted. Therefore, we will get all the arcs directing from the drivers in MF. For the drivers in NMF, page fetching enables the data pages that are correspond to the drivers to be brought into the main memory at some time. Thus, these drivers will become element in MF, and hence, they will be fully explored. Therefore, all the drivers will derive the same set of driven elements as that in the original UPFE. Hence, upon termination of the derivation stage, the same total-frontier-edges set as that of the original UPFE will be found in PB. □

Theorem 7-3: Level-relaxed UPFE terminates and computes the query closure correctly.

Sketched Proof : From Lemma 7-2, total-frontier-edges PB must be correctly computed after the derivation stage. Then by Theorem 7-1, and by the fact that the propagation stage of UPFE is the implementation of BWFT on the relation PB, the level-relaxed UPFE must terminate and produce the correct query closure. □

7.3. Experiments with Level-Relaxed Algorithms

Level relaxation requires the control of the fetching and the retention of data pages in the main memory. In our implementation, we adopt the reservation strategy, so that some part of the main memory is for the extensive use of some relations. In algorithms like the level-relaxed BWFT and level-relaxed UPFE, some portion of main memory should be reserved for storing pages of the base relation A. Therefore, in that particular implementation, we have to assign a maximum number of buffer pages, say *numofpage*, reserved by these relations. This implies that while the number of pages of A that can be fetched at any one time can be controlled, the buffer memory for the other relations will be less. This results in more disk I/O for the other relations. Generally, it is very hard to predict the optimal value for *numofpage*, and this is left for further investigation.

In our simulation studies, we have used different values for *numofpage* to reserve different pages, and then look for the best disk I/O performance for the set of parameters used. However, since in our experiments, we also need to vary the buffer size to examine the effect on the performance, instead of assigning some value as the maximum number of reserved pages, we choose to use the ratio of the maximum reserved pages to the size of the main memory (buffer size), *reserve_ratio*. In our experiments, the values for *reserve_ratio* we use are:

0.3, 0.4, 0.5, 0.6, 0.7.

Thus, if the ratio of 0.3 and buffer size of 10 are used, then the maximum number of reserved buffers in main memory is 3 (or 0.3×10). Level-relaxed UPFE also requires reserved memory for total-frontier-edges *PB* which is ready only at the end of the preprocessing stage. Therefore, the memory reserved for *A* can be released at that time and be re-used by *PB*. For each set of data, the two level-relaxed algorithms (i.e. level-relaxed BWFT and level-relaxed UPFE) will be run using different buffer sizes (10, 25, 50, 75, 100, 150, 200). For each buffer size used, the reserved ratio will be varied from 0.3 to 0.7, and the value of the least I/O performance among the ratios will be recorded only for that particular buffer size. Therefore, we are varying the maximum reserved pages in order to find the best possible I/O performance for the buffer size.

In the test runs, we used the same set of random generated data as that in Chapter 6. However, there is not much improvement of the level-relaxed versions as compared to the original versions. It is suggested in [7] that the level-relaxed versions will perform better in clustered data than in uniformly distributed data. Hence, in the section that follows, we will test the level-relaxed versions on clustered data.

7.4. Clustered Data

Clustered data, in our context, refers to the set of tuples of which a driver in a particular set of pages will drive to some other drivers in the same set of pages. Thus, the degree of cluster effect of a data relation is defined as the probability that a driver will find its driven elements in the same page where it resides. In order to generate different clustered base relation, we have the following procedure:

1. Using a random generator, we get one column of attribute values for the set of the tuples from the range $1, \cdots, |D|$ (where $|D|$ is the size of the domain). This column is actually the attribute on which the selection is applied (this is also the column for the drivers).

2. Sort the values, and divide them into pages.

3. For each of the pages, find the minimum and maximum values.

4. For each of the generated attribute value in a page, we generate another value to pair with it to form a tuple. We repeatedly use the random generator at most n times to produce such value in the range $1, \cdots, |D|$, until we find a value between the minimum and maximum value of the page we found before. If after n times, no such value can be found, then we will take the last number generated and pair it with the attribute value in question.

When n is large, there will be a high chance of occurrence of the value of the second attribute in a tuple t_1 in the first attribute of another tuple t_2, where t_1 and t_2 are in the same page. Thus, when n is large, the degree of clustering effect will be high. Hence, we vary the number n to control the degree of clustering for the data. We call this number n **degree of clustering** (DC).

7.4.1. Experimental Results on Clustered Data

In this section, we will present the experimental results and make some observations on them. All the clustered data sets we generated contain 1000 tuples. We use the same technique as described in section 6.1 here to control the data volume. In the analysis that follows, we will use the values of JS and SS to signify the volume of the final results. However, the reader should note that the formulae for JS and SS in section 6.1 are not valid for the join and selection selectivities respectively for the clustered data. In order to calculate the selectivities for clustered data, we have to consider the conditional probabilities given the value for degree of clustering. Nonetheless the interpretation that when JS is large, the size of the result relation of joining the relations is expected to be large, is still valid here. Hence, we will not derive the formula for the selectivities for the clustered data. We will use the same formulae for SS and JS as in Section 6.1 (on page 63).

We produce different clustered data with different values of JS and SS. In all the experiments, we use a value of 20 for DC (degree of clustering) for the clustered data. We do not use different DC because we do not intend to investigate the effect of the comparative performance of the level-relaxed versions and the original algorithms on different degree of clustering.

The results are plotted in Figure 7-1 and Figure 7-2. These results are from clustered data of JS = 0.0005 and SS = 0.5. Since the results from other JS and SS values are quite similar, we do not include them here.

7.4.2. Analysis on Clustered Data

We make the following observations and explanation with reference to the graphs in Figure 7-1 and 7-2:

1. The curve of LWFT (level-relaxed BWFT) is below that of BWFT in the "BWFT vs LWFT" plot in Figure 7-1 for small buffer sizes. When buffer size becomes larger (buffer size = 200), BWFT curve is below the LWFT curve. Thus, the level-relaxed BWFT will outperform BWFT when the buffer size is not large (B.S. ≤ 150 pages), but it will not perform better than BWFT when the buffer size is large. In fact, we expect that level-relaxed BWFT should be better than BWFT in all cases when a clustered data set is used. The fact that level-relaxed BWFT will lose when the buffer size is large is surprising and interesting. The reason is due to the implementation strategy we use for the level relaxation. Recall that we use the reservation method to retain some portion of main memory to hold the base relation. However, reserving part of the main memory to a relation means that the working space for the other relations will be reduced. Thus, level-relaxed BWFT requires more buffer space for operation on the intermediate relations. In other words, the Least Maximum Buffer Requirement (LMBR) of level-relaxed BWFT will be larger than that of BWFT. In Section 6.3, we have already discussed the effect of small LMBR on the algorithm's disk I/O when the buffer is large. Therefore, with smaller LMBR, BWFT should perform better in its disk I/O than level-relaxed BWFT when the buffer size is large.

2. From the plot of "UPFE vs LPFE" in Figure 7-1, the observation as that above can also be seen. The curve of LPFE (level-relaxed UPFE) is also seen to be below that of UPFE when buffer size is small, but will be higher than that of UPFE when the buffer size is large. The same phenomenon being observed indicates that our implementations to the algorithms are consistent. Hence, the level-relaxed version versus its original algorithm should produce the same trend for all the algorithms.

3. The LWFT curve is seen to be below the LPFE curve when the buffer size is small (B.S. ≤ 100 pages), but it will be above LPFE curve when the buffer

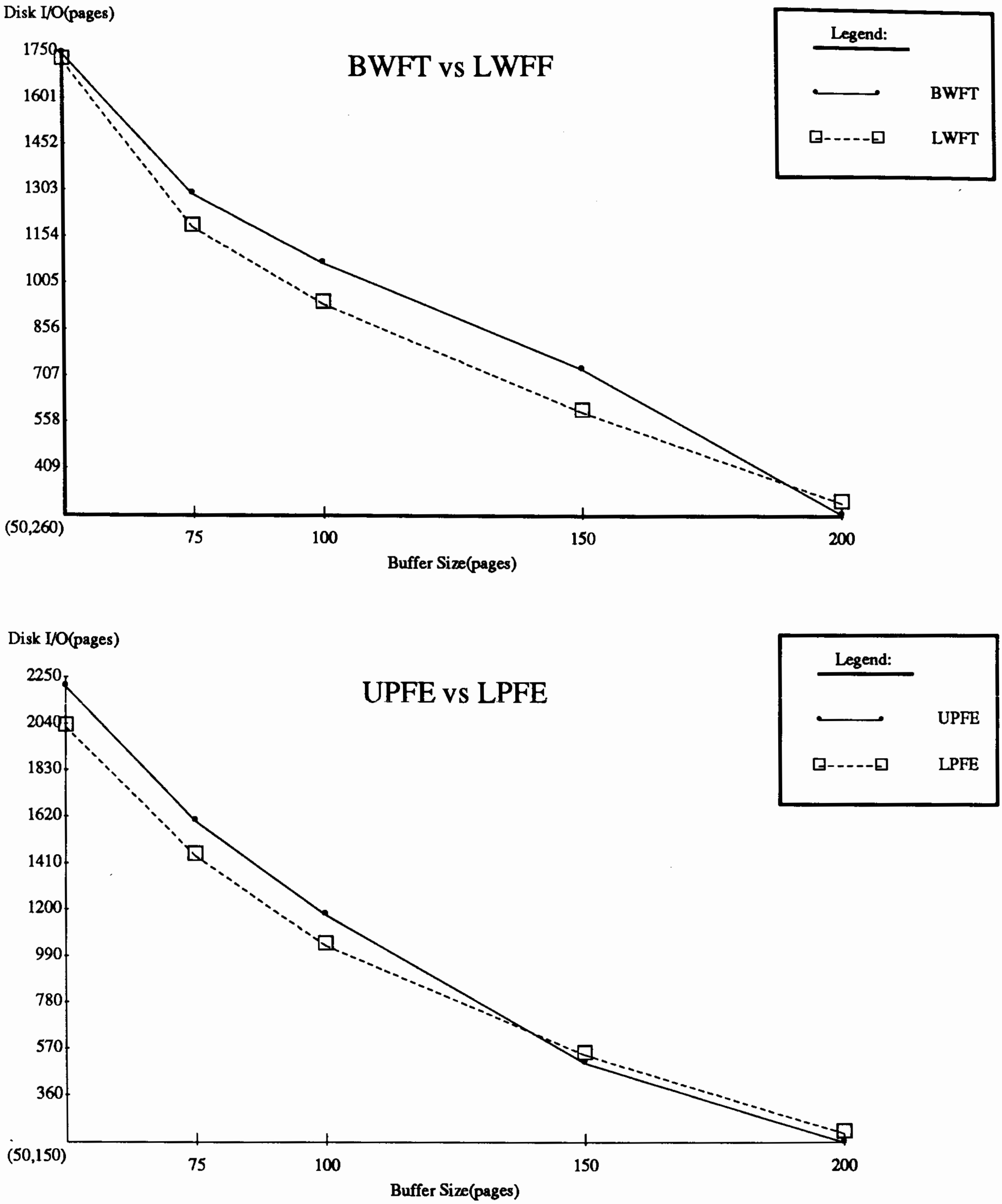


Figure 7-1: Original vs. Level-relaxed Algorithms with JS=0.0005, SS=0.5

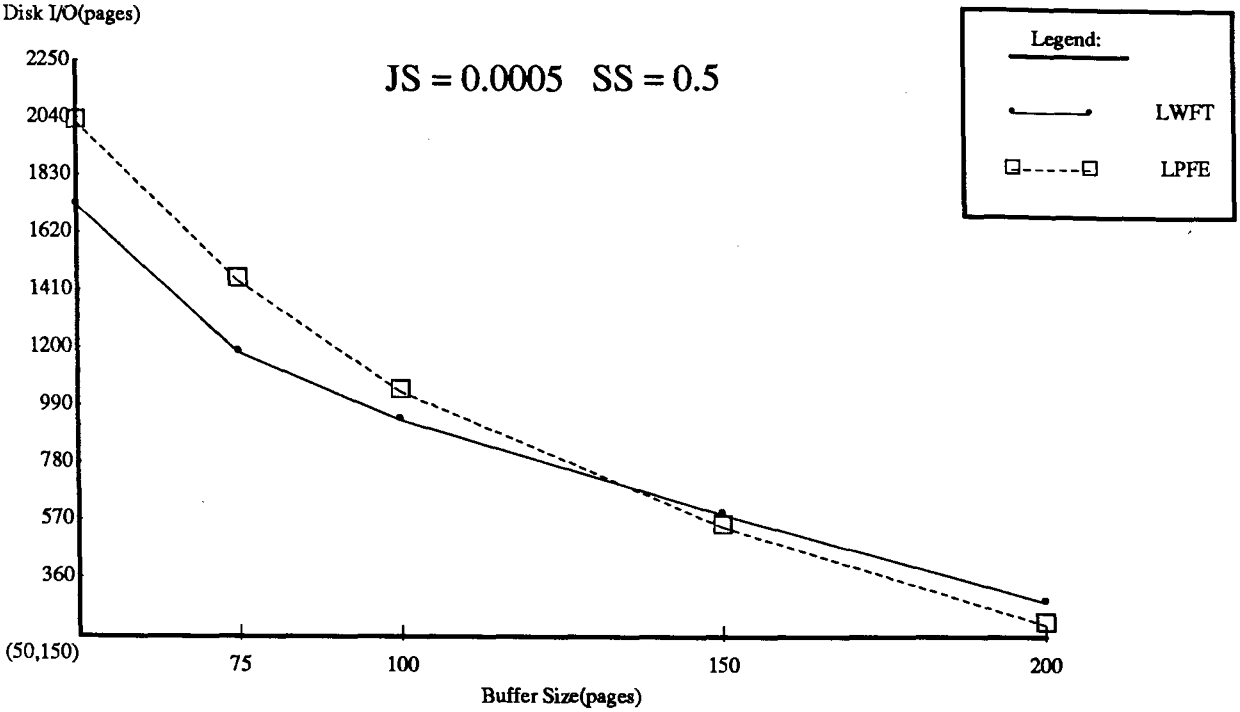


Figure 7-2: Level-relaxed BWFT vs. level-relaxed UPFE

size is large (B.S ≥ 150 pages). In last chapter we concluded that BWFT outperforms UPFE in its disk I/O when the buffer size is small. However, when the data volume of the final results is not large, UPFE will be better than BWFT with large buffer. Thus we can make a similar conclusion regarding the level-relaxed versions of the two algorithms. That is to say, level-relaxed UPFE will be better than level-relaxed BWFT when the data volume of the final results are not large, and when the buffer size is large. This is understandable, because the level-relaxed version should preserve the property of the original algorithm, and thus the comparative performance of the level-relaxed versions should resemble the original ones.

Chapter 8
Conclusions

In this research, we have developed some binary algorithms for query closure processing. These algorithms are compared and analyzed using different characteristics of data on different buffer sizes. From our analytical studies augmented by a simulation study, we have found some interesting characteristics of the algorithms and interesting results:

1. BWFT on the average performs well in its disk I/O for a wide range of data sets and buffer sizes as compared to the other non-level-relaxed algorithms. It will have the least disk I/O when it is run on Parallel-Chains Data.

2. UPFE will outperform the other algorithms when the data volume of the final query closure is not large, and the buffer size is large. If UPFE is run in One-Cycle Data with moderate cycle length, it will outperform the other algorithms in its disk I/O.

3. UIEC has good disk I/O performance as compared to the other algorithms when the base relation requires a lot of iterative processing and generates a large volume of answers to the binary query closure. UIEC performs very well in Sawtooth Data.

4. UWFE will perform better than UPFE when the volume of data handled by the algorithms increases, even though UWFE does not perform as well as UPFE in most cases.

5. Small volume of data can be handled equally well by the four non-level-relaxed algorithms.

6. Level-relaxed algorithms in general have better performance than the non-level-relaxed algorithms though not by much. In fact, level-relaxation does not work too well for large buffer sizes.

Our research on the binary query closure processing ought to stimulate further research on the processing of more complex recursions. During this research, we uncovered some interesting aspects that need further investigation:

- By separating the propagation process from the driver derivation process of UWFE, we obtain the preprocessing algorithm UPFE. Similarly, we can delay the total closure processing in UIEC until all the driver sets have been considered. Thus, we may have a different preprocessing algorithm that has relation reference pattern (RRP) simpler than that of UIEC, and thus may have smaller Least Maximum Buffer Requirement (LMBR) value than that of UIEC. This new preprocessing algorithm may have better disk I/O than that of UIEC. A further studies should be carried out.

- As a preprocessing algorithm, UPFE can be improved when a more efficient method is used in its preprocessing stage. A better method may be one that requires selections and projections on the base relation only to find out all the relevant tuples in the preprocessing stage. More intensive studies are needed.

- BWFT can be modified to incorporate the logarithmic property. To be precise, at each iteration of BWFT, instead of joining the frontier and the base relation, we can join the current closure and the power of the base relation. The power of the base relation is initialized to be the base relation, and it is updated in each iteration by joining to itself. Although we may get larger intermediate relations, the saving in iteration steps may provide a net decrease in the disk I/O. Further studies should lead to conclusive results.

- Since the propagation processes of UPFE and UWFE employ BWFT algorithm, an improvement to BWFT discussed above may lead to an improvement to UPFE and UWFE. Thus, further performance studies are needed.

- We are using the *Least Recently Used Algorithm* (LRU) for choosing a page to be swapped to disk when the buffer is full. The use of LRU faces a lot of criticism on not being able to provide a good buffer management environment, and thus a better replacement algorithm is needed. The effect of choosing different replacement algorithms on the query closure algorithms should be studied more intensively so that a better replacement algorithm can be found.

- Our implementation of the level relaxation algorithms needs further refinement. Better alternatives to the reservation strategy that we used should be investigated.

- The studies of the degree of cluster effect of a binary base relation may provide an insight to the nature of level relaxation strategy. In particular, simulation studies on the performance of the level-relaxed algorithms using different generated data sets by varying the degree of clustering may provide information that can lead to further improvement of the algorithms.

Appendix A
Experiment on Join Operation

In Chapter 6, we described the implementation strategy of join operation. In implementing the nested-loop indexed join method, the operand with the larger size is always chosen for the indexing operation. In this appendix we justify this indexing strategy. We do so by comparing this indexing strategy with an alternative one.

In the experiment, we will join two binary relations R_1 and R_2. The size of relation R_1 is increased, while that of R_2 is fixed. R_2 always contains 1000 tuples. We assume that the index table has been built for R_2, but not for R_1. There are two schemes of the join operation:

1. Scheme I: R_1 is always the outer relation. Indexing is done on R_2. Thus, inside the first loop, the join attribute of a tuple of R_1 will be used as key to access the index table of R_2 to find a matching tuple.

2. Scheme II: This is the indexing strategy we adopted. That is, the outer relation will be the smaller of the relations R_1 and R_2. If R_2 is chosen for the outer relation, then an index table must be built for the larger relation R_1.

The results of the experiment are plotted in Figure A-1. The X-axis represents the size of relation R_1, and the Y-axis represents the disk I/O incurred by the two schemes. In this experiment, we fixed the buffer size to be 10 pages.

From Figure A-1, we can see that when the size of R_1 is less than 1000 tuples, both schemes get the same disk I/O. This is because both schemes choose R_2 for indexing. As the size of R_1 increases, the disk I/O of Scheme II is more than that of Scheme I, indicating that more disk I/O is needed for building the table for R_1. However, when the size of R_1 increases to 5 times that of R_2 (i.e., about 5000 tuples), then Scheme I and Scheme II have

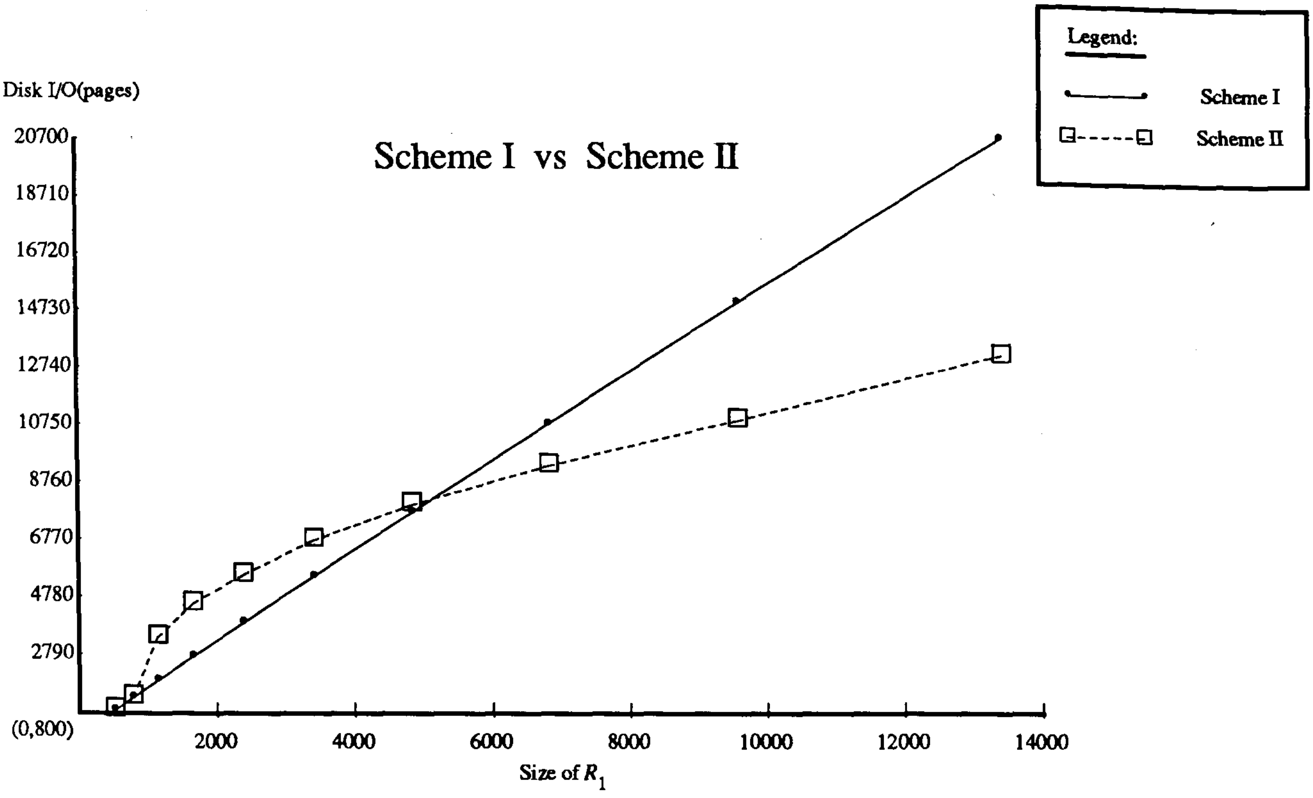


Figure A-1: Disk I/O on different choices of operands for index in join operation

about the same disk I/O performance. When the size of R_1 grows very large, Scheme II performs better than Scheme I.

From this we conclude that Scheme II will be better than Scheme I when one operand of the join operation is substantially larger than the other operand. This implies that although the cost in building the index table is quite high, this cost can be offset by the saving in the cost of sequential access to smaller relation in the outer loop according to the indexing strategy we adopted. In fact, in most cases we encountered in our experiments, the size-increasing relation R_1 is either the relation for i-closure or i-implied-edges-closure. These intermediate relations (i-closure and i-implied-edges-closure) always take part in a union operation after the join operation. Once an index table is built for these relations, the table will be updated in the union operation with much less effort than building the whole table. Hence, the cost of the disk I/O should be much lower than the one we presented here. The indexing strategy that we adopted is thus justified.

References

1. F. Bancilhon. Naive Evaluation of Recursively Defined Relations. In M. Brodie and J. Mylopoulos, Ed., *On Knowledge Base Management Systems*, Springer-Verlag, 1986, pp. 165-178.

2. F. Bancilhon an R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. Proc. of 1986 ACM-SIGMOD International Conference on Management of Data, Washington, DC, May, 1986, pp. 16-52.

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

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

5. J. Han and L. J. Henschen. Compiling and Processing Transitive Closure Queries in Relational Database Systems. Tech. Rept. EECS Tech. Rep. 86-06-DBM-02, Northwestern Univ., June, 1986.

6. J. Han. Selection of Processing Strategies for Different Recursive Queries. Proc. of the 3rd International Conference on Data and Knowledge Bases, Jerusalem, Israel, June, 1988, pp. 59-68.

7. J. Han, G. Qadah and C. Chaou. The Processing and Evaluation of Transitive Closure Queries. Proc. of the International Conference on Extending Database Technology (EDBT'88), Venice, Italy, March, 1988, pp. 49-75. [Lecture Notes in Computer Science 303, Springer-Verlag, 1988].

8. L. J. Henschen and S. Naqvi. "On Compiling Queries in Recursive First-Order Databases". *J ACM 31*, 1 (1984), pp. 47-85.

9. Y. E. Ioannidis. On the Computation of the Transitive Closure of Relational Operators. Proc. of the 12th Int'l Conf. Very Large Data Bases, Kyoto, Japan, August, 1986, pp. 403-411.

10. Y. Ioannidis and R. Ramakrishnan. Efficient Transitive Closure Algorithms. 14th International Conference on Very Large Data Bases, Los Angeles, USA, August, 1988, pp. 382-394.

11. H. V. Jagadish, R. Agrawal and L. Ness. A Study of Transitive Closure as a Recursion Mechanism. Proc. of 1987 ACM-SIGMOD Conference on Management of Data, San Fransisco, California, May, 1987, pp. 331-344.

12. H. Lu. New Strategies for Computing the Transitive Closure of a Database Relation. Proc. of the 13th International Conference on Very Large Data Bases, Brighton, England, September, 1987, pp. 267-274.

13. W. S. Luk, H. M. Mok. Disk I/O Performance of Some Linear Recursive Query Processing Algorithms. Submitted for publishcation.

14. J. Minker. "Search Strategy and Selection Function for an Inferential Relational System". *ACM Transactions on Database Systems 3*, 1 (1978), pp. 1-31.

15. H. M. Mok. Disk I/O Performance of Linear Recursive Query Processing. Master Th., School of Computer Science, Simon Fraser University,July 1987.

16. J. F. Naughton. One-Sided Recursions. Proc. of the 6th ACM Symposium on Principles of Database Systems, San Diego, CA, March, 1987, pp. 340-348.

17. R. Reiter. Deductive Question-Answering on Relational Databases. Logic and Data Bases, Plenum, New York, 1978, pp. 149-178.

18. J. D. Ullman. Database Theory: Past and Future. Proc. of the 6th ACM Symposium on Principles of Database Systems, San Diego, CA, March, 1987, pp. 1-10.

19. P. Valduriez and H. Boral. Evaluation of Recursive Queries Using Join Indices. Proc. of the 1st International Conference on Expert Database Systems, Charleston, South Carolina, April, 1986, pp. 271-293.

20. H. S. Warren. A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations. Communications of ACM 18, April, 1975, pp. 218-220.

21. S. Warshall. "A Theorem on Boolean Matrices". *J. ACM 9*, 1 (January 1962), pp. 11-12.