PERFORMANCE EVALUATION OF
OUTER JOIN OPERATIONS
ON ADDS SYSTEM

By

WEN FONG LEE

Bachelor of Science in Arts & Sciences

Oklahoma State University

Stillwater, Oklahoma

1983

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE
May, 1987

PERFORMANCE EVALUATION OF
OUTER JOIN OPERATIONS
ON ADDS SYSTEM

Thesis Approved:

Yuri Breitbart
Thesis Adviser

D. E. Hedrick

Donald D. Fisher

Norman N. Durham
Dean of the Graduate College

PREFACE

This paper describes the performance evaluation of an outerjoin operation on the ADDS system. It includes the definition of outerjoin, the algorithms used, the test results, and the recommendation of the evaluation.

I would like to express sincere gratitude to my major adviser, Dr. Yuri Breitbart, for his guidance, motivation, and invaluable help. I am also thankful to Dr. Donald D. Fisher and Dr. Michael J. Folk, for their insightful suggestions and encouragement during the course of this work. An extra thank you must go to Dr. G. E. Hedrick for agreeing to serve on my committee as a last minute substitute.

Special thanks are due to Amoco Research Center and its library, for allowing me to do my research and use their facilities during the course of this work.

My deepest gratitude to my wife Helen for her encouragement, for her love, and for just being there.

TABLE OF CONTENTS

Chapter page

Chapter page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

1.1 Background Information

In today's business environment, the ability to share and integrate data so as to provide useful information plays an important part in the decision making process. This ability is made possible by a relational database management system. One of the important functions of a relational database management system is to perform relational operations. The performance of the relational database management system is determined by the amount of time used to generate information. At present, there is no commercially available relational database system that provides outerjoin relational function (in a single step). The main objective of this thesis is to answer complex outerjoin queries, expressed in a nonprocedural language on a distributed database system, with better performance than the conventional database system.

JOIN is a relational operation which is used to merge two or more relations to form a bigger relation based on some conditions or restrictions. In order to integrate the data from pieces of distinct data files into a single data file one needs to use the JOIN operation. However, one should be careful in using the operation. Under some circumstances this operation can provide less information, that an "unmatched tuple" in the relations to be joined may not participate in the result of the JOIN. In other words, if a tuple in one of the original relations does not match any tuples in the other (under the join-defining predicate), then that tuple will not appear in the join result [12]. Therefore, some information will be lost, and maybe impossible to reverse the operation to produce the original relations before the JOIN operation. For example, suppose we have two relations, PARTS and SUPPLIERS.

PARTS

p#	pname	s#
p1	Gear	s1
p2	Nut	s2
p3	Bolt	s4

SUPPLIERS

s#	sname	city
s1	Ajax	London
s2	Acme	Paris
s3	Ace	Rome

Note: PARTS table has part p3 whose supplier number s4 is not in SUPPLIERS table, and the SUPPLIERS table has a supplier s3 which does not appear in the PARTS table.

A join of the two tables, PARTS and SUPPLIERS shown above, where PARTS.s# = SUPPLIERS.s#, would result in the following table.

p#	pname	PARTS.s#	SUPPLIERS.s#	sname	city
p1	Gear	s1	s1	Ajax	London
p2	Nut	s2	s2	Acme	Paris

Projecting this table on Parts and Suppliers respectively, one will not obtain the original tables. Therefore, some data on Parts and Suppliers was lost while performing Join operation.

A special kind of join called outerjoin, by contrast, does not lose such information. The outerjoin operation appends special additional tuples to the result of the corresponding join operation. "There is one such additional tuple in each of the original relations; it consists of a copy of that unmatched tuple, extended with null values in the other attribute positions" [12].

The outerjoin operation was first introduced by Heath [14], and has been formally defined by Lacroix and Pirotte [21], Codd [10], Rosenthal and Reiner [24], and Date [12]. Proposals for supporting outerjoin in SQL/DS were presented in [7] and [12].

The outerjoin operation introduces null values (denoted by "?") in the join of the two relations which are supposed to contain no null value. If they do contain null values, then the joins will be based on the logic rules given in Chapter 2.

Outerjoin is sometimes referred to as theta outerjoin. Theta (θ) denotes one of the comparison operators $=$, $\neq$, $>$,

>=, <, and <=. In this research, outerjoin with the "=" operator is called outer-equal-join, and the rest are simply referred to as outer-theta-join or outer-θ-join.

An example of outer-equal-join of the PARTS and SUPPLIERS tables shown earlier, where PARTS.s# = SUPPLIERS.s#, would result in the following table.

p#	pname	PARTS.s#	SUPPLIERS.s#	sname	city
p1	Gear	s1	s1	Ajax	London
p2	Nut	s2	s2	Acme	Paris
p3	Bolt	s4	?	?	?
?	?	?	s3	Ace	Rome

1.2 Approaches

A few algorithms for the join operation have been presented and discussed in [3], [4], and [28]. Since the join operation is closely related to the outerjoin operation, we will be using the join algorithms [28] available to design and implement the outerjoin algorithms.

The easiest and best known join method is the nested loop algorithm [28] which, without indexing, has an execution time proportional to n**2 for relations of cardinality n [28]. Another popular join method, based on sorting and merging, can reduce this time to a*n*log n, where a is a constant [3]. A better join method, based on hashing [2], [28], can further reduce the time to b*n, where b is also a constant. However, this last method allows the performance of semijoins only [28]. Semijoin is not directly applicable in an outerjoin operation due to the way outerjoin is defined (definition in Chapter 2). In this paper, we will not analyze the semijoin algorithms to perform outerjoin.

1.3 Scope and Limitations

The primary scope of this research is to examine and evaluate nested loop and sort/merge method of outerjoin algorithms. We use ADDS [5] to conduct performance evaluation of these methods. All programs are written in the PL/I language and on IBM 3090 VM machine. ADDS data structures will be used in all routines. We present the performance evaluation results and their analysis for the following test cases:

(1) outer equal-join with one restriction;
(2) outer-theta-join with one restriction;
(3) outer equal-join with multiple restrictions;
(4) outer theta-join with multiple restrictions.

The performance evaluation is based on the following characteristics:

- total C.P.U. time.
- total I/O time.
- total number of comparisons.
- storage requirements.

As a result of our analysis, we conclude that neither the nested loop nor the sort/merge outerjoin method is the best algorithm for the outer-theta-join operations. The choice of the algorithm depends on (1) join attributes (unique or non-unique), (2) join condition(s), (3) the number of resulted join tuples, if one can predict, and (4) the size of the relations.

The thesis contains 6 chapters: Chapter 1 introduces the concept and background information of outerjoin; Chapter 2 presents the definitions and terminologies used in this paper; Chapter 3 discusses the nested loop outerjoin method; Chapter 4 explains the sort/merge outerjoin method; and Chapter 5 analyzes the results; and Chapter 6 summarizes and present the conclusion of the thesis.

CHAPTER II

DEFINITIONS AND TERMINOLOGY

In this chapter, we define outerjoin operation and illustrate the definition with examples. Before we define Outerjoin, we will present the relational structure terminology and the concept of 'null' values. The assumptions and the technical terms are discussed as appropriate.

2.1 Relational Structure Terminology

A domain is a set of values of similar type: for example, all possible part serial numbers for a given inventory. A domain is simple if all its values are atomic (nondecomposable by the database management system) [10].

Let $D_1,D_2,.....,D_n$ be n ($n > 0$) domains (not necessarily distinct). The cartesian product $\times \{D_i: i = 1,2,....,n\}$ is the set of all n-tuples $<t_1,t_2,...,t_n>$ such that t_1 belongs to D_1, t_2 belongs to D_2,....,t_n belongs to D_n. A relation R is defined on these n domains if it is a subset of this cartesian product. Such a relation is said to be of degree n [10].

In place of the index set (1,2,....,n) we may use any unordered set, provided we associate with each tuple component not only its domain, but also its distinct index, which we shall call its attribute. An attribute represents the use of a domain within a relation. That is, n distinct attributes of a relation of degree n distinguish the n different uses of the domains upon which the relation is defined (the number of distinct domains may be less than n). A tuple then becomes a set of pairs(A,ν), where A is an attribute and ν is a value drawn from the domain of A, instead of a sequence $<\nu_1,\nu_2,\ldots,\nu_n>$ [10].

A relation consists of a set of tuples, each tuple having the same set of attributes. If the domains are simple then such a relation will have the following properties [10]:

(1) there is no duplication of rows(tuples);
(2) the row order is insignificant;
(3) the column (attribute) order is insignificant;
(4) all table entries are atomic values.

The extended cartesian product of two relations S and P, $S \times P$, is the set of all tuples t such that t is the concatenation of a tuple s belonging to S and a tuple p belonging to P. The concatenation of a tuple $s = (s_1,\ldots,s_m)$ and a tuple $p = (p_{m+1},\ldots,p_{m+n})$-in that order-is the tuple $t=(s_1,\ldots,s_m,p_{m+1},\ldots,p_{m+n})$.

2.2 Null value

'Null' is a special value indicating that data is missing or not applicable [7]. The null value is outside the normal range of values for its column (i.e., it is not the same as any valid number of string). Whenever a null data value participates in an arithmetic operation (+,-,*,/), the result is the null value. Whenever a null value participates in a comparison predicate with any value (including another null value), the truth value of the predicate is "unknown" (represented by "?"). If a predicate whose value is "?" participates in a boolean expression, the following 3-values logic truth table applied.

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

If the WHERE-clause of a query, applied to a row of a table or join, evaluate to the "?" truth-value, the WHERE-clause is treated as FALSE (i.e., is not true) in this paper. The unary operator does not have any effect on the null value (i.e., if x is null, then +x and -x are also considered to be null). Thus, if an employee has the null value for salary, that employee is not selected by any of the following search conditions:

```
WHERE SALARY > 1000

WHERE SALARY < 1000

WHERE SALARY = 1000

WHERE NOT (SALARY = 1000)
```

There is no consistency as to whether the rows with null values in the join-columns should participate in the join operation [7] and the duplicate joined all-null tuples should be eliminated [12]. As for this paper, the null values in the join-columns will participate in the join operation. Null tuples are treated as normal tuple with null values and the duplicate joined all-null tuples will not be eliminated.

2.3 Outer-join Definitions

To define outer-join, let us assume two relations R1(A,B1) and R2(B2,C) with attributes R1.A, R1.B1, R2.B2, R2.C. For simplicity we assume that the left to right order of attributes within a relation is significant. Assume that R1.B1 and R2.B2 may validly be compared with each other. Let theta denote any one of the operators =, ≠, <, <=, >, >=, that applies to R1.B1 and R2.B2. Define J to be the theta-join of R1 on B1 with R2 on B2;

$$J = R1 \{ B1 \text{ theta } B2 \} R2$$

We assume that the attributes of J inherit their names from the corresponding attributes of R1 and R2; i.e., the attributes of J are A, B1, B2, and C. And we also assume that these names are all distinct. Define ~R1 as follows:

~R1 = R1 - J {A,B1}

Where J{A,B1} is the projection of J on A and B1, and "-" is the set's difference operator. ~R1 is thus the set of tuples of R1 not appearing in the projection of J on (A,B1), the set of "unmatched" tuples of R1, with respect to the join J. Similarly, define ~R2 as follows:

~R2 = R2 - J {B2,C}

Then the outer-theta-join of R1 on B1 with R2 on B2, written

OJOIN * (R1, R2) WHERE R1.B1 theta R2.B2

is defined to be equal to the expression

J union (~R1 × (?,?)) union ((?,?) × ~R2)

where "?" denotes the null value, as before, and "×" denotes the extended cartesian product.

There are also left and right outer-theta-joins. The left outer-theta-join of R1 on B1 with R2 on B2 is defined as:

J union (~R1 × (?,?))

Similarly, the right outer-theta-join of R1 on B1 with R2 on B2 is defined as:

J union ((?,?) × ~R2)

In this paper, if theta is equality, we normally refer to the outerjoin operation as outer-equal-join. Otherwise, we refer to the outerjoin operation as outer-theta-join.

Example 1

Consider the following database in which relation S represents suppliers and relation P represents parts

S (S#, CITY)

P (P#, CITY)

Sample values:

S

S#	CITY
S1	London
S2	Paris
S3	?
S4	NY
S5	SFO

P

P#	CITY
P1	London
P2	Oslo
P3	?
P4	NY
P5	LA

The outer-equal-join of these two relations on S.CITY and P.CITY i.e., the relation

OJOIN * (S, P) WHERE S.CITY = P.CITY

produces the following relation, called it SXP.

SXP

S#	S.CITY	P#	P.CITY
S1	London	P1	London
S2	Paris	?	?
S3	?	?	?
S4	NY	P4	NY
S5	SFO	?	?

?	?	P2	Olso
?	?	P3	?
?	?	P5	LA

Tuples 3 and 7 show that a null value is not equal to a null value. Using the same query and relations in example 1, the results for left and right outer-theta-join are as follow:

SXP

S#	S.CITY	P#	P.CITY
S1	London	P1	London
S2	Paris	?	?
S3	?	?	?
S4	NY	P4	NY
S5	SFO	?	?

(left outer-theta-join)

SXP

S#	S.CITY	P#	P.CITY
S1	London	P1	London
S2	Paris	?	?
S3	?	?	?
S4	NY	P4	NY
S5	SFO	?	?
?	?	P2	Olso
?	?	P3	?
?	?	P5	LA

(right outer-theta-join)

CHAPTER III

NESTED LOOP METHOD

The simplest way to implement an outerjoin operation is by using the nested loop algorithm. This algorithm is considered to be the most inefficient uniprocessor join algorithm by [28], but it is well suited for parallel execution [28]. The parallel execution of the nested loop algorithm described and evaluated in [28] is not used to evaluate the outerjoin operation. But the idea of parallel execution of nested loop algorithm on a uniprocessor is used. In parallel execution with P processors, each having (b+1) pages of local memory, the smaller relation is chosen as the external one (i.e. outer relation) and is sequentially distributed among P processors in blocks of (b-1) pages. Then, the second (internal) relation is broadcasted page by page to P processors. Therefore, each processor joins each (b-1) page block of the external relation with the entire internal relation [28].

For a uniprocessor, we can read in the (b-1) pages of the outer relation into the main memory. Then the internal relation is read a page at a time to perform join operation on tuples from this page with each of the tuples from the outer relation in memory. The process continues until all the pages in the outer relation have been read. In this case, instead of having P processors, it is like having (b-1) processors; and the number of pages is equal to the outer relation divided by (b-1) pages, which is even smaller than the number of distributed pages using parallel processing method assuming that (b-1) is greater than P.

3.1 Nested Loop

First, we look at a simple nested loop algorithm for the outerjoin operation.

Algorithm 3.1 is a simple nested loop Outerjoin algorithm. It means that, for every tuple (tuple1) reads in from file1 (outer loop), all tuples from file2 (inner loop) are read. If tuple1 does not match any of the tuples(tuple2) in file2, then tuple1 is joined with a null tuple of tuple2. A null tuple is a tuple with null values (described in Chapter 2) for its attributes. When a tuple in file2 matches a tuple from file1, that tuple2 is then marked used. After all the tuples (tuple1) from file1 are compared with all tuples (tuple2) in file2, file2 is scanned through one more time to pick up all the unmatched (or unmarked used) tuples (tuple2) and join each of them with a null tuple of file1.

```
/*  file1  - contains tuples from outer relation
 *  file2  - contains tuples from inner relation
 *  file3  - output relation
 *  tuple1 - tuple from file1 or outer relation
 *  tuple2 - tuple from file2 or inner relation
 *  WHERE  - function to evaluate where clause
 */

   OPEN FILE(file1) INPUT;
   READ FILE(file1) INTO(tuple1);
   DO WHILE( NOT eof1);
      tuple1_used_at_least_once = false;
      OPEN FILE(file2) INPUT;
      READ FILE(file2) INTO(tuple2);
      DO WHILE( NOT eof2 );
         IF WHERE(predicates) THEN DO;
            tuple1_used_at_least_once = true;
            WRITE FILE(file3) FROM(tuple1||tuple2);
            mark tuple2 used in file2;
         END;
         READ FILE(file2) INTO(tuple2);
      END;
      IF not tuple1_used_at_least_once THEN
         WRITE FILE(file3) FROM(tuple1||nulls2);
      READ FILE(file1) INTO(tuple1);
   END;
   OPEN FILE(file2) INPUT;
   eof2 = FALSE;
   READ FILE(file2) INTO(tuple2);
   DO WHILE( NOT eof2 );
      IF tuple2 did not mark used THEN
         WRITE FILE(file3) FROM(nulls1||tuple2);
      READ FILE(file2) INTO(tuple2);
   END;
```

ALGORITHM 3.1 - NESTED LOOP OUTERJOIN

To compute the cost of algorithm 3.1, we assume N number of tuples in file1 and M number of tuples in file2. Other notations use in computing the costs in this paper are as follow:

I - I/O time per tuple.

E - Execution time per "Where clause" evaluation (or per comparison).

O - Total I/O time for outputs.

Δ - Other overhead costs.

$$\text{Cost} = (N * I) + (N * M * I) + (N * M * E) + (M * I) + O + \Delta ; \tag{3.1}$$

The two significant variables are the input time and the "where clause" evaluation time. The total I/O time for output is not significant because the total number of tuples written out is fixed no matter what methods you used. Therefore the I/O time for output cannot be reduced. Hence, what is left for improvements are the I/O time for inputs and "where clause" evaluation time. For the next few sections, we try to minimize the number of inputs and the number of "where clause" evaluations which in turn reduce the I/O time and evaluation time respectively.

Cost (3.1) for algorithm 3.1 is easily reduced to

$$\text{Cost} = (N * I) + (N * M * I) + (N * M * E) + O + \Delta ; \tag{3.2}$$

by eliminating the last scan through file2. Then, we have to introduce a test to capture all the unmatched tuples (tuple2) in file2. The way to do the testing is to check for unmatched and unmarked used tuples (tuple2) in the last pass (for the last tuple1), and join them with the null tuple of tuple1. See Appendix K for the algorithm.

3.1.1 NESTED LOOP USING BLOCK FACTOR

In algorithm 3.1, the tuples are read in one at a time and only one tuple from each relation is in memory at any one time. In this section, we assume that B pages of memory are available. As discussed earlier, we use the idea of parallel execution for the nested loop method on a uniprocessor system. Assuming that one tuple per page, we read in B tuples at a time, making the relation into blocks of tuples.

First, we block the relation on the outer loop. (From now on, the relations on the outer loop and inner loop will be referred as outer relation and inner relation respectively.)

Algorithm 3.2 divides the outer relation into block(s) of B tuples, except maybe the last block. For every tuple read in from inner relation, the tuple is evaluated with all the tuples in the block. The I/O cost of algorithm 3.2 is

$$N + M \left\lceil \frac{N}{B} \right\rceil ,$$

and therefore linear when $M \leqq B$. Using the same variables in Cost (3.2), the cost for algorithm 3.2 is:

$$\text{Cost} = (N * I) + \left\lceil M/B \right\rceil (N * I) + (N * M * E) + O + \Delta. \tag{3.3}$$

```
/*  file1  - contains tuples from outer relation
 *  file2  - contains tuples from inner relation
 *  file3  - output relation
 *  eof1   - end of file1 ( initially false)
 *  eof2   - end of file2 ( initially false)
 *  tuple1 - tuple from file1 or outer relation
 *  tuple2 - tuple from file2 or inner relation
 *  factor - B pages of memory available
 *  WHERE  - function to evaluate "where clause"
 *  tuple1_used_at_least_once - tuple1 used indicator
 *  tuple2_used - tuple2 used indicator
 *               (initially set to false)
 *  TOTtuple1 - total number of tuples in outer relation
 *  TOTtuple2 - total number of tuples in inner relation
 */

   OPEN FILE(file1) INPUT;
   READ FILE(file1) INTO(tuple1);
   DO WHILE( NOT eof1);
      count = 0;
      DO WHILE( NOT eof1 & count < factor );
         TOTtuple1 = TOTtuple1 - 1;
         count = count + 1;
         tup_buf(count) = tuple1;
         tuple1_used_at_least_once(count) = false;
         READ FILE(file1) INTO(tuple1);
      END;
      OPEN FILE(file2) INPUT;
      eof2 = FALSE;
      READ FILE(file2) INTO(tuple2);
      DO J = 1 TO TOTtuple2;
         DO I = 1 TO count;
            IF WHERE(predicates) THEN DO;
               tuple2_used(J) = true;
               tuple1_used_at_least_once(count) = true;
               WRITE FILE(file3) FROM(TUP_BUF(I)||tuple2);
            END;
            ELSE IF TOTtuple1 = 0 & NOT tuple2_used(J) THEN
               WRITE FILE(file3) FROM(nulls1||tuple2);
         END;
         READ FILE(file2) INTO(tuple2);
      END;
      DO I = 1 TO count;
         IF NOT tuple1_used_at_least_once(I) THEN
            WRITE FILE(file3) FROM(tup_buf(I)||nulls2);
      END;
   END;
```

ALGORITHM 3.2 - NESTED LOOP OUTERJOIN WITH BLOCKING

The nested loop algorithm with blocking has been considered the fastest known algorithm to perform a cartesian product between two relations by [25]. [25] also stated that in the worst case, the I/O costs of nested loop algorithm with blocking are better than I/O costs of merging algorithm by a factor of about B.

All these algorithms (3.1 & 3.2) require order of N by M operations (i.e. the cartesian product of the two relations) for all cases of outerjoin operation. This is not desirable when the total number of output tuples is less than the cartesian product of the two relations. Before we introduce the sort/merge algorithm, we would like to present a similar algorithm to sort/merge algorithm called sort/ nested loop algorithm. The difference between the sort/ merge and the sort/nested loop algorithm is that the later does not perform merging or create any intermediate relations.

3.2 SORT/NESTED LOOP

To improve the nested loop algorithm, we introduce sorting to both relations. That is, we sort both relations based on some attributes in the predicates.

By introducing sorting on both relations, we eliminate those tuples that have no possibility of joining at all from the loops. For example, we have 2 Suppliers & Parts relationship relations from STORE1 and STORE2, and we perform an outerjoin on the two relations (with outerjoin condition:

STORE1.s# = STORE2.s#).

STORE1

S#	P#
s3	p1
s5	p3
s5	p7
s8	p5
s9	p2

STORE2

S#	P#
s1	p2
s2	p6
s5	p4
s5	p9
s5	p8
s7	p9

Without sorting, using the nested loop method discussed in the previous section, requires 25 iterations. With sorting on the s# column in ascending order, we reduce the number of iterations from 30 to 19. See Appendix I for details.

In Algorithm 3.3, the attributes x and y represent all attributes with the same Relationship R in the predicates of the outerjoin. The attributes x and y can be the same attributes and represent at least one attribute from each relation. For example, R1.a = R2.a and R1.b = R2.c and R1.d < R2.e then x is {a, b} and y is {a, c}.

```
/* for simplicity the following is assumed.
 * outer relation is sorted based on x.
 * inner relation is sorted based on y.
 * outer - outer relation.
 * inner - inner relation.
 * eor_outer - end of outer relation
 * eor_inner - end of inner relation
 */
```

```
 pos = pos + 1;
 READ outer; READ inner(pos);
 DO WHILE( not eor_outer);
   DO WHILE( not (eor_outer or eor_inner or
             outer.x R inner.y) );
      IF outer.x < inner.y or R is '>' or '>=' THEN
         output (outer || nulls );
         READ outer;
      END;
      ELSE IF outer.x > inner.y THEN
         IF not used(pos) THEN DO;
            output (nulls || inner);
            used(pos) = true;
         END;
         pos = pos + 1;
         READ inner(pos);
      END;
   END;
   IF  not (eor_outer or eor_inner ) THEN DO;
      curpos = pos;
      tpl_not_used = true;
      DO UNTIL( eor_inner or not outer.x R inner.y );
         IF outer RR inner  THEN DO;
            output (outer || inner);
            used(pos) = true;
            tpl_not_used = false;
         END;
         pos = pos + 1;
         READ inner(pos);
      END;
      IF tpl_not_used THEN
         output (outer || nulls);
      pos = curpos;
      READ inner(pos);
      READ outer;
   END;
   ELSE IF not eor_outer THEN
      DO UNTIL ( eor_outer );
         output (outer || nulls);
         READ outer;
      END;
END;

DO WHILE( not eor_inner );
    IF not used(pos) THEN
       output (nulls || inner);
    pos = pos + 1;
    READ inner(pos);
END;
```

ALGORITHM 3.3 - SORT/NESTED LOOP

The cost for this sort/nested loop algorithm cannot be computed easily, because it is actually breaking the nested loops into pieces of single loop. For each matching tuple, there is a single loop for the remaining matching (at least 'FIRST CONDITION', which is discussed in the next chapter) tuples in the inner relation.

$$\text{Cost} = (M + N_i + N_0) * (E + I) + O + \Delta \qquad (3.4)$$

where N_i is the remaining matching tuples in the inner relation for each ith single loop; N_0 is beginning unmatched tuples of the inner relation; M is the total number of outer tuples; E and I is the execution and input time respectively, and O is the output time.

The algorithms with blocking of outer and inner relations using sort/nested loop method are shown in Appendix B and C respectively.

CHAPTER IV

SORT/MERGE METHOD

In this chapter, we will look into another type of join method, the sort/merge method. This join algorithm has been considered to be better than the nested loop method in terms of the number of operations by [28]. This algorithm employs a sort of the operand relations on the join attributes, followed by merge-type operation of the two sorted relations to complete the join [28]. The implementation of the sort/merge join is slightly more complex than it seems from this simple description. If neither of the two join attributes is an unique key to its relation (i.e. the join implements a many-to-many relationship), intermediate relations may have to be built. Therefore this is normally considered to be a more complex algorithm than the nested loop methods (without any sorting). (Note: we are assuming many-to-many relationship in implementing the join.)

The sort/merge equal-join algorithm [18] sorts the relations based on the join attributes. The algorithm then scans through both relations from the top until it reaches a point where the join attributes from both relations are equal or either one of the relations runs out. Assuming that it reaches a point where the join attributes from both relations are equal, then it continues to find more matching tuples from the inner relation and performs the join operation. At the same time, these matching tuples from the inner relation are stored in intermediate storage. These tuples in the intermediate storage are then used to join with the tuples from the outer relation if their join attributes are equal to the attribute of the first tuple that satisfies the join condition. The intermediate storage for the matching tuples from the inner relation can be as large as the inner relation, which occurs in the worst case. Hence, we can say that the size of the intermediate storage is influenced by the choice of the inner and outer relations.

4.1 Sort/Merge Method for Outer-equal-join.

We modify the sort/merge join algorithm into a sort/ merge outer-equal-join algorithm. This is easily done by joining the tuple with null tuple from the other relation when the tuple is determined to be non-joined tuple.

```
/* outer, inner : the two relations to be joined.
 * outer~, inner~ : buffers for the last read elements.
 * outer~.f, inner~.g : the join attributes.
 * current : a variable indicating the current join value.
 * intermediate : holds intermediate tuples
 */
  sort(outer by f); sort(inner by g);                  (step 1)
  READ outer; READ inner;
                                                       (step 2)
  DO WHILE NOT (eor_inner OR eor_outer OR
              outer~.f ≠ inner~.g);
      IF outer~.f < inner~.g THEN DO
         OUTPUT (outer~ || inner~.nulls);
         read(outer);
      END
      ELSE DO
         OUTPUT (outer~.nulls || inner~);
         read(inner);
      END
    END:
    IF NOT (eor_inner OR eor_outer) THEN DO;
      (* Cartesian product of joining subrelations *)
      intermediate = '';
      current := outer~.f;                             (step 3)
      DO UNTIL(inner~.g ≠ current OR eor_inner );
        OUTPUT (outer~ || inner~);
        intermediate = intermediate + 'inner~';
        read(inner);
      END;
      read(outer);                                     (step 4)
      WHILE(outer~.f = current AND NOT (eor_outer));
        FOR EACH irec IN intermediate DO
            OUTPUT (outer~ || irec);
        read(outer);
      END
    END
  UNTIL eor_outer OR eor_inner
END.
```

ALGORITHM 4.1 - SORT/MERGE OUTER-EQUAL-JOIN

First, the algorithm (Algorithm 4.1) scans through both relations until it finds the matching tuples and proceeds to the third step, and at the same time the unmatched tuples are joined with the null tuples. In the third step, it tries to find as many matching tuples from relation 2 as possible until the join condition is not met. During the third step, all the tuples that satisfy the join condition are kept in intermediate storage for later use in the fourth step. In the fourth step, the next outer tuples are joined with all the intermediate tuples if the next outer tuple is equal to the current outer tuple. The process is repeated until both relations are exhausted. With exception to step 2, this sort/merge outer-equal-join algorithm is exactly like the sort/merge equal-join algorithm.

For example, if you have an outer-equal-join on REL1.A and REL2.A on the following tables:

REL1

A	B
2	3
2	2

REL2

A	B
1	1
2	4
2	3

The first tuple in REL1 is not equal to the first tuple in REL2, so you advance to the second tuple in REL2 because the first tuple of REL2.A is less than first tuple of REL1.A, and first tuple of REL2 is joined with null tuple of REL1. The second tuple of REL2.A is equal to the first tuple of REL1.A, therefore, the second tuple of REL2 is kept in intermediate storage and join with the first tuple of REL1. Then the third tuple in REL2 is compared with first tuple from REL1. Since the third tuple of REL2.A is equal to the first tuple of REL1.A, it is also kept in the intermediate storage and join with the first tuple of REL1. So, the intermediate storage contains two tuples from REL2. There are no more tuples from REL2, so we go on to the next step. That is to see if the next (second) tuple of REL1.A is equal to the first tuple of REL1.A. If they are, the second tuple of REL1 is joined with all the tuples in the intermediate storage. The algorithm terminates because both relations run out of tuples. The resulting table from the above operations is as follow:

REL1.A	REL1.B	REL2.A	REL2.B
?	?	1	1
2	2	2	4
2	2	2	3
2	3	2	4
2	3	2	3

The cost of I/O (CI) for the sort/merge Outer-equal-join algorithm is

$$CI = (M + N) * I + O; \tag{4.1}$$

The cost of comparisons (CC) for the sort/merge Outer-equal-join algorithm in the worst case is [19]

$$CC = (M + N - 1) * E; \tag{4.2}$$

The minimum comparisons for the sort/merge Outer-equal-join algorithm is

$$CC = N*E; \tag{4.3}$$

(note: Comparisons within the same relation is not included in the cost.)

So far, the assumption is that there is an infinite amount of main storage, which is not necessarily true in the real environment. Therefore, all the intermediate tuples may have to be stored in a secondary storage. This inevitably decrease the performance of the sort/merge outerjoin significantly.

4.2 Sort/Merge Method for Outer-theta-join.

To do an outer-theta-join using the sort/merge method is not a simple task. It is found to be a very complex and time consuming algorithm by [28]. We use the idea of sort/merge to implement the outer-theta-join. This outer-theta-join method breaks the predicate's structure into three parts. The first part of the predicate is called 'FIRST CONDITION' predicate, which is the main condition where the sort/merge is applied. The second part is called the 'LESS THAN CONDITION', which is the less than condition for elementary predicates contained in the 'FIRST CONDITION'. The 'LESS THAN CONDITION' is used to eliminate unmatched tuples from the relations. The last part is called the 'SECOND CONDITION' or 'REMAINING CONDITION', which consists of the remaining join conditions not in the 'FIRST CONDITION'.

Since the 'FIRST CONDITION' will be used as the main condition for the sort/merge algorithm, then the relation will be sorted based on the attributes in the 'FIRST CONDITION'. The criteria for 'FIRST CONDITION' are based on the following conditions:

(1) have only one kind of relational operator;
(2) select all predicates with "=" operator;
(3) if none of (2), select all predicates with operator of the same type;
(4) predicates selected with key attributes are placed first.

Evaluation of the predicates will be from left to right and terminates if any predicate returns false. Example of how an outerjoin query is set up for the sort/merge algorithm is as follow:

Let the query be as :

```
   Ojoin * ( R1, R2 ) where R1.x = R2.x and
                            R1.y = R2.y and
                            R1.z < R2.z
```

(assuming that x and y are key attributes)

Then

a) FIRST CONDITION is

R1.x = R2.x and R1.y = R2.y

b) LESS THAN CONDITION is

R1.x < R2.x or (R1.x = R2.x and R1.y < R2.y)

c) SECOND CONDITION is

R1.z < R2.z

Note: if there is no SECOND CONDITION then evaluating
SECOND CONDITION is always true.

The sort/merge algorithm for outer-theta-join can be written by combining the outer-equal-join algorithm with the sort/nested loop algorithm and some additional modifications. Assuming that there is no dynamic storage, the intermediate relation will be kept in inner relation using virtual indexes on the tuples. That is, we cursor the indexes of the first and last tuples from the inner relation (the intermediate tuples) which satisfied the 'FIRST CONDITION'.

```
/* for simplicity the following is assumed.
 * outer relation is sorted based on x.
 * inner relation is sorted based on y.
 * outer - outer relation.
 * inner - inner relation.
 * outer.x - the 'FIRST CONDITION' attributes for outer
 *           relation
 * inner.y - the 'FIRST CONDITION' attributes for inner
 *           relation
 * R     - Relation, i.e xRy, for the 'FIRST CONDITION'
 * RR    - Relation for the 'SECOND CONDITION'
 * n_inner - total number of inner tuples.
 * p_outer - pth outer tuple.
 * p_inner - pth inner tuple.
 * start_inner - first inner tuple that satisfies the FIRST
 * inner.used - indicate whether inner tuple used or not.
 * outer.used - indicate whether outer tuple used or not.
 * current - current outer tuple.
 * lookahd - look ahead inner tuple.
 * WHERE - evaluates the predicates.
 * FIRST - FIRST CONDITION predicate.
 * SECOND - SECOND CONDITION predicate.
 * LESS   - LESS THAN CONDITION predicate.
 * ++ - increment by one.
 */

READ outer;
```

```
READ inner;
DO WHILE( not(eor_outer and eor_inner);           (STEP 1)
 DO WHILE( not (eor_outer or eor_inner
    or WHERE(FIRST) ));
  IF  WHERE(LESS)  THEN DO;
     output(outer || nulls);
     READ outer;
  END;
  ELSE IF not eor_inner THEN DO;
     output(nulls || inner);
     READ inner(++p_inner);
  END;
 END;
 IF not (eor_outer or eor_inner) THEN
        start_inner = p_inner;
        CURRENT = outer;
        outer.used = '0'B;
                                                  (STEP 2)
        DO UNTIL( eor_inner or WHERE(FIRST) );
           IF WHERE(SECOND) THEN DO;
              output (outer || inner);
              mark inner used;
           END;
           outer.used = '1'B;
           lookahd = inner;
           READ inner(++p_inner);
        END;
        IF not outer.used THEN
           output (outer || nulls);
/*                                      */
/* is the next remaining outer tuple    */
/* equal to the current outer tuple     */
/* Or satisfies the 'WHERE' evaluation  */
/*                                      */
        READ outer;
        outer.used = false;
                                                  (STEP 3)
        DO WHILE( not eor_outer and ( outer.x = current.x
           or WHERE(FIRST))) ;
           MORE = true;
           IF no SECOND CONDITION THEN
              DO I = start_inner TO p_inner-1;
                 READ inner(i);
                 output (outer || inner);
                 mark inner used;
              END;
              outer.used = true;
           END;
           ELSE
           DO I = start_inner TO p_inner-1;
              READ inner(I);
              IF WHERE(SECOND) THEN DO;
                 output (outer || inner);
                 mark inner used;
```

```
                 outer.used = true;
              END;
           END;
           IF not outer.used then
              output (outer || nulls);
           READ outer;
           outer.used = false;
        END;
        ADVAN = true;
                                                   (STEP 4)
        IF ( not eor_outer and R is not '=' and
                    (p_inner-1) > start_inner ) THEN DO;
           IF WHERE(FIRST) on lookahd THEN DO;
             p_inner = start_inner + 1;
             reset eor_inner;
             ADVAN = false;
           END;
           READ inner(p_inner);
        END;
                                                   (STEP 5)
        IF no second condition and ADVAN is true THEN DO;
           DO I = start_inner TO p_inner-1;
              READ inner(I);
              IF not inner.used THEN
                 output (nulls || inner);
           END;
           READ inner(p_inner);
        END;
     END;
     ELSE IF not eor_outer THEN                    (STEP 6)
     DO WHILE ( not eor_outer);
        output (outer || nulls);
        READ outer;
     END;
     ELSE IF not eor_inner THEN                    (STEP 7)
     DO WHILE ( not eor_inner);
        IF not inner.used THEN
           output (nulls || inner);
        READ inner(++p_inner);
     END;
  END;
 END;
END;
```

ALGORITHM 4.2 - SORT/MERGE OUTERJOIN

Let us examine the sort/merge outerjoin Algorithm 4.2.

Step 1. This step eliminates all non-possible join tuples.

These eliminated tuples are joined with the appropiate null tuple. For all the outer and inner tuples eliminated, the number of iterations reduced is equal to the cartesian product of the outer and inner tuples eliminated.

Step 2. In this step, try to join the outer tuple with as many inner tuples as possible before it is eliminated from the process. But the inner tuples that matched (fully or partially, called the intermediate tuples) with the outer tuple are not necessarily eliminated after step 3. The reason is if there is a 'SECOND CONDITION' and the 'FIRST CONDITION' is not an equal type of condition, then there is a possibility that the outer tuple might match the intermediate tuples.

Step 3. In this part, use the case when the attributes of the 'FIRST CONDITION' of the next remaining outer tuples and the current outer tuple are equal. Then by transitive definition i.e. if c = a and a = b then c = b [20], the next remaining outer tuple is equal to the inner tuples of step 2 as far as the 'FIRST CONDITION' is concerned. If there is not 'SECOND CONDITION' then all the intermediate tuples are joined with the next remaining outer tuple without having to do any comparison. The real advantage is when there is a large number of intermediate tuples from step 2, let say k, then k comparisons are saved(at least partially, if there is a 'SECOND CONDITION').

Step 4. In this part, try to determine whether there is any possibility for the remaining outer tuple to match the intermediate tuples of step 2. This is done by having a lookahead tuple. If the next remaining outer tuple does not satisfy the lookahead tuple on the 'FIRST CONDITION', then there is no possible join for the next remaining outer tuple to match with the intermediate tuples of step 2.

Step 5. In this part, all the unused intermediate tuples of step 2 and 3 are joined with the null tuple.

Step 6. Join all the remaining outer tuples with null tuple when the end of inner relation has been encountered.

Step 7. Join all the remaining inner tuples with null tuple when the end of outer relation has been reached.

The cost of I/O (CI) for the above algorithm is

$$CI = (M + N_j) * I; \quad (\text{for } j = 1 \text{ to } M) \qquad (4.4)$$

where N_j is the intermediate tuples for the jth iteration. The cost of comparisons (CC) depends on the type of outer-join operations. For outer-equal-join the cost of comparisons is the same as the cost of comparisons for sort/merge equal-join algorithm (COST 4.2). The Cost of comparisons for outer-theta-join (non-equal) is between N*E and M*N*E.

$$N*E <= CC <= M*N*E$$

The above algorithm assumes no primary storage for the tuples. Dynamic storage can be used to hold intermediate tuples, but if the storage for the intermediate tuples is larger than the memory available then there is a problem. This problem can be solved by splitting the operation into a few outer-join operations that have enough memory to hold the intermediate tuples. (This is very similar to blocking the inner relation with dynamic storage.) For methods using blocking on outer and inner relations see Appendix D and E respectively.

CHAPTER V

ANALYSIS AND EVALUATION OF OUTERJOIN METHODS

In this chapter, we analyze each of the algorithms presented in earlier chapters. For the purpose of testing, we introduce two tables or relations called REL1 and REL2 (Appendix A). The values of these two tables are randomly generated. Each table has five columns, naming A, B, C, D, E for REL1 and U, V, W, X, Y, Z for REL2. For simplicity, all the values are assumed to be positive integers.

The algorithm is analyzed to see how well it performs against outer-join queries of:

a) one predicate of equal condition.
b) one predicate of less than condition.
c) one predicate of greater than condition.
d) multiple predicates of theta conditions.

Again, for simplicity the queries are as follow:

For one predicate of equal condition the outer join query is

OJOIN * (REL1, REL2) WHERE REL1.A = REL2.U and, the query for one predicate less than condition is

OJOIN * (REL1, REL2) WHERE REL1.A < REL2.U and, the query for one predicate great than condition is

OJOIN * (REL1, REL2) WHERE REL1.A > REL2.U and, for multiple predicates of theta conditions are

```
OJOIN * ( REL1, REL2 ) WHERE REL1.A = REL2.U AND
                                     REL1.B < REL2.V
```

For each of the above queries, the query is run for 10 times on an outerjoin algorithm to get the average results. The results that are recorded for measuring the performance of the outerjoin algorithm are:

1) the number of tuples in the outer relation;
2) the number of tuples in the inner relation;
3) the number of 'read's performed;
4) the actual number of comparisons;
5) the total number of joins performed;
6) the total number of outerjoin tuples produced;
7) the average C.P.U. time, in seconds, required to perform the outer-join operation.

For the purpose of comparison, the following variables and values are used:

1) The relation sizes are 100 tuples for outer relation, and 150 tuples for the inner relation;
2) The blocksize is 50 tuples if blocking is used;
3) Each tuple is 30 bytes;
4) The I/O buffer is 10k bytes;
5) The size of the VM machine is 2m bytes;
6) C.P.U. time is measured in seconds, and only the actual operation of outerjoin will be measured. Sorting time

for the relations is not included in the computation.

5.1 No Dynamic Storage

The results obtained from the above queries for algorithms which did not use dynamic storage, as in this research, are presented in table I, II, III, and IV.

Since the nested loop method (NL) is the simplest and easiest way to implement outerjoin, it is used as the control method to determine how well the other methods perform relatively.

Table I shows that sort/merge method (SM) has the least number of inputs (or reads) and NL has the most (the maximum inputs using the formula in [26]). The sort/nested loop method (SN) has 96.96% less inputs and 98.28% less comparisons than NL. With SM, we save 98.27% inputs and 99.0% comparisons. In respect of C.P.U. time, SN is about 4.5 times and SM is about 3.6 times faster than NL. Looking at figure 1, SM is definitely the best, followed by SN, in terms of C.P.U. time and number of joins for outer-equal-join queries. Figure 2 shows that the number of comparisons stays the same with respect to number of joins for NL and SM. As for SN, the number of comparisons increases as the number of joins increases. Figure 3 shows the number of inputs with respect to the number of joins.

TABLE I

OJOIN * (TABLE1, TABLE2) WHERE A = U
(NO DYNAMIC STORAGE)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	15100	15000	112	156	2.2630
SN	100	150	462	260	112	156	0.4790
SM	100	150	259	150	112	156	0.1180

TABLE II

OJOIN * (TABLE1, TABLE2) WHERE A < U
(NO DYNAMIC STORAGE)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	15100	15000	7943	7945	3.6760
SN	100	150	8292	8093	7943	7945	2.6240
SM	100	150	8096	7796	7943	7945	2.7810

TABLE III

OJOIN * (TABLE1, TABLE2) WHERE A > U (NO DYNAMIC STORAGE)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	15100	15000	6945	6948	3.6440
SN	100	150	7292	6946	6945	6948	2.3280
SM	150	100	7099	4829	6945	6948	2.5120

TABLE IV

OJOIN * (TABLE1, TABLE2) WHERE A = U AND B < V (NO DYNAMIC STORAGE)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	15100	15000	7	243	2.8070
SN	100	150	459	260	7	243	0.4920
SM	100	150	458	156	7	243	0.5160

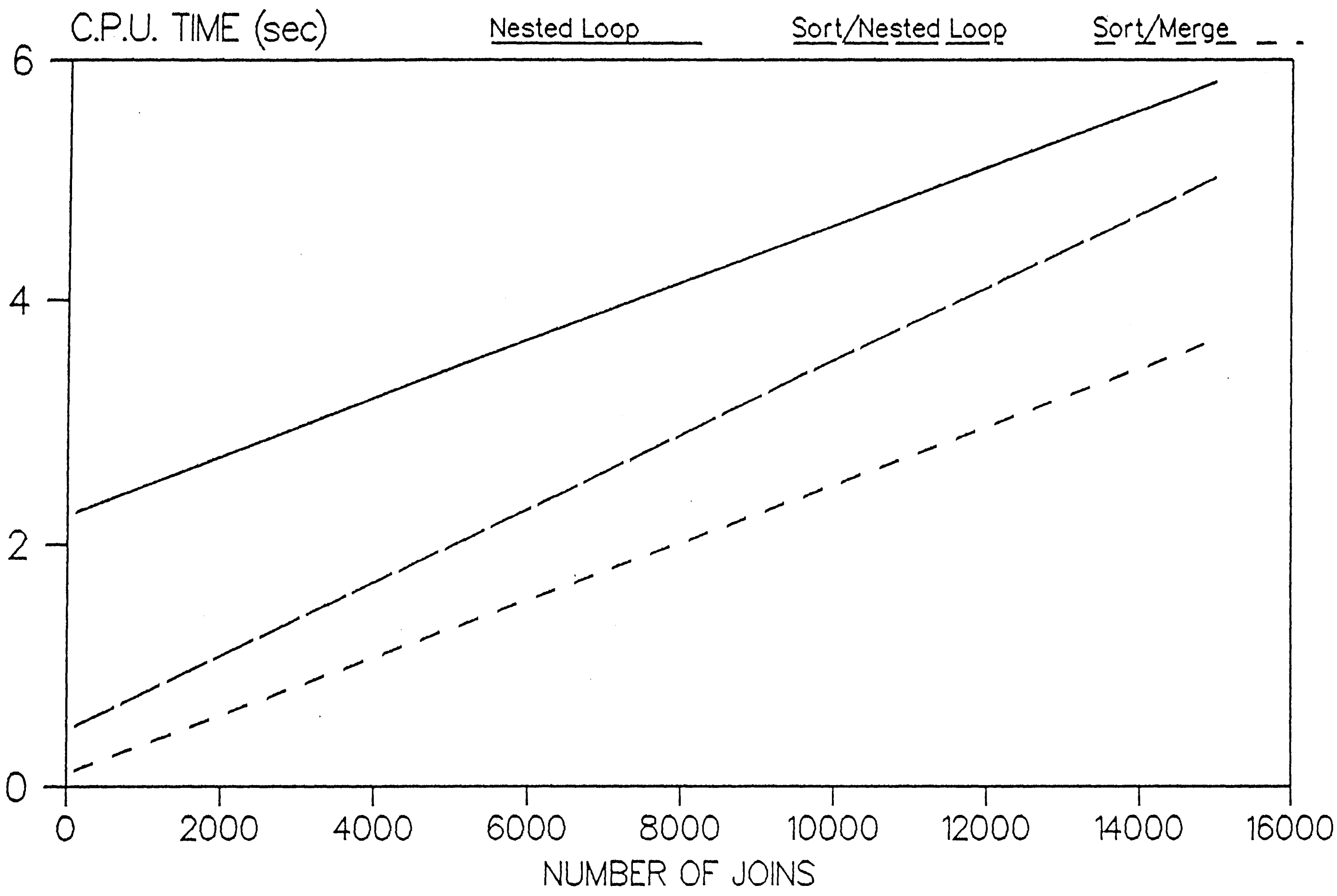


Figure 1. C.P.U. Time per Join Tuple

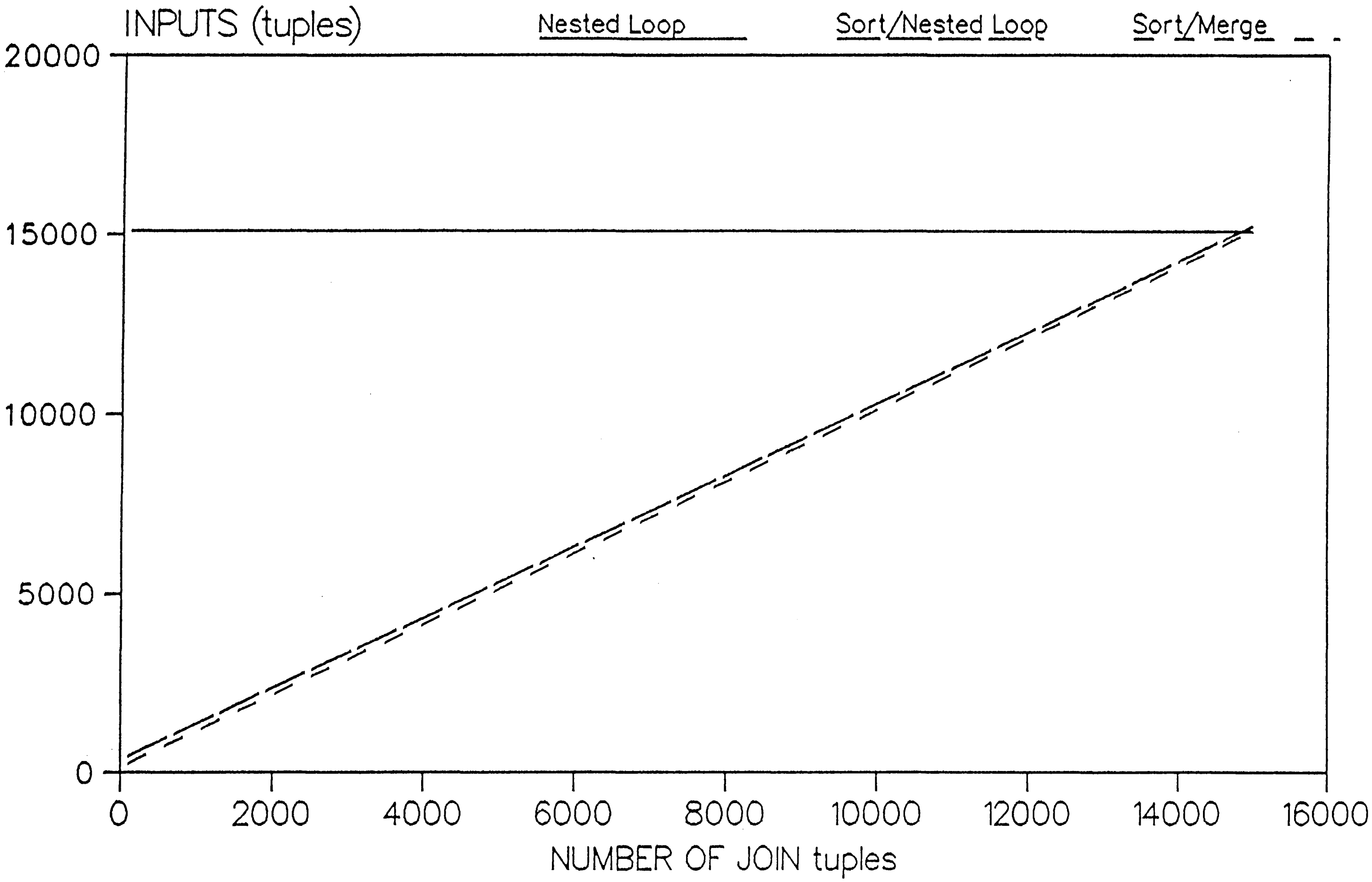


Figure 2. The Number of Inputs per Join Tuple

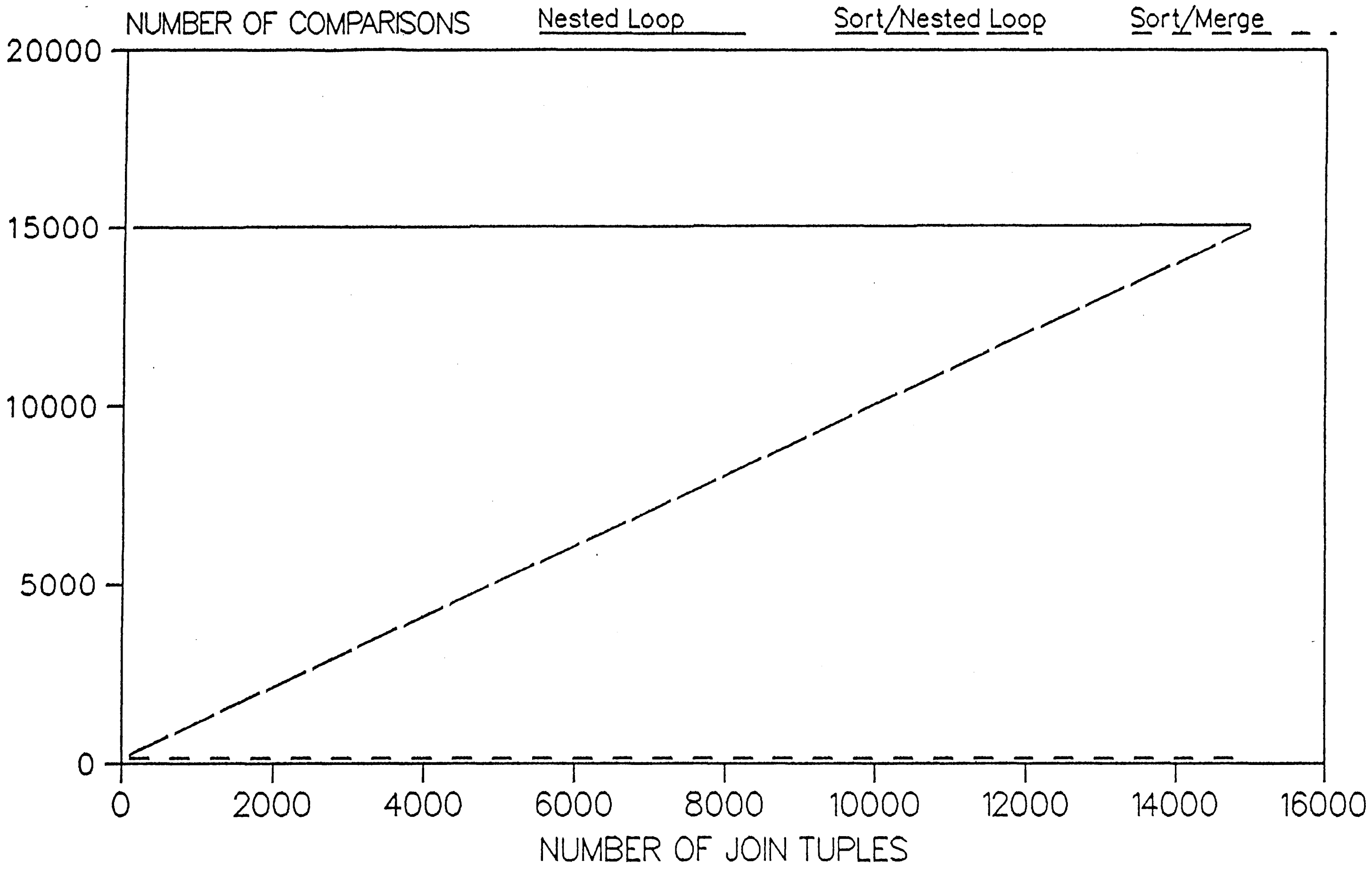


Figure 3. The Number of Comparisons per Join Tuple

For non-equal outerjoin queries, as shown in TABLE II and III, we save about 50% of inputs and comparisons when using SN and SM.

The number of tuples in the outer and inner columns on TABLE III are different for SM because of the '>' condition which is not handled by the algorithm directly. The SM handles the '>' condition by swapping the outer and inner relations so that the '>' condition(s) become '<' condition(s). The results of the query are not affected by the swapping. (For example of these effects, see Appendix H).

The results for query 4 (multiple predicates query) is shown in TABLE IV. As we might expect, SM and SN are better than NL. Notice that the C.P.U. time for SN is slightly better than the C.P.U. time for SM. For an outer-equal-join (fully or partially), the results in terms of C.P.U. time depends largely on the relations and the number of joins. As we have seen earlier, the difference between SN and SM is the way they handle the intermediate tuples. SN does not handle intermediate tuples, that is the inner tuples that match the outer tuple, are not kept for re-use in looping the outer relation as it is done in SM. The cursor of the inner relation for SN always returns to the position of the first matching inner tuple after each loop. SN always has better results if the intermediate tuples in SM have to be reused in the operation, that is, backing up to the previous tuples starting from the first intermediate tuple, making the process like that of SN. Since SM has a higher overhead than SN, SN has a better result. However, this is not true if the intermediate tuples are joined with more than one outer tuple or never reused(only in the case when intermediate tuple is more than one).

5.2 Use of dynamic storage

The next two sections present the test results for algorithms that use some form of dynamic storage.

As mentioned earlier, using some form of dynamic storage can improve the outerjoin operation. The dynamic storage is fixed because we can not assume infinite amounts of dynamic storage. Therefore, the fixed dynamic storage is not used to hold the intermediate tuples in SM.

First we would like see how blocking on outer relation can improve or in some cases worsen the algorithms. TABLE V, VI, VII, and VIII show the results for query 1, 2, 3, and 4 respectively.

If you compare the results of TABLE V with TABLE I, you'll see that NL improved the most, more than 50%, in C.P.U. time. This is due to the idea of (b - 1) processors described in chapter 3. In this case, 50 tuples from the outer relation in memory are going against one tuple from inner relation at a time.

TABLE V

OJOIN * (TABLE1, TABLE2) WHERE A = U
(BLOCK OUTER RELATION, BLOCKSIZE = 50)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	400	15000	112	156	1.0100
SN	100	150	418	228	112	156	0.6160
SM	100	150	409	300	112	156	0.1000

TABLE VI

OJOIN * (TABLE1, TABLE2) WHERE A < U
(BLOCK OUTER RELATION, BLOCKSIZE = 50)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	400	15000	7943	7945	1.7630
SN	100	150	320	8063	7943	7945	2.6710
SM	100	150	8166	7866	7943	7945	3.0780

TABLE VII

OJOIN * (TABLE1, TABLE2) WHERE A > U (BLOCK OUTER RELATION, BLOCKSIZE = 50)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	400	15000	6945	6948	1.6670
SN	100	150	350	6847	6945	6948	2.4330
SM	150	100	7209	4939	6945	6948	2.7260

TABLE VIII

OJOIN * (TABLE1, TABLE2) WHERE A = U AND B < V (BLOCK OUTER RELATION, BLOCKSIZE = 50)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	400	15000	7	243	1.1540
SN	100	150	418	228	7	243	0.5270
SM	100	150	508	306	7	243	0.3890

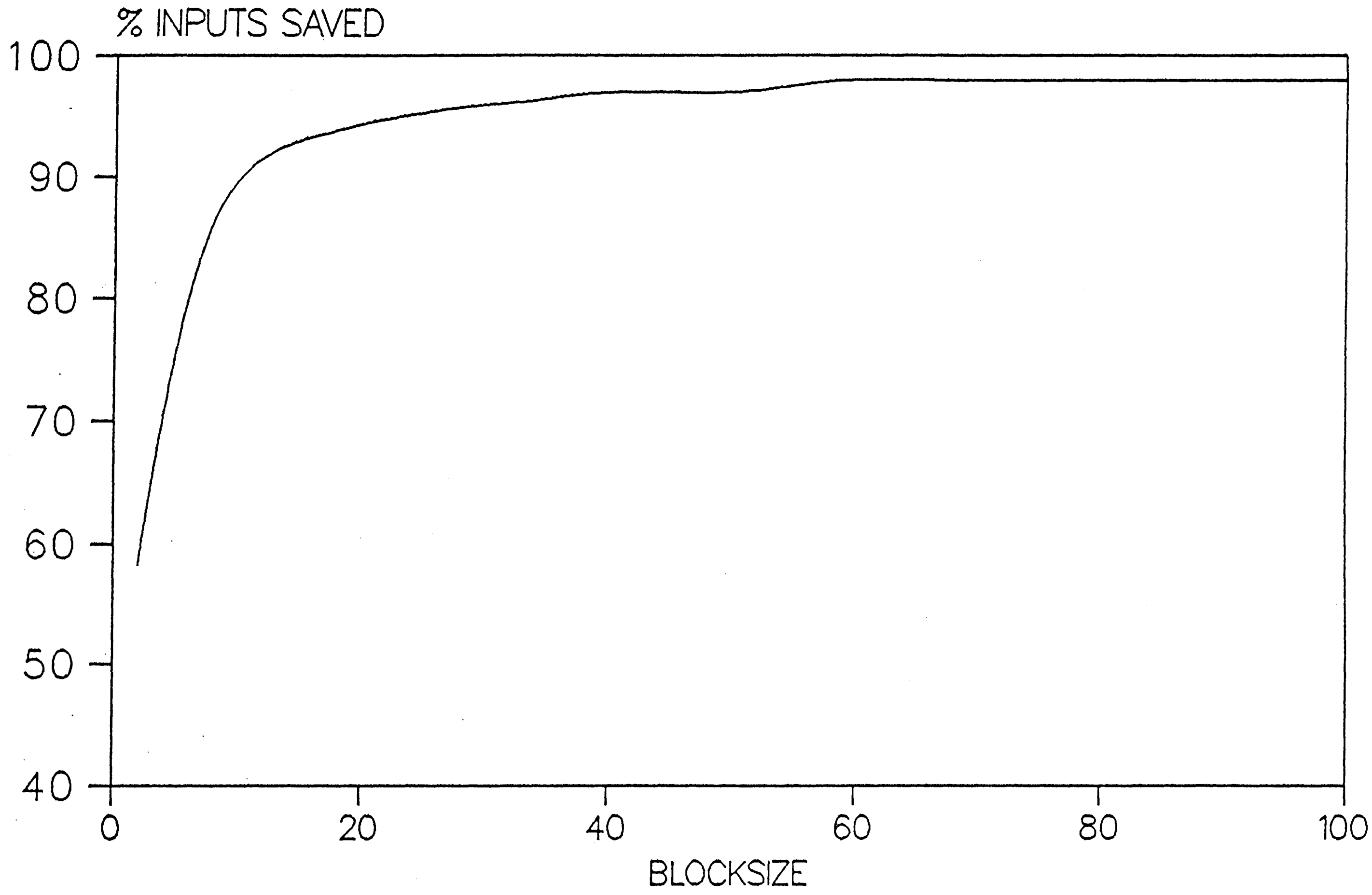


Figure 4. The Percentage of Inputs Saved Relatively to Blocksize

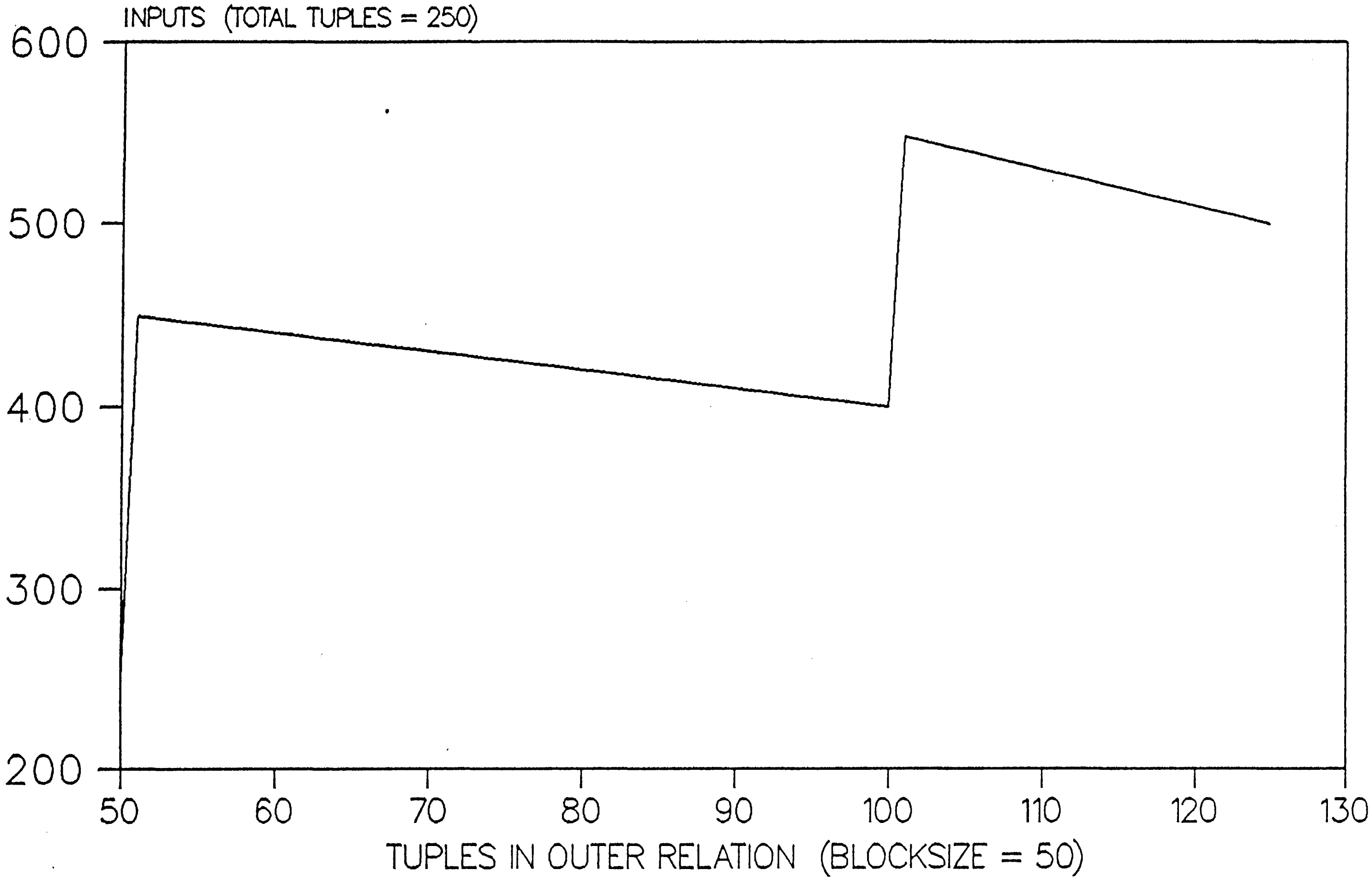


Figure 5. The Number of Inputs Affected by Size of Relations (Block Outer Relation)

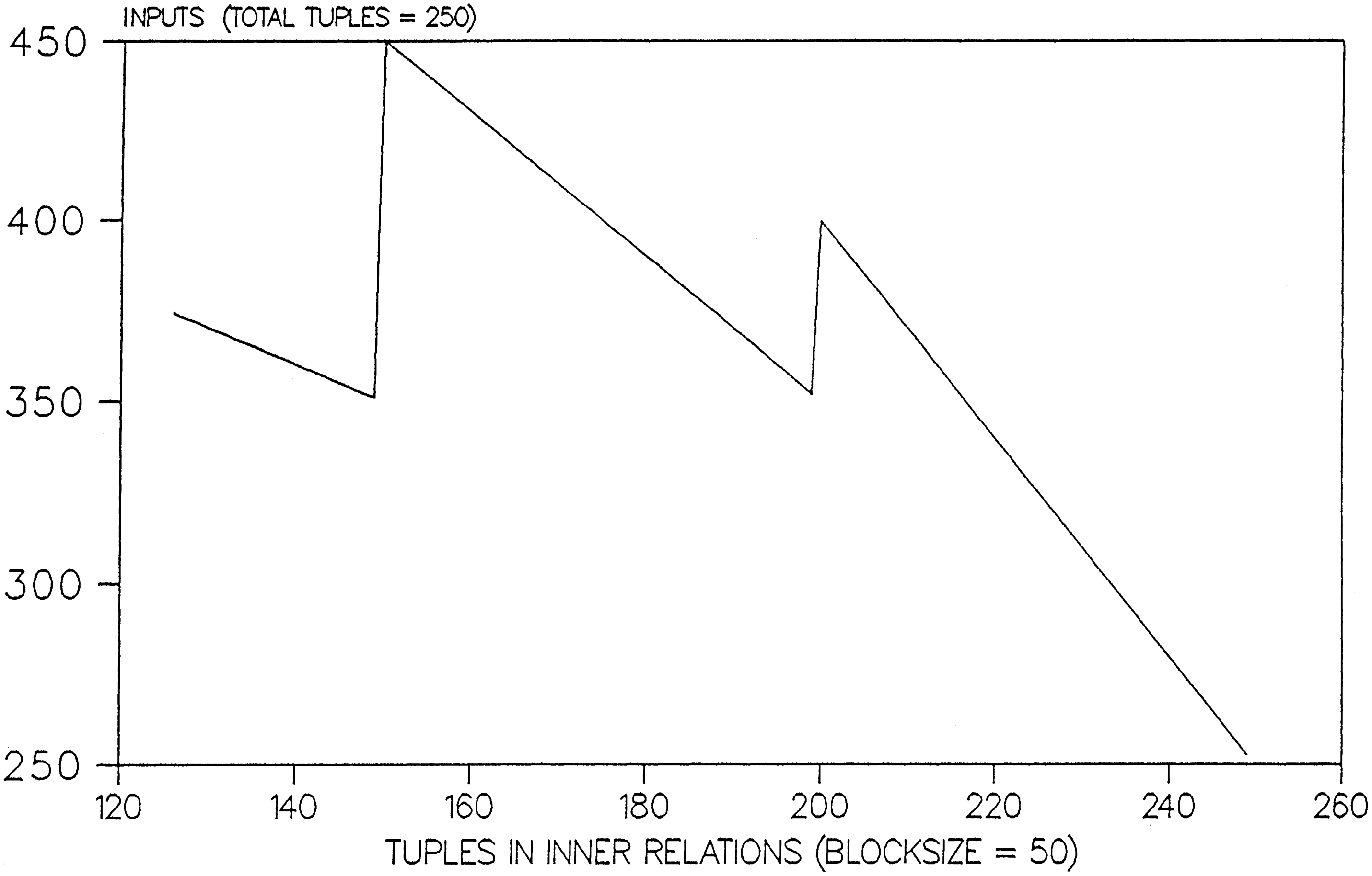


Figure 6. The Number of Inputs Affected by Size of Relations (Block Inner Relation)

5.2.1 Block the Outer Relation

SN has a slight improvement in the number of inputs and comparisons but not in C.P.U. time. In fact, the C.P.U. time increases. The increase in C.P.U. time is due to the number of SN operations (total number of outer tuples divided by the blocksize) it has to perform which in this case is 2. First with the initial 50 tuples of the outer relation and the inner relation, and second with the next 50 tuples of the outer relation and inner relation. The same thing happens to SM, although its performance is not worse than without the dynamic storage. Therefore, we can conclude that blocking on outer relation does not help SN or SM.

5.2.2 Block the Inner Relation

You have seen the results for blocking on outer relations. In this section, you will see how blocking on the inner relation can improve the algorithms. The results for query 1, 2, 3, and 4, using the blocking on an inner relation, are shown in TABLE IX, X, XI, and XII respectively.

TABLE IX

OJOIN * (TABLE1, TABLE2) WHERE A = U (BLOCK INNER RELATION, BLOCKSIZE = 50)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	450	15000	112	156	1.0500
SN	100	150	360	268	112	156	0.0852
SM	100	150	360	258	112	156	0.1700

TABLE X

OJOIN * (TABLE1, TABLE2) WHERE A < U (BLOCK INNER RELATION, BLOCKSIZE = 50)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	450	15000	7943	7945	1.7650
SN	100	150	358	7888	7943	7945	1.3460
SM	100	150	358	2640	7943	7945	0.9010

TABLE XI

OJOIN * (TABLE1, TABLE2) WHERE A > U (BLOCK INNER RELATION, BLOCKSIZE = 50)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	450	15000	6945	6948	1.6880
SN	100	150	450	6867	6945	6948	1.2750
SM	150	100	319	2427	6945	6948	1.1490

TABLE XII

OJOIN * (TABLE1, TABLE2) WHERE A = U AND B < V (BLOCK INNER RELATION, BLOCKSIZE = 50)

Method used	Number of tuples		Number of			Total output tuples	Average C.P.U (sec)
	outer	inner	inputs	comparisons	joins		
NL	100	150	450	15000	7	243	1.2210
SN	100	150	360	268	7	243	0.0950
SM	100	150	360	264	7	243	0.1800

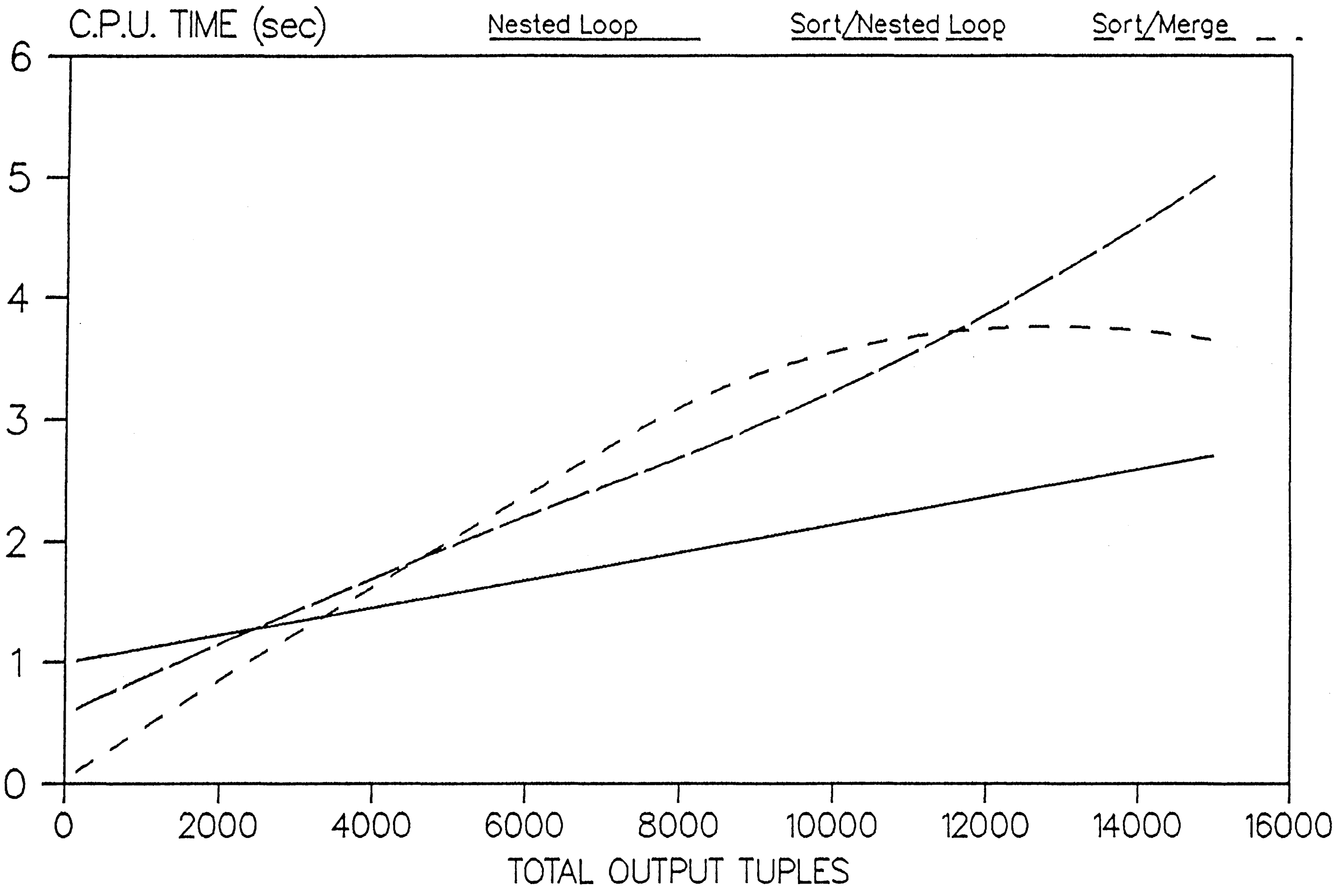


Figure 7. C.P.U. Time per Tuple (Block Outer Relation)

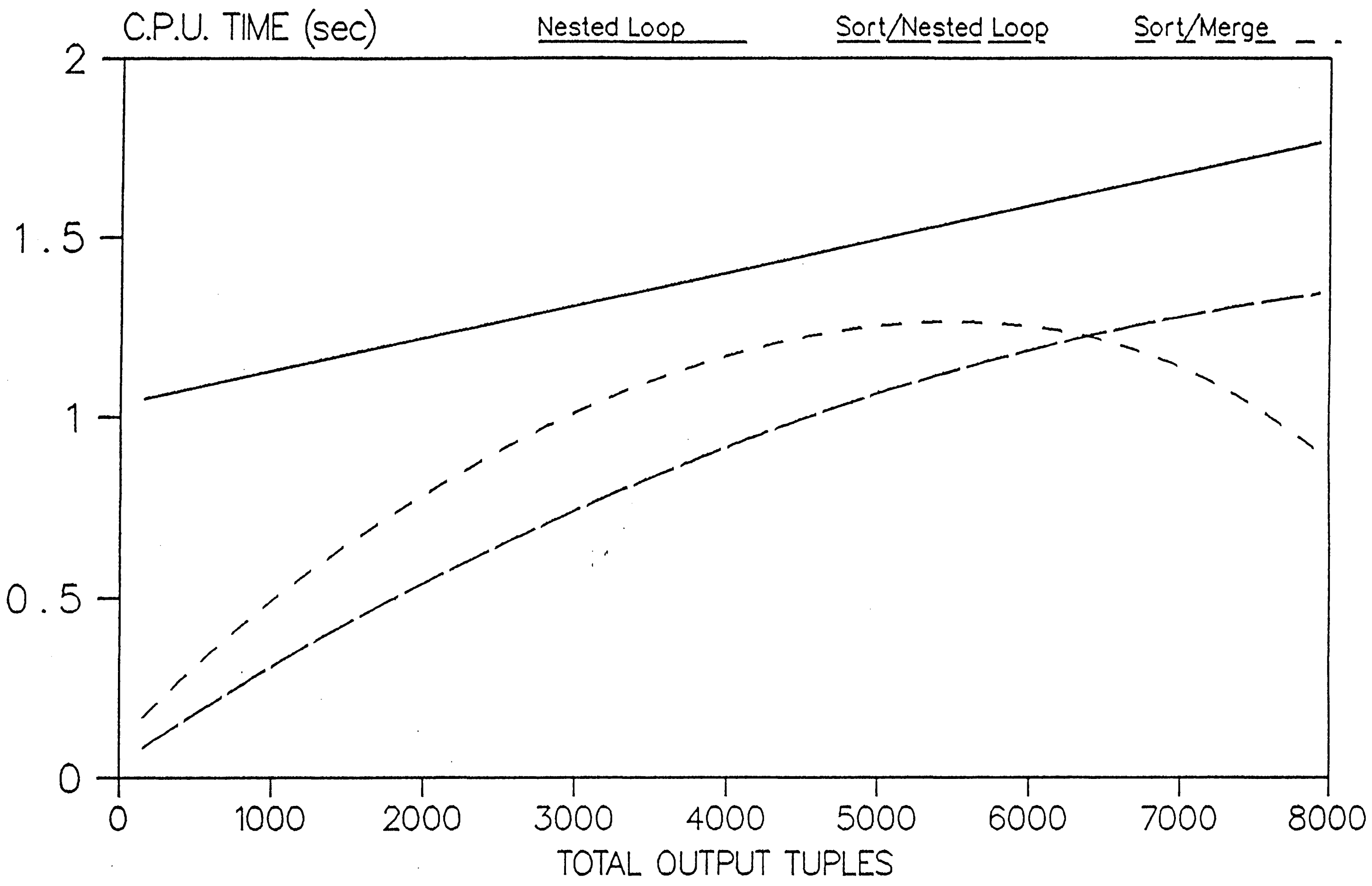


Figure 8. C.P.U Time per Tuple (Block Inner Relation)

NL performs almost the same as when blocking on outer relation is used, except that blocking on inner relation results in more inputs, thereby causing a slight increase in C.P.U. time. The number of inputs is relatively dependent on the number of tuples in outer and inner relation and the blocksize. For blocking on inner relation, the formula is inputs = outer relation * (inner relation/blocksize) + inner relation. For blocking on outer relation, the formula is inputs = outer relation + inner relation * (outer relation/ blocksize). If we hold the blocksize constant, then you will see how the size of outer and inner relation affect the number of inputs in the NL algorithms. See Figure 5 and 6.

Blocking on inner relation does improve the algorithm in all the categories (inputs, comparisons, and C.P.U. time), except for outer-equal-join queries. Theoretically speaking, blocking on an inner relation should improve the outer-equal-join queries too. If the number of intermediate tuples is small or the attributes are unique key attributes, then blocking has no effect at all because it does not take advantage of the inner tuples already in memory.

As for SN, blocking on inner relation improve the algorithm in all the categories and queries, except the number of comparisons for queries with equal join conditions.

5.3 Checking for no possible join

It is possible that queries do not produce any join tuples. We can check to determine whether there is any possibility for the relations to have join tuples. Prior to initiating the query, if no join tuple is produced, then forfeit the entire outerjoin process and simply join the relations with null tuples. However, checking procedure is applicable to the 'sort' methods only. The ways to determine whether there is any possibility for the relations to have join tuples are:

1) For the equal-join condition, select the larger between the two first tuples of outer and inner relations. If the larger value of the tuple is greater than the last tuple of the lesser value's relation then we can say that there is no possible way to have a joined tuple for the two relations;

2) For queries with the 'less than' (or 'less than equal to') condition, if the first tuple of the outer relation is not less than (or less than equal to) the last tuple of the inner relation, then there is no possible join in the two relations;

3) For queries with the 'greater than' (or 'greater than equal to') condition, if the first tuple of the inner relation is not less than (or less than equal to) last tuple of the outer relation, then there is no possible join in the two relations.

The algorithm to do the checking is in Appendix F.

This checking algorithm is cost-effective and only adds 3 reads and 2 comparisons to the outer-join operation. If there is no possibility of join in the outer-join operation, then the saving is at least (the total number of tuples - 2) comparisons, depending on the type of outer-join algorithms used.

CHAPTER VI

CONCLUSION AND FUTURE RESEARCH

6.1 Conclusion

Basically, there are three types of algorithms to implement the outer-theta-join operation. The three algorithms are 1) nested loop, 2) sort/nested loop, and 3) sort/merge methods. The nested loop, with no dynamic storage, is considered to be the worst by many people [28] because of the nature of the algorithm. But with dynamic storage, the nested loop can be very good (Figure 7), especially for the non-equal condition(s) type of outerjoin queries, even though the number of comparisons remains the same for all queries (Figure 3). The big saving is in the number of inputs for the operations (Figure 2). Figure 4 shows the percentage of inputs saved when the blocksize is increased from 2 to 100 tuples, for relations REL1 and REL2 presented in appendix A. The sort/nested loop algorithm produces the most consistent results for all the queries; that is, the number of inputs, the number of comparisons and C.P.U. time increase almost proportionally with respect to the number of joins. It is definitely much better than the nested loop algorithm (Figure 1), especially when you have a small number of joined tuples or unique key attribute(s) on outer-equal-join queries. (When the join condition results in cartesian product of the two relations, the sort/nested loop method produces the same results as the nested loop algorithm.) The sort/nested loop algorithm reduces the unnecessary passes through the tuples that have no possibility of making the join condition(s). Therefore, sort/nested loop has fewer inputs and comparisons than the nested loop algorithm. The overhead for sorting is 1) select attributes to be sorted on, 2) sort the relations, and 3) build the necessary predicates for the algorithm. Unless the relations are very small, the sort/nested loop is faster with the overheads involved in setting up the relations. Especially with today's sorting algorithms, the relations can be sorted in the order of n(log n) [19].

The sort/merge outer-join algorithm uses the idea of sort/merge algorithm [19]-that is, it performs like the sort/nested loop outer-join algorithm. The difference is that when it finds the first possible matching tuples from the outer and inner relations, it keeps all the inner tuples that are likely to match the current outer tuple as the intermediate tuples. These intermediate tuples are then used against the remaining outer tuples that are equal to the current outer tuple or the outer tuples that are likely to match them. This saves the k outer remaining tuples that are likely to match from going through the same inner tuples again. For queries with multiple predicates of different conditions, it is necessary to go through the same intermediate tuples, although only partially in this case. This is because the intermediate tuples have satisfied the 'FIRST CONDITION' and only the 'SECOND CONDITION' is not known yet. This algorithm works well for outer-equal-join queries with duplicate tuples in both relations. For queries of relations with no duplicate tuples, both sort/nested loop and sort/merge produce the same results in terms of inputs and comparisons. The sort/nested loop algorithm produces a better C.P.U. time than sort/merge algorithm if both produce the same results for inputs and comparisons, because sort/merge algorithm has a higher overhead than the sort/nested loop algorithm.

Since there is no such thing as an infinite amount of main memory, a fixed amount of main memory is used to improve the outerjoin algorithms. The relation is blocked according to the amount of main memory available (equal to the blocksize) for the operation. By blocking either the outer or inner relations, the nested loop algorithm seems to improve the most on the number of inputs and C.P.U. time. The number of comparisons stays the same because of the presence of nested loops. Blocking on outer relation for sort/nested loop and sort/merge algorithms does not seem to help the algorithms (Figure 7). Instead, it worsens (in general, except maybe for outer-equal-join queries) the results due to the number of times the process is performed (one block of outer relation per inner relation). The number of times the process is performed can be easily reduced to one for sort/nested loop. This is similar to using the algorithms without any blocking or main storage. Therefore, blocking on outer relation is not recommended for sort/nested loop or sort/merge algorithms. On the other hand, blocking on inner relation does improve the sort/nested loop and sort/merge algorithms (Figure 8). However, the number of processes required to perform the outerjoin operation is equal to the size of inner relation divided by the blocksize. Since the blocking is on inner relation, the intermediate tuples are in memory and reduce the number of inputs that are normally required for each reference to an intermediate tuple with no blocking on the inner relation. This is why blocking on inner relation is better than blocking on outer relation for sort/nested loop and sort/merge algorithms.

If the size of the inner relation is equal to the blocksize, then performing outerjoin queries using sort/merge is the best. If the size of the inner relation is greater than the blocksize, the number of processes required to perform the outerjoin is greater than one using the sort/nested loop or sort/merge blocking inner relation algorithms. With some modifications to these algorithms, the number of processes can be reduced to one. The modification is not easy because of the condition that exists when the intermediate tuples are split into different blocks. It is rather difficult to back up to the previous block and hold the current block. If the number of intermediate tuples is less than or equal to the blocksize the window blocking method can be used to move the active block as neccessary.

There is another thing that we can do with the relations in sorted order. That is, we can perform a quick check on the two relations to see whether there is any possibilities for the two relations to have joined tuples. If we determine that it is not possible to have any joined tuple, then we do not have to go through the operation. Instead, we can just join all the tuples from the outer and inner relations with null values. In this way, we have the minimum number of inputs (i.e. the number of outer and inner relations) and zero comparison.

Lastly, we conclude that there is no one best algorithm for the outer-theta-join operations with multiple predicates. The choice of the algorithm depends on (1) join attributes (unique or non-unique), (2) join condition(s), (3) the number of resulted joined tuples, if it can be predicted, and (4) the size of the relations. Our recommendation for choosing the type of algorithms to a query is summarized in Appendix G.

6.2 Future Research

In this thesis we only perform tests on two tables/relations and a blocksize of 50 tuples. There are other tests required to provide a reliable recommendation. These tests include:

1) using different relations, but holding the relation sizes constant;
2) using relations of different sizes;
3) using relations of different sizes and holding the output constant;
4) using different blocksizes;
5) holding the relation size constant, and changes the output for the same query.

BIBLIOGRAPHY

[1] Aho, A.H, C. Beeri, and J. Ullman, The Theory of Joins in relational Data Bases , Proc. 19th IEEE symp. on Foundation of Computer Sci., (1977) 107-113.

[2] Babb, E., "Implementing a Relational Database by Means of Specialized Hardware." ACM Transactions on Database Systems 4, 1 (March 1979), 1-29.

[3] Blasgen, M.W, and Eswaran, K.P., "Storage and Access in Relational Databases." IBM Syst.J. 16,4 (1977), 363-378.

[4] Boral, H., D.J. DeWitt, D. Friedland, and W.K. Wilkinson, "Parallel algorithms for the execution of relational operations." Tech. Rep. 402, Computer Science Dept., Univ. of Wisconsin, Madison, January 1980.

[5] Breitbart, y., P. Olson, and G. Thompson, "Database Integration in a Distributed Heterogenous Database System", Int'l Conf. on Data Engineering, Los Angeles, Ca, February 1986, 301-310.

[6] Ceri,S., and G. Pelagatti, "Correctness of Query Execution Strategies in Distributed Databases", ACM Transactions on Database Systems 8, 4 (December 1983), 577-607.

[7] Chamberlin, D.D., "A Summary of User Experience with the SQL Data Sublanguage", Proc. Int'l. Conf. on Database, Aberdeen, Scotland, July 1980, 181-203.

[8] Chapman, J.A., "Logical Data Base Design for Relational Data Base Systems", Naval Postgraduate School, Monterey, California, June 1978.

[9] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", ACM Transactions on Database Systems 13, 6 (June 1970), 377-387.

[10] Codd, E.F., "Extending the Database Relational Model to Capture More Meaning", ACM Transactions on Database Systems 4, 4 (December 1979), 397-434.

[11] Date, C.J., An Introduction to Database Systems, Third Edition, Addison-Wesley, 1981.

[12] Date, C.J., "The Outer Join", Proc. Second Int'l. Conf. on Databases, Cambridge, England, September 1983, 76-106.

[13] Dayal, U., "Processing Queries with Quatifiers: A Hoticulture Approach", PODS, Atlanta, Georgia, March 1983, 342-353.

[14] Heath, I.J. IBM internal memo (April 1971).

[15] Hsiao, C.C., "Highly Parallel Processing of Relational Databases", Purdue University, West Lafayette, Indiana, August 1983.

[16] Kim, W., "On Optimizing a SQL-like Nested Query", ACM Transactions on Database Systems 7, 3 (September 1982), 443-469.

[17] Kim, W., D. Reiner, and D. Batory, eds., Query Processing in Database Systems, Springer-Verlag, 1984.

[18] Kim, W., D. Reiner, and D. Batory, eds., Query Processing in Database Systems, Springer-Verlag, 1985.

[19] Knuth, D.E. The Art of Computer Programming vol. 3: Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.

[20] Levy, L.S. Discrete Structures of Computer Science , John Wiley & Sons Inc., New York, 1980.

[21] Lacroix, M., and A. Pirotte, "Generalized Joins", SIGMOD Record 8, 3 (September 1976) 14-15.

[22] Lohman, G., D. Daniels, L. Haas, and R. Kistler, Selinger, P., "Optimization of Nested Queries in a Distributed Relational Database", in VLDB 1984.

[23] Maier, D., The Theory of Relational Databases , Computer Science Press, Rockville, Maryland, 1983.

[24] Rosenthal, A., D. Reiner, Extending the Algebraic Framework of Query Processing to Handle Outerjoins, Computer Corp. of America, Cambridge, Massachusetts, 1984.

[25] Sacco, G.M. "Fragementation: A Technique for Efficient Query Processing", ACM Transactions on Database Systems 11, 2 (June 1986), 113-133.

[26] Selinger, P.G., "Access Path Selection in Distributed Database Management Systems", IBM Research Laboratory, San Jose, California, 1980.

[27] Ullman, J.D. Principles of Database Systems. 2nd Ed., Rockville, Maryland, 1976.

[28] Valduriez, P., and G. Gardarin, "Join and semijoin Algorithms for Multiprocessor Database Machine." ACM transactions on Database Systems 9, 1 (March 1984), 133-161.

[29] Vankirk, Dale., "User-friendly interface to The Roth Relational Database", Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 1983.

APPENDIXES

APPENDIX A

TABLE1 AND TABLE2 RELATIONS

RELATION TABLE1

A	B	C	D	E	F
452	970	71	852	457	975
179	925	567	548	723	811
980	835	34	763	577	80
394	817	264	373	557	624
38	88	917	476	500	394
806	619	51	646	879	448
463	29	61	673	67	274
26	56	262	812	666	813
571	195	538	263	960	128
739	636	639	972	185	27
68	37	768	362	376	525
350	34	922	471	72	232
219	509	935	804	922	815
140	322	125	82	136	168
658	849	852	697	629	42
95	122	904	75	560	160
773	334	432	70	489	417
58	169	924	941	640	411
45	294	424	130	446	143
368	535	626	70	729	53
267	157	571	168	203	879
632	648	311	145	540	108
40	822	840	641	803	602
798	331	936	541	452	668
458	120	885	468	345	173
174	392	535	521	6	661
378	667	404	56	772	372
558	797	667	489	203	402
196	868	536	110	117	695
129	964	806	99	806	161
521	379	957	734	131	682
259	196	63	495	265	207
661	774	840	878	684	63
903	799	162	459	449	395
330	737	326	637	854	512
718	900	73	756	818	732
473	531	661	459	108	269
572	518	200	452	209	184
620	34	665	770	477	437
603	892	537	491	887	900
683	131	678	68	143	215
466	623	697	303	11	228

A	B	C	D	E	F
604	7	664	983	532	413
503	685	227	665	969	77
897	827	66	109	892	84
256	650	692	677	428	372
101	635	737	674	854	161
291	235	366	0	143	290
842	652	563	229	454	748
64	603	822	462	133	138
505	281	164	850	725	980
526	851	36	162	271	941
178	182	426	590	803	815
344	379	318	516	86	524
552	593	306	552	469	167
596	740	640	234	768	400
706	695	44	606	531	682
612	461	285	39	536	495
163	284	459	285	955	913
104	896	202	371	7	231
139	561	480	891	156	201
248	631	991	899	749	12
373	284	555	337	335	22
879	471	916	85	651	635
469	93	119	766	302	94
445	541	851	605	278	659
130	278	748	468	655	17
879	536	572	904	863	267
207	765	185	134	748	737
51	650	293	652	101	647
968	617	868	977	343	605
953	207	859	408	384	428
489	814	137	740	353	318
268	543	497	246	650	184
565	604	757	145	578	527
491	149	537	642	493	627
73	208	936	793	858	377
240	623	6	400	697	278
161	150	472	51	522	784
940	455	493	895	838	530
196	0	194	973	563	710
505	175	215	410	129	765
862	814	368	761	435	230
201	900	660	865	143	270
317	45	189	813	874	660
331	808	737	698	590	915
819	211	58	895	664	545
899	569	824	194	789	139
725	100	761	447	733	772
920	272	636	38	1	260
272	795	241	573	507	589
522	865	297	374	274	659
33	296	84	743	915	832
494	652	545	152	957	857
481	23	411	726	950	114
449	880	466	39	542	18
536	987	305	804	207	276
246	929	654	685	991	917
525	890	441	787	281	872
836	158	224	617	192	890

RELATION TABLE2

U	V	W	X	Y	Z
604	7	664	983	532	413
503	685	227	665	969	77
897	827	66	109	892	84
256	650	692	677	428	372
101	635	737	674	854	161
291	235	366	0	143	290
842	652	563	229	454	748
64	603	822	462	133	138
505	281	164	850	725	980
526	851	36	162	271	941
178	182	426	590	803	815
344	379	318	516	86	524
552	593	306	552	469	167
596	740	640	234	768	400
706	695	44	606	531	682
612	461	285	39	536	495
163	284	459	285	955	913
104	896	202	371	7	231
139	561	480	891	156	201
248	631	991	899	749	12
373	284	555	337	335	22
879	471	916	85	651	635
469	93	119	766	302	94
445	541	851	605	278	659
130	278	748	468	655	17
879	536	572	904	863	267
207	765	185	134	748	737
51	650	293	652	101	647
968	617	868	977	343	605
953	207	859	408	364	428
489	814	137	740	353	318
268	543	497	246	650	194
565	604	757	145	578	527
491	149	537	642	493	627
73	208	936	793	858	377
240	623	6	400	697	278
161	150	472	51	522	784
940	455	493	895	838	530
196	0	194	973	563	710
505	175	215	410	129	765
862	814	368	761	435	230
201	900	660	865	143	270
317	45	189	813	874	660
331	808	737	698	590	915
819	211	58	895	664	545
899	569	824	194	789	139
725	100	761	447	733	772
520	272	636	38	1	260
272	795	241	573	507	589
522	865	297	374	274	659
33	295	84	743	915	532
494	652	545	152	957	857
481	23	411	726	950	114
449	880	466	39	542	18
536	987	305	804	207	276
246	929	654	685	991	917
525	890	441	787	281	872
836	158	224	617	192	890

U	V	W	X	Y	Z
452	970	71	852	457	975
179	925	567	548	723	811
980	835	34	763	577	80
394	817	264	373	557	624
38	88	917	476	500	394
806	619	51	646	879	448
463	29	61	673	67	274
26	56	262	812	666	813
571	195	538	263	960	128
739	838	639	972	185	27
68	37	768	362	376	525
350	34	922	471	72	232
219	509	935	804	922	815
140	322	125	82	136	166
658	849	852	697	629	42
95	122	904	75	560	160
773	334	432	70	489	417
58	169	924	941	640	411
45	294	424	130	446	143
368	535	626	70	729	63
267	157	571	168	203	879
632	648	311	145	540	108
40	822	840	641	803	602
798	331	936	541	452	668
458	120	885	468	345	173
174	392	535	521	6	661
378	667	404	56	772	372
558	797	667	489	203	402
196	868	536	110	117	695
129	964	806	99	806	161
521	379	957	734	131	682
259	196	63	495	265	207
661	774	840	878	684	63
903	799	162	459	449	385
330	737	326	637	854	512
718	900	73	756	818	732
473	531	661	459	108	269
572	518	200	452	209	184
620	34	665	770	477	437
603	892	537	491	887	900
683	131	678	68	143	215
466	623	697	303	11	228
583	734	651	998	331	854
954	645	226	733	181	851
637	210	529	456	916	627
620	562	93	76	210	140
90	511	579	238	659	577
576	379	778	481	797	253
713	769	836	283	471	337
114	168	858	528	229	998
866	657	306	22	604	807
310	685	492	242	802	1
933	275	93	979	200	944
272	480	725	331	82	213
541	868	94	444	71	486
175	532	4	936	692	277
455	130	439	366	144	850
751	931	812	665	765	788

U	V	W	X	Y	Z
576	403	310	472	465	16
389	646	842	187	721	994
930	522	304	168	165	666
494	914	600	195	529	939
63	621	10	639	306	355
266	262	849	977	110	612
748	646	924	885	0	5
562	725	805	418	26	106
405	889	398	115	717	494
980	255	483	684	130	333
147	301	739	550	185	634
843	882	110	385	321	327
858	871	154	142	786	300
216	642	995	494	57	658
200	864	661	258	857	14
634	810	783	799	795	995
966	181	648	853	804	743
683	463	478	181	4	341
681	637	220	166	429	210
95	912	161	485	910	986
856	427	11	465	381	491
514	32	792	273	630	693
802	571	582	877	140	174
586	11	17	243	853	161
416	614	104	841	432	882
533	569	370	584	720	284
829	362	444	191	485	406
867	162	797	66	59	249
346	934	341	437	40	274
398	204	886	124	643	1
564	922	804	235	509	990
188	848	731	740	603	97
83	921	403	614	196	767
624	351	813	386	684	496

APPENDIX B

SORT/NESTED LOOP METHOD

(BLOCKING OUTER RELAITON)

```
/* for simpilicity the following is assumed.
 * outer relation is sorted based on x.
 * inner relation is sorted based on y.
 * outer - outer relation.
 * inner - inner relation.
 * outer.x - the 'FIRST CONDITION' attributes for outer
 *           relation
 * inner.y - the 'FIRST CONDITION' attributes for inner
 *           relation
 * R       - Relation, i.e xRy, for the 'FIRST CONDITION'
 * RR      - Relation for the 'SECOND CONDITION'
 * n_outer - total number of outer tuple.
 * p_outer - pth outer tuple.
 * p_inner - pth inner tuple.
 * inner.used - indicate whether inner tuple used or not.
 * used(i) - indicate whether outer ith tuple used or not.
 */
  GET outer;
  DO UNTIL( not eor(outer));
    /*  find the first matching tuple of outer in
        inner relation */
    p_inner = 1;
    GET inner(p_inner);
    n_outer = 0;
    DO I = 1 TO blksize WHILE( not eor(outer) );
       n_outer = n_outer + 1;
       used(I) = false;
       GET outer(I);
    END;

    p_inner = 1;
    p_outer = 1;
```

```
  DO UNTIL( eor(inner) & p_outer > n_outer );
   DO WHILE( not (eor(outer) or p_outer < n_outer)
     or outer(p_outer).x R inner.y) );
     IF outer(p_outer).x < inner.y or
        R is '>' or '>=' THEN
        IF not used(p_outer) then
             output (outer || nulls );
        p_outer = p_outer + 1;
     END;
     ELSE IF outer(p_outer).x > inner.y THEN
           IF eor(outer) and inner.used then
              output (nulls || inner):
        p_inner = p_inner + 1;
        READ inner(p_inner);
     END;
   END;
   IF  not (eor(outer) or eor(inner) ) THEN DO;
     curp_inner = p_inner;
     tpl_not_used = true;
     DO UNTIL( eor(inner) or
        not outer(p_outer).x R inner.y );
        IF outer RR inner  THEN DO;
           output (outer || inner);
           mark inner(p_inner) used;
           tpl_not_used = false;
        END;
        p_inner = p_inner + 1;
        GET inner(p_inner);
     END;
     IF tpl_not_used THEN
        output (outer || nulls);
     p_inner = curp_inner;
     GET inner(p_inner);
     GET outer;
   END;
   ELSE IF p_outer <= n_outer THEN
     DO WHILE ( p_outer <= n_outer );
        IF not used(p_outer) THEN
           output (outer || nulls);
        p_outer = p_outer + 1;
     END;
     ELSE IF eor(outer) and not eor(inner) then
       DO WHILE( not eor(inner) );
          IF ( not inner.used ) THEN
             output (nulls || inner);
          GET inner;
       END;
     END;
     ELSE IF not ( eor(inner) and eor(outer) ) THEN
         reset eor(inner);
  END;
END;
```

APPENDIX C

SORT/NESTED LOOP METHOD

(BLOCKING INNER RELATION)

```
/* for simpilicity the following is assumed.
 * outer relation is sorted based on x.
 * inner relation is sorted based on y.
 * outer - outer relation.
 * inner - inner relation.
 * outer.x - the 'FIRST CONDITION' attributes for outer
 *           relation
 * inner.y - the 'FIRST CONDITION' attributes for inner
 *           relation
 * R      - Relation, i.e xRy, for the 'FIRST CONDITION'
 * RR     - Relation for the 'SECOND CONDITION'
 * n_inner - total number of inner tuples.
 * p_outer - pth outer tuple.
 * p_inner - pth inner tuple.
 * outer.used - indicate whether outer tuple used or not.
 * used(i) - indicate whether inner ith tuple used or not.
 */
 GET inner;
 DO UNTIL( not eor(inner));
  p_outer = 1;
  GET outer(p_outer);
  n_inner = 0;
  DO I = 1 TO blksize WHILE( not eor(inner) );
     n_inner = n_inner + 1;
     used(I) = false;
     GET inner(I);
  END;

  p_outer = 1;
  p_inner = 1;
```

```
  DO UNTIL( eor(outer) & p_inner > n_inner );
   /*  find the first matching tuple of inner in
       outer relation */
   DO WHILE( not (eor(inner) or p_inner < n_inner)
      or inner(p_inner).x R outer.y) );
      IF outer.x < inner(p_inner).y or
         R is '>' or '>=' THEN
         IF eor(inner) and not outer.used THEN
              output (outer || nulls );
         GET outer;
      END;
      ELSE
         IF eor(inner) and inner(p_inner).used THEN
            output (nulls || inner):
         p_inner = p_inner + 1;
      END;
   END;
   IF  not (eor(inner) or eor(outer) ) THEN DO;
      curp_inner = p_inner;
      tpl_not_used = true;
      DO UNTIL( p_inner = n_inner or
                outer.x R inner(p_inner).y );
         IF outer RR inner  THEN DO;
            output (outer || inner);
            mark inner(p_inner) used;
            tpl_not_used = false;
         END;
         p_inner = p_inner + 1;
      END;
      IF tpl_not_used and eor(inner) THEN
         output (outer || nulls);
      p_inner = curp_inner;
      GET outer;
   END;
   ELSE IF p_inner <= n_inner THEN
      DO WHILE ( p_inner <= n_inner );
         IF not used(p_inner) THEN
            output (nulls || inner);
         p_inner = p_inner + 1;
      END;
      ELSE IF eor(inner) and not eor(outer) then
        DO WHILE( not eor(outer) );
           IF ( not outer.used ) THEN
              output (outer || nulls);
           GET outer;
        END;
      END;
      ELSE IF not ( eor(outer) and eor(inner) ) THEN
          reset eor(outer);
 END;
END;
```

APPENDIX D

SORT/MERGE METHOD (BLOCKING OUTER RELATION)

```
/* for simpilicity the following is assumed.
 * outer relation is sorted based on x.
 * inner relation is sorted based on y.
 * outer - outer relation.
 * inner - inner relation.
 * outer.x - the 'FIRST CONDITION' attributes for outer
 *            relation
 * inner.y - the 'FIRST CONDITION' attributes for inner
 *            relation
 * R      - Relation, i.e xRy, for the 'FIRST CONDITION'
 * RR     - Relation for the 'SECOND CONDITION'
 * n_outer - total number of outer tuple.
 * p_outer - pth outer tuple.
 * p_inner - pth inner tuple.
 * inner.used - indicate whether inner tuple used or not.
 * used(i) - indicate whether outer ith tuple used or not.
 */
GET outer;
DO UNTIL( not eor(outer));
  p_inner = 1;
  GET inner(p_inner);
  n_outer = 0;
  DO I = 1 TO blksize WHILE( not eor(outer) );
     n_outer = n_outer + 1;
     used(I) = false;
     GET outer(I);
  END;

  p_inner = 1;
  p_outer = 1;
  DO UNTIL( eor(inner) & p_outer > n_outer );
/*  find the first outer tuple and */
/*  inner tuple that statisfied    */
/*  the relation R.                */
```

```
    DO WHILE( not (eor(outer) or p_outer < n_outer)
       or outer(p_outer).x R inner.y) );
      IF outer(p_outer).x < inner.y or
         R is '>' or '>=' THEN
         IF not used(p_outer) then
            output (outer || nulls );
         p_outer = p_outer + 1;
      END;
      ELSE IF outer(p_outer).x > inner.y THEN
         IF eor(outer) and inner.used then
            output (nulls || inner):
         p_inner = p_inner + 1;
         GET inner(p_inner);
      END;
    END;
    IF  not (eor(outer) or eor(inner) ) THEN DO;
       start_inner = p_inner;
       tpl_not_used = true;
       DO UNTIL( eor(inner) or not outer(p_outer).x R
                 inner.y );
          IF outer RR inner  THEN DO;
             output (outer || inner);
             mark inner(p_inner) used;
             tpl_not_used = false;
          END;
          p_inner = p_inner + 1;
          GET inner(p_inner);
       END;
       IF not outer(p_outer).used THEN
          output (outer || nulls);
/*                                     */
/* Is the next remaining outer tuple   */
/* equal to the current outer tuple    */
/* Or satisfies the 'WHERE' evaluation */
/*                                     */
       p_outer = p_outer + 1;
       outer(p_outer).used = false;
       more = false;
       DO WHILE( not eor(outer) and ( outer.x = current.x
           or WHERE(FIRST))) ;
           MORE = true;
           IF no SECOND CONDITION THEN
              DO I = start_inner TO p_inner-1;
                 GET inner(i);
                 output (outer(p_outer) || inner);
                 mark inner used;
              END;
              outer(p_outer).used = true;
           END;
           ELSE
           DO I = start_inner TO p_inner-1;
              GET inner(I);
              IF WHERE(SECOND) THEN DO;
                 output (outer(p_outer) || inner);
```

```
                 mark inner used;
                 outer(p_outer).used = true;
              END;
           END;
           IF not outer(p_outer).used then
              output (outer(p_outer) || nulls);
           p_outer = p_outer + 1;
           outer(p_outer).used = false;
        END;
/*                                     */
/* Determine whether to advance to new */
/* inner tuple or back to the          */
/* start_inner + 1 inner tuple         */
/*                                     */
        ADVAN = true;
        IF ( not eor(outer) and R is not '=' and
                    (p_inner-1) > start_inner ) THEN DO;
           IF WHERE(FIRST) on lookahd THEN DO;
             p_inner = start_inner + 1;
             reset eor(inner);
             ADVAN = false;
           END;
           GET inner(p_inner);
        END;
/*                                     */
/* join all unused inner tuples with   */
/* nulls tuple.                        */
/*                                     */
        IF no second condition and ADVAN is true THEN DO;
           DO I = start_inner TO p_inner-1;
              GET inner(I);
              IF not inner.used THEN
                 output (nulls || inner);
           END;
           GET inner(p_inner);
        END;
    END;
    ELSE IF p_outer <= n_outer THEN
       DO WHILE ( p_outer <= n_outer );
          IF not used(p_outer) THEN
             output (outer || nulls);
          p_outer = p_outer + 1;
       END;
       ELSE IF eor(outer) and not eor(inner) then
         DO WHILE( not eor(inner) );
            IF ( not inner.used ) THEN
               output (nulls || inner);
            GET inner;
         END;
       END;
       ELSE IF not ( eor(inner) and eor(outer) ) THEN
           reset eor(inner);
   END;
END;
```

APPENDIX E

SORT/MERGE METHOD (BLOCKING INNER RELATION)

```
/* for simpilicity the following is assumed.
 * outer relation is sorted based on x.
 * inner relation is sorted based on y.
 * outer - outer relation.
 * inner - inner relation.
 * outer.x - the 'FIRST CONDITION' attributes for outer
 *           relation
 * inner.y - the 'FIRST CONDITION' attributes for inner
 *           relation
 * R     - Relation, i.e xRy, for the 'FIRST CONDITION'
 * RR    - Relation for the 'SECOND CONDITION'
 * n_inner - total number of inner tuples.
 * p_outer - pth outer tuple.
 * p_inner - pth inner tuple.
 * outer.used - indicate whether outer tuple used or not.
 * used(i) - indicate whether inner ith tuple used or not.
 */
GET inner;
DO UNTIL( not eor(inner));
  p_outer = 1;
  GET outer(p_outer);
  n_inner = 0;
  DO I = 1 TO blksize WHILE( not eor(inner) );
     n_inner = n_inner + 1;
     used(I) = false;
     GET inner(I);
  END;

  p_outer = 1;
  p_inner = 1;
  DO UNTIL( eor(outer) & p_inner > n_inner );
/*  find the first outer tuple and */
/*  inner tuple that statisfied    */
/*  the relation R.                */
```

```
  DO WHILE( not (eor(inner) or p_inner < n_inner)
     or inner(p_inner).x R outer.y) );
     IF outer.x < inner(p_inner).y or
        R is '>' or '>=' THEN
        IF eor(inner) and not outer.used THEN
           output (outer || nulls );
        GET outer;
     END;
     ELSE DO;
       IF eor(inner) and inner(p_inner).used THEN
          output (nulls || inner):
       p_inner = p_inner + 1;
    END;
    END;
    IF  not (eor(inner) or eor(outer) ) THEN DO;
       start_inner = p_inner;
       outer.used = true;
       DO UNTIL( p_inner = n_inner or
                 outer.x R inner(p_inner).y );
          IF outer RR inner  THEN DO;
             output (outer || inner);
             mark inner(p_inner) used;
             outer.used = false;
          END;
          p_inner = p_inner + 1;
       END;
       IF outer.used and eor(inner) THEN
          output (outer || nulls);
/*                                      */
/* Is the next remaining outer tuple    */
/* equal to the current outer tuple     */
/* Or satisfies the 'WHERE' evaluation */
/*                                      */
       GET outer;
       outer.used = true;
       more = false;
       DO WHILE( not eor(outer) and ( outer.x = current.x
           or WHERE(FIRST))) ;
           MORE = true;
           IF no SECOND CONDITION THEN
              DO I = start_inner TO p_inner-1;
                 output (outer || inner(I));
                 mark inner used;
              END;
              outer.used = true;
           END;
           ELSE
           DO I = start_inner TO p_inner-1;
              IF WHERE(SECOND) THEN DO;
                 output (outer || inner(I));
                 mark inner used;
                 outer.used = true;
              END;
           END;
```

```
           IF not outer.used then
              output (outer || nulls);
           GET outer;
           outer.used = false;
        END;
/*                                     */
/* Determine whether to advance to new */
/* inner tuple or back to the          */
/* start_inner + 1 inner tuple         */
/*                                     */
        ADVAN = true;
        IF ( not eor(outer) and R is not '=' and
                    (p_inner-1) > start_inner ) THEN DO;
           IF WHERE(FIRST) on lookahd THEN DO;
             p_inner = start_inner + 1;
             ADVAN = false;
           END;
        END;
/*                                     */
/* join all unused inner tuples with   */
/* nulls tuple.                        */
/*                                     */
        IF no second condition and ADVAN is true THEN DO;
           DO I = start_inner TO p_inner-1;
              IF not inner(I).used THEN
                 output (nulls || inner(I));
           END;
        END;
       p_inner = start_inner;
    END;
    ELSE IF p_inner <= n_inner THEN
       DO WHILE ( p_inner <= n_inner );
          IF not used(p_inner) THEN
             output (nulls || inner);
          p_inner = p_inner + 1;
       END;
       ELSE IF eor(inner) and not eor(outer) then
         DO WHILE( not eor(outer) );
            IF ( not outer.used ) THEN
               output (outer || nulls);
            GET outer;
         END;
       END;
       ELSE IF not ( eor(outer) and eor(inner) ) THEN
           reset eor(outer);
 END;
END;
```

APPENDIX F

FUNCTION TO CHECK FOR POSSIBLE JOINS IN THE RELATIONS

```
/*
 * Function: Check possible join
 *
 * outer.x - attributes of the outer relation
 * inner.y - attributes of the outer relation
 * ret_bit - return bit
 */
 Get outer;
 Get inner;
 ret_bit = false;
 IF 'FIRST CONDITION' is true THEN
     ret_bit = true;
 ELSE IF '=' condition THEN DO;         /*   =      */
    IF outer.x < inner.y  THEN DO;  /* DATA1 < DATA2 */
       Get last outer
      /* IS LAST DATA1 >= FIRST DATA2 THEN POSSIBLE JOIN */
       IF outer.x >= inner.y THEN
           ret_bit = true;
    END;
    ELSE DO;     /* DATA1 > DATA2 */
       Get last inner
      /* IS FIRST DATA1 <= LAST DATA2 THEN POSSIBLE JOIN */
       IF outer.x <= inner.y THEN
           ret_bit = true;
    END;
 END;
 ELSE IF '<' ('<=') condition THEN DO; /* <,  <= */
    Get last inner
    IF outer.x < inner.y (outer.x <= inner.y) THEN
       ret_bit = true;
 END;
 ELSE IF '>' ('>=') condition THEN DO; /* >,  >= */
    Get last outer
    IF outer.x > inner.y (outer.x >= inner.y) THEN
       ret_bit = 'true;
 END;
 RETURN(ret_bit);

END Function
```

APPENDIX G

RECOMMENDED METHODS

a) No dynamic storage.

Conditions	Recommeded Methods
1. Equal join conditions	sort/merge outer-equal-join
2. Unique key with one predicate.	sort/nested loop
3. Non-unique key with one predicate.	sort/merge
4. Unique key with multiple predicates.	sort/nested loop
5. Non-unique key with multiple predicates.	sort/merge

b) Blocksize greater than one but less than both relations.

Conditions	Recommeded Methods
1. Equal join conditions	sort/merge outer-equal-join
2. Unique key with one predicate and θ condition.	nested loop
3. Non-unique key with one predicate and θ condition.	nested loop
4. Unique key with multiple predicates and '=' FIRST CONDITION.	sort/nested loop
5. Non-Unique key with multiple predicates and '=' FIRST CONDITION.	sort/merge
6. Other conditions.	nested loop

c) Blocksize greater than either relations.
(Note: make the smaller relation the inner relation)

Conditions	Recommeded Methods
1. Equal join conditions	sort/merge outer-equal-join
2. Unique key with '=' as the FIRST CONDITION.	sort/nested loop
3. Other conditions.	sort/merge

APPENDIX H

EXAMPLE OF ONE QUERY CAN BE DONE IN TWO WAYS

Let the outerjoin query be

ojoin (T1,T2) where T1.a > T2.a

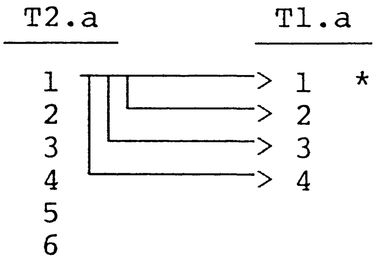
Let T1 and T2 be the following relations:

T1	T2
1	1
2	2
3	3
4	4
	5
	6

First, look at the query ojoin (T1,T2) where T2.a < T1.a

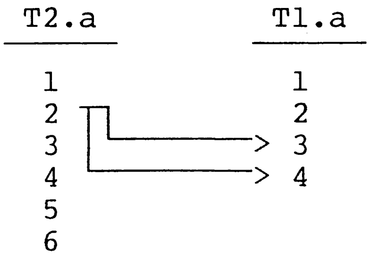
(T1 and T2 sorted in ascending order)

T1.a	T2.a
1	1
2	2
3	3
4	4
	5
	6



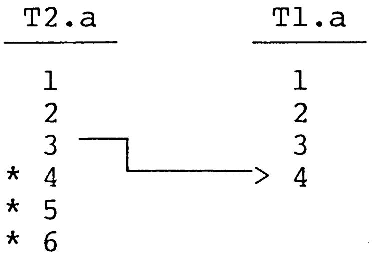
===>

T1.a >	T2.a
1	?
2	1
3	1
4	1



===>

T1.a >	T2.a
1	?
2	1
3	1
4	1
3	2
4	2



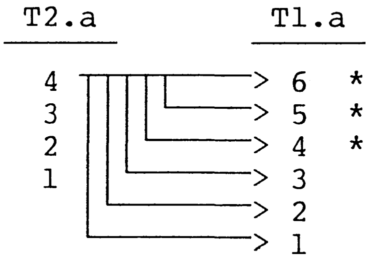
===>

T1.a >	T2.a
1	?
2	1
3	1
4	1
3	2
4	2
4	3
?	4
?	5
?	6

Now, we look at the query ojoin (T1,T2) where T1.a > T2.a and see whether it is different from doing ojoin (T1,T2) where T2.a < T1.a

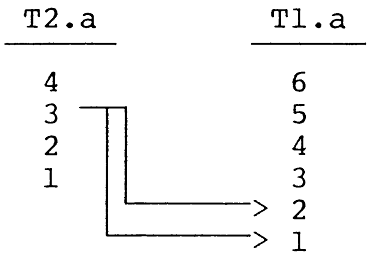
(T1 and T2 are sorted in descending order)

T1	T2
4	6
3	5
2	4
1	3
	2
	1



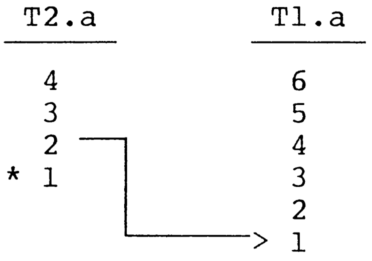
===>

T1.a >	T2.a
?	6
?	5
?	4
4	3
4	2
4	1



===>

T1.a > T2.a	
?	6
?	5
?	4
4	3
4	2
4	1
3	2
3	1



===>

T1.a > T2.a	
?	6
?	5
?	4
4	3
4	2
4	1
3	2
3	1
2	1
1	?

* indicate unmatched tuple which is joined with null tuple.

APPENDIX I

EXAMPLE OF THE SORT/NESTED LOOP OPERATION

The operation of outerjoin on STORE1 and STORE2 where the STORE1.S# = STORE2.S# using sort/nested Loop. Let relation STORE1 and STORE2 of Suppliers & Parts relationship as follow:

STORE1

	S#	P#
(1)	s3	p1
(5)	s5	p3
(9)	s5	p7
(14)	s8	p5
(19)	s9	p2

STORE2

	S#	P#
(2)	s1	p2
(3)	s2	p6
(4,10,15)	s5	p4
(6,11,16)	s5	p9
(7,12,17)	s5	p8
(8,13,18)	s7	p9

The numbers in parenthesis indicate the order of the tuples are read in and compared. For example, (1) is read in first and compared with (2).

Operations		Results of operations
(1) and (2)	==>	? ? s1 p2
(1) and (3)	==>	? ? s2 p6
(1) and (4)	==>	s3 p1 ? ?
(5) and (4)	==>	s5 p3 s5 p4
(5) and (6)	==>	s5 p3 s5 p9
(5) and (7)	==>	s5 p3 s5 p8
(5) and (8)	==>	
(9) and (10)	==>	s5 p7 s5 p4
(9) and (11)	==>	s5 p7 s5 p9
(9) and (12)	==>	s5 p7 s5 p8
(9) and (13)	==>	

```
  (14) and (15)     ==>
  (14) and (16)     ==>
  (14) and (17)     ==>
  (14) and (18)     ==>  ?    ?    s7   p8
  (14)              ==>  s8   p5   ?    ?
  (19)              ==>  s9   p2   ?    ?
```

The resulted output:

STORE1.s#	STORE1.p#	STORE2.s#	STORE2.p#
?	?	s1	p2
?	?	s2	p6
s3	p1	?	?
s5	p3	s5	p4
s5	p3	s5	p9
s5	p3	s5	p8
s5	p7	s5	p4
s5	p7	s5	p9
s5	p7	s5	p8
?	?	s7	p8
s8	p5	?	?
s9	p2	?	?

To see how the sort/merge algorithm is different from the sort/nested Loop algorithm for the above example, see Appendix J.

APPENDIX J

EXAMPLE OF THE SORT/MERGE OPERATION

The operation of outerjoin on STORE1 and STORE2 where the STORE1.S# = STORE2.S# using the Sort/Merge algorithm. Let relation STORE1 and STORE2 of Suppliers & Parts relationship as follow:

STORE1

	S#	P#
(1)	s3	p1
(5)	s5	p3
	s5	p7
(6)	s8	p5
(8)	s9	p2

STORE2

	S#	P#
(2)	s1	p2
(3)	s2	p6
(4)	s5	p4
	s5	p9
	s5	p8
(7)	s7	p9

In order to understand the Sort/Merge algorithm, you have to imagine that the tuples are divided into groups of tuples of the same kind. The groups are then merged together, as follow:

Operations	Results of the operations
(1) & (2) ==>	? ? s1 p2
(1) & (3) ==>	? ? s2 p6
(1) & (4) ==>	s3 p1 ? ?

```
(5)  &  (4)  ==>       s5  p3  s5  p4
                       s5  p3  s5  p9
                       s5  p3  s5  p8
                       s5  p7  s5  p4
                       s5  p7  s5  p9
                       s5  p7  s5  p8
(6)  &  (7)  ==>       ?   ?   s7  p9
(6)          ==>       s8  p5  ?   ?
(8)          ==>       s9  p2  ?   ?
```

The resulted output:

STORE1.s#	STORE1.p#	STORE2.s#	STORE2.p#
?	?	s1	p2
?	?	s2	p6
s3	p1	?	?
s5	p3	s5	p4
s5	p3	s5	p9
s5	p3	s5	p8
s5	p7	s5	p4
s5	p7	s5	p9
s5	p7	s5	p8
?	?	s7	p8
s8	p5	?	?
s9	p2	?	?

APPENDIX K

NESTED LOOP OUTERJOIN ALGORITHM

```
/*  file1  - contains tuples from outer relation
 *  file2  - contains tuples from inner relation
 *  file3  - output relation
 *  tuple1 - tuple from file1 or outer relation
 *  tuple2 - tuple from file2 or inner relation
 *  last_tuple1 - the last tuple from outer relation
 *                indicator
 *  WHERE  - function to evaluate where clause
 */

   OPEN FILE(file1) INPUT;
   READ FILE(file1) INTO(tuple1);
   DO WHILE( NOT eof1);
      tuple1_used_at_least_once = false;
      OPEN FILE(file2) INPUT;
      READ FILE(file2) INTO(tuple2);
      DO WHILE( NOT eof2 );
         IF WHERE(predicates) THEN DO;
            tuple1_used_at_least_once = true;
            WRITE FILE(file3) FROM(tuple1||tuple2);
            mark tuple2 used in file2;
         END;
         ELSE IF last_tuple1 and
            tuple2 did not mark used THEN DO;
            WRITE FILE(file3) FROM(nulls1||tuple2);
         END;
         READ FILE(file2) INTO(tuple2);
      END;
      IF not tuple1_used_at_least_once THEN
         WRITE FILE(file3) FROM(tuple1||nulls2);
      READ FILE(file1) INTO(tuple1);
   END;
```

VITA

WEN FONG LEE

Candidate for the Degree of

Master of Science

Thesis: PERFORMANCE EVALUATION OF OUTERJOIN OPERATION ON ADDS SYSTEM

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Malaysia, May 26, 1959, the son of Meng Teik Lee. Married to Soon Har Chan on September 14, 1984.

Education: Graduated from Ibrahim Secondary School, Sungai Patani, Kedah, Malaysia, in 1979; received Bachelor of Science in Arts & Sciences degree in Computing and Information Sciences from Oklahoma State University in December, 1983; completed requirements for the Master of Science degree at Oklahoma State University in May, 1987.

Professional Experience: Teaching Assistant, Department of Computing and Information Sciences, Oklahoma State University, August, 1982, to May, 1986; Research Scientist, Amoco Production Company, Tulsa, Oklahoma, May, 1985, to present.