# PERFORMANCE EVALUATION OF

## OUTER JOIN OPERATIONS

# ON ADDS SYSTEM

Ву

WEN FONG LEE

Bachelor of Science in Arts & Sciences

Oklahoma State University

1997 - C

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:

.

.

1 V J Adviser Thesis Ē. He. onald & Fish

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.

iii

# TABLE OF CONTENTS

Cha	pt	e	r
-----	----	---	---

I.	INTRODUCTION	1
	1.2 Approaches	1 4 5
II.	DEFINITIONS AND TERMINOLOGY	7
	51	7 9 0
III.	NESTED LOOP METHOD	4
	<pre>3.1 Nested Loop</pre>	8
IV.	SORT/MERGE METHOD	4
	4.1 Sort/Merge Method for Join	
۷.	ANALYSIS AND EVALUATION OF OUTERJOIN METHODS 3	7
	5.1 No Dynamic Storage	6 2 2
VI.	CONCLUSION AND FUTURE RESEARCH	0
	6.1 Conclusion	-
BIBLIO	GRAPHY	б
APPEND	IXES $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $69$	9

APPENDIX A -	REL1 AND REL2 RELATIONS	69
APPENDIX B -	SORT/NESTED LOOP METHOD (BLOCKING OUTER RELATION)	74
APPENDIX C -	SORT/NESTED LOOP METHOD (BLOCKING INNER RELATION)	76
APPENDIX D -	SORT/MERGE METHOD (BLOCKING OUTER RELATION)	78
APPENDIX E -	SORT/MERGE METHOD (BLOCKING INNER RELATION)	81
APPENDIX F -	FUNCTION TO CHECK POSSIBLE JOIN	84
APPENDIX G -	RECOMMEDED METHODS	85
APPENDIX H -	EXAMPLE OF ONE QUERY CAN BE DONE IN TWO WAYS	86
APPENDIX I -	EXAMPLE OF THE SORT/NESTED LOOP OPERATION	89
APPENDIX J -	EXAMPLE OF THE SORT/MERGE OPERATION	91
APPENDIX K -	NESTED LOOP OUTERJOIN ALGORITHM	93

# LIST OF TABLES

# Table

I		, TABLE2) where A = U storage)	40
II		, TABLE2) where A < U storage)	40
III		, TABLE2) where A > U storage)	41
IV	<b>.</b>	, TABLE2) where A = U AND B < V storage)	41
v		, TABLE2) where A = U relation, blocksize = 50)	47
VI		TABLE2) where A < U relation, blocksize = 50)	47
VII		TABLE2) where A > U relation, blocksize = 50)	48
VIII		TABLE2) where A = U AND B < V relation, blocksize = 50)	48
IX		TABLE2) where A = U relation, blocksize = 50)	52
х	. Ojoin (TABLE1,	TABLE2) where A < U relation, blocksize = 50)	
XI	. Ojoin (TABLEL,	TABLE2) where A > U relation, blocksize = 50)	
XII	. Ojoin (TABLE1,	TABLE2) where A = U AND B < V relation, blocksize = 50)	

# LIST OF FIGURES

# Figure

# Page

1.	C.P.U. time per Join Tuple	43
2.	The Number of Inputs per Join Tuple	44
3.	The Number of Comparisons per Join Tuple	45
4.	The Percentage of Inputs Saved Relatively to Blocksize	48
5.	The Number of Inputs with respect to the Size of the Relations (Block Outer Relation)	49
6.	The Number of Inputs with respect to the Size of	
	the Relations (Block Inner Relation)	50
7.	C.P.U. Time per Tuple (Block Outer Relation)	54
8.	C.P.U. Time per Tuple (Block Inner Relation)	55

#### 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 opera-The performance of the relational database managetions. ment 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

SUPPLIERS

p#	pname	s#
pl	Gear	sl
p2	Nut	s2
p3	Bolt	s4

s#	sname	city
sl	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
pl	Gear	sl	sl	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 ( $\theta$ ) 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- $\theta$ -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
pl	Gear	sl	sl	Ajax	London
p2	Nut	s2	s2	Acme	Paris
р3	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  $D_1, D_2, \ldots, D_n$  be n (n > 0 ) domains (not necessarily distinct). The cartesian product × { $D_i$ : i = 1,2,...,n} is the set of all n-tuples  $\langle t_1, t_2, \ldots, t_n \rangle$  such that  $t_1$  belongs to  $D_1$ ,  $t_2$  belongs to  $D_2, \ldots, t_n$  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, $\nu$ ), where A is an attribute and  $\nu$  is a value drawn from the domain of A, instead of a sequence  $\langle \nu_1, \nu_2, \dots, \nu_n \rangle$  [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 × 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	Т	F	?	OR	Т	F	?	NOT	
Т	Т	F	?	T	T	Т	T	Т	F
F	F	F	F	F	T	F	?	F	T
?	?	F	?	?	T	?	?	?	?

If the WHERE-clause of a query, applied to a row of a table or join, evaluate to the "?" truth-value, the WHEREclause 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 Rl(A,Bl) and R2(B2,C) with attributes Rl.A, Rl.Bl, R2.B2, R2.C. For simplicity we assume that the left to right order of attributes within a relation is significant. Assume that Rl.Bl and R2.B2 may validly be compared with each other. Let theta denote any one of the operators =,  $\neq$ , <, <=, >, >=, that applies to Rl.Bl and R2.B2. Define J to be the theta-join of Rl on Bl 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 Rl and R2; i.e., the attributes of J are A, Bl, B2, and C. And we also assume that

these names are all distinct. Define ~Rl as follows:

$$\widetilde{R}I = RI - J \{A, BI\}$$

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

$$\sim R2 = R2 - J \{B2,C\}$$

Then the outer-theta-join of Rl on Bl with R2 on B2, written

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

is defined to be equal to the expression

J union (  $\operatorname{Rl} \times (?,?)$  ) union (  $(?,?) \times \operatorname{R2}$  )

where "?" denotes the null value, as before, and " $\times$ " denotes the extended cartesian product.

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

J union (  $~Rl \times (?,?)$  )

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

J union ( (?,?) 
$$\times ~$$
 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:

	5	I	<u>.</u>	
S#	CITY		P#	CITY
S1 S2 S3 S4 S5	London Paris ? NY SFO		P1 P2 P3 P4 P5	London Oslo ? NY 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 S2 S3 S4	London Paris ? NY	Pl ? ? P4	London ? ? NY
S5	SFO	?	?

2	?	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 S2 S3 S4 S5	London Paris ? NY SFO	P1 ? ? P4 ?	London ? ? NY ?			

(left outer-theta-join)

	SXP						
S#	S.CITY	P#	P.CITY				
S1 S2 S3 S4 S5 ? ?	London Paris ? NY SFO ? ? ?	P1 ? P4 ? P3 P5	London ? ? NY ? Olso ? 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(filel) INTO(tuplel);
  DO WHILE( NOT eofl);
      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 tuplel used at least once THEN
         WRITE FILE(file3) FROM(tuple1||nulls2);
      READ FILE(filel) INTO(tuplel);
  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 filel 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 ).

0 - Total I/O time for outputs.

 $\Delta$  - Other overhead costs.

Cost = 
$$(N * I) + (N * M * I) + (N * M * E) +$$
  
 $(M * I) + O + \Delta;$  (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 Cost = (N \* I) + (N \* M \* I) + (N \* M \* E) + O +  $\Delta$ ; (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

$$N + M \begin{bmatrix} N \\ B \end{bmatrix}$$
,

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

Cost = 
$$(N * I) + M/B$$
  $(N * I) + (N * M * E) + O + \Delta$ .  
(3.3)

```
/* file1 - contains tuples from outer relation
 *
   file2 - contains tuples from inner relation
*
   file3 - output relation
 *
   eof1 - end of file1 ( initially false)
           - end of file2 ( initially false)
 *
    eof2
 *
   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"
 *
 *
   tuplel used at least once - tuplel 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(filel) INTO(tuplel);
   DO WHILE( NOT eofl);
      count = 0;
      DO WHILE( NOT eofl & count < factor );
         TOTtuple1 = TOTtuple1 - 1;
         count = count + 1;
         tup buf(count) = tuplel;
         tuplel used at least once(count) = false;
         READ FILE(filel) INTO(tuplel);
      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;
               tuplel 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 tuplel 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			STORE2		
S#	P#		S#	P#	
s3 s5 s5 s8 s9	pl p3 p7 p5 p2		s1 s2 s5 s5 s5 s7	p2 p6 p4 p9 p8 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, Rl.a = R2.a and Rl.b = R2.c and Rl.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;
   ΙF
      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.

Cost = 
$$(M + N_i + N_0) * (E + I) + O + \Delta$$
 (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		REL2	
A	В	A	В
2 2	3 2	1 2 2	1 4 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
? 2 2 2 2	<b>?</b> 2 3 3	1 2 2 2 2	1 4 3 4 3

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

$$CI = (M + N) * I + O;$$
 (4.1)

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

CC = (M + N - 1) \* E; (4.2)

$$CC = N^*E; \qquad (4.3)$$

(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-thetajoin 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 * ( Rl, R2 ) where Rl.x = R2.x and
Rl.y = R2.y and
Rl.z < R2.z
```

(assuming that x and y are key attributes) Then

a) FIRST CONDITION is

Rl.x = R2.x and Rl.y = R2.y

b) LESS THAN CONDITION is

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

c) SECOND CONDITION is

Rl.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
         - Relation, i.e xRy, for the 'FIRST CONDITION' - Relation for the 'SECOND CONDITION'
* R
* RR
* 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) ));
     WHERE(LESS) THEN DO;
output(outer | nulls);
  IF
     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 = 'l'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 | T 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 appropriate 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  

$$CI = (M + N_j) * I;$$
 (for j = 1 to M) (4.4)

where N<sub>j</sub> is the intermediate tuples for the jth iteration. The cost of comparisons (CC) depends on the type of outerjoin 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.

 $N*E \leq CC \leq M*N*E$ 

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:

- 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

Method	Number o	of tuples	. 1	Number of		Total	Average
used	outer	inner	inputs	comparisons	joins	output tuples	1
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
	•		•		· · · ·	р <b>я</b>	

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

# TABLE II

. .....

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

Method	Number o	of tuples				Total	Average
used	outer	inner	inputs	comparisons	joins	output tuples	
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

٩.

. . .

# TABLE III

Method	Number c	of tuples	٢	lumber of		Average	
used	outer	inner	inputs	comparisons		output tuples	<i>·</i> · ·
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

,

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

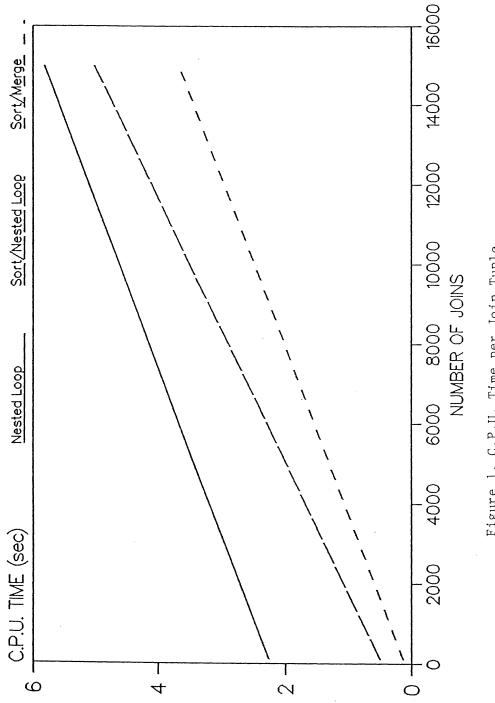
# TABLE IV

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

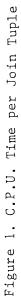
.

Method	Number o	of tuples	Number of			Total	Average
used	outer	inner	inputs	comparisons	joins	output tuples	
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

,



·,



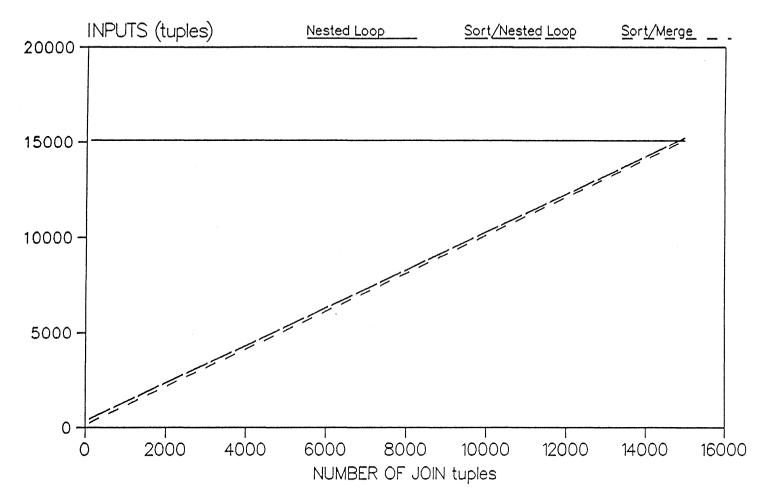


Figure 2. The Number of Inputs per Join Tuple

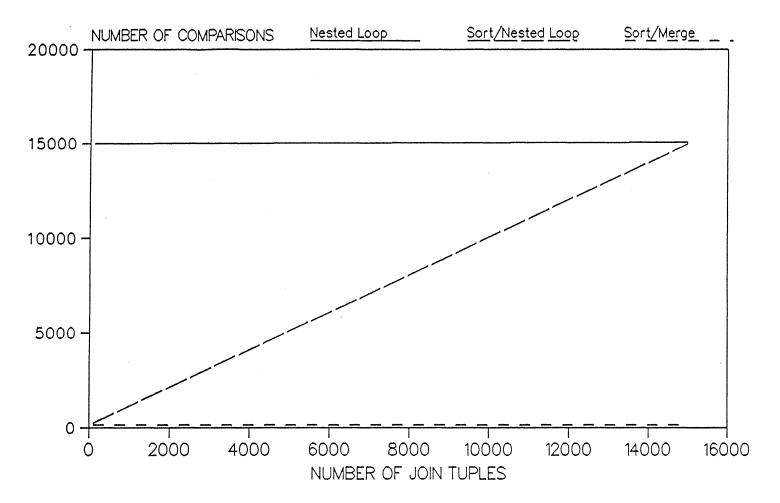


Figure 3. The Number of Comparisons per Join Tuple

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

Method	Number o	of tuples	٤	Number of	Total output	Average C.P.U	
used	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

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

#### TABLE VI

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

Method	Number c	of tuples	Number of ,			Total	Average
used	outer	inner	inputs	comparisons	joins		C.P.U (sec)
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

~

1

### TABLE VII

Method	Number (	of tuples	1	Number of	Total	Average	
used	outer	inner	inputs	comparisons	joins	1 L	C.P.U (sec)
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

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

#### TABLE VIII

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

Method	Number o	of tuples	1	Number of		Total	1
used	outer	inner	inputs	comparisons	joins	output tuples	
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

~

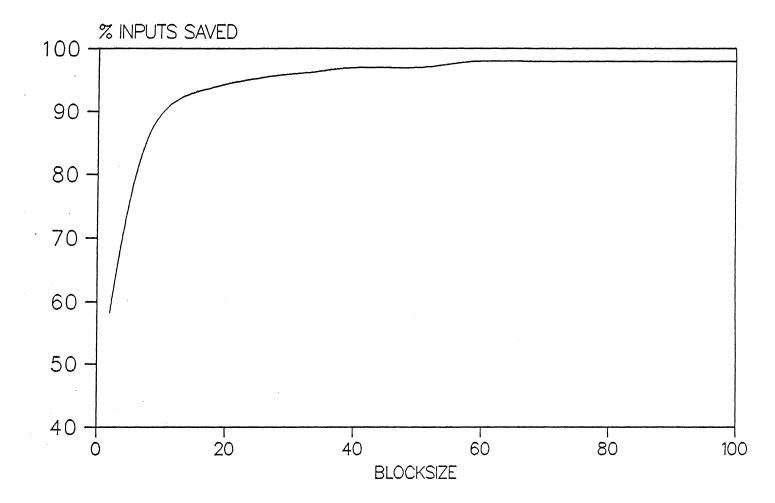
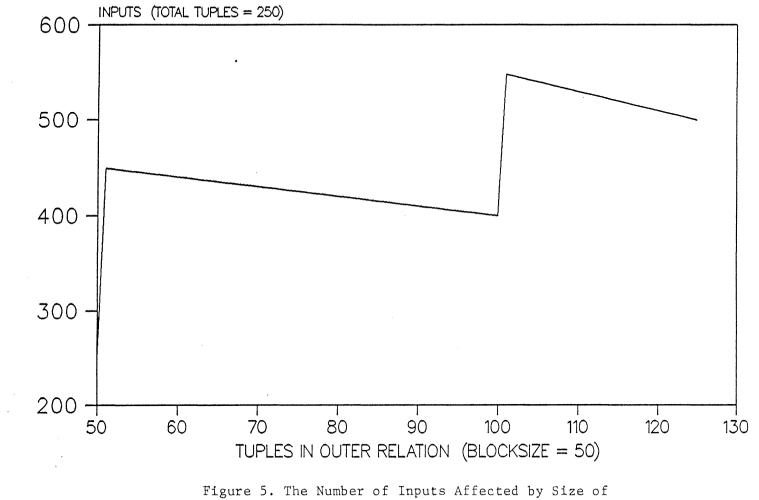
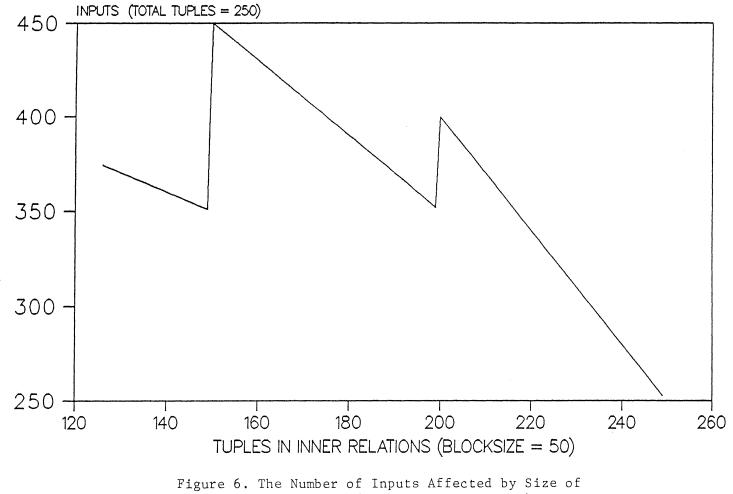


Figure 4. The Percentage of Inputs Saved Relatively to Blocksize



.

Relations (Block Outer Relation)



. \*

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

1

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

Method	Number o	of tuples	ł	Number of		Average	
used	outer	inner	inputs	comparisons		output tuples	· · ·
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

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

# TABLE X

^

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

Method	Number (	of tuples	1	Number of ,			Average
used	outer	inner	inputs	comparisons	joins	output tuples	
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

Method	Number o	of tuples	t	lumber of	Total	Average	
used	. outer	inner	inputs	comparisons	joins	output tuples	
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

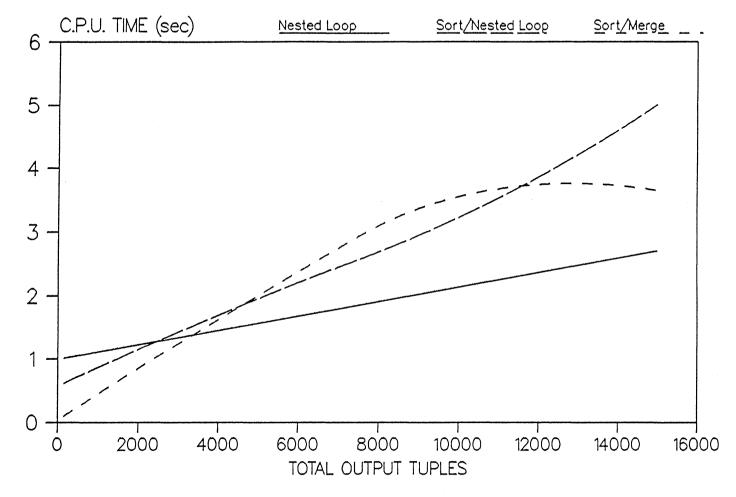
# OJOIN \* (TABLE1, TABLE2) WHERE A > U (BLOCK INNER RELATION, BLOCKSIZE = 50)

# TABLE XII

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

Method	Number o	of tuples	1	Number of	Total	Average	
used	Outer	inner	inputs	comparisons	joins	output tuples	
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)

ភ ភ

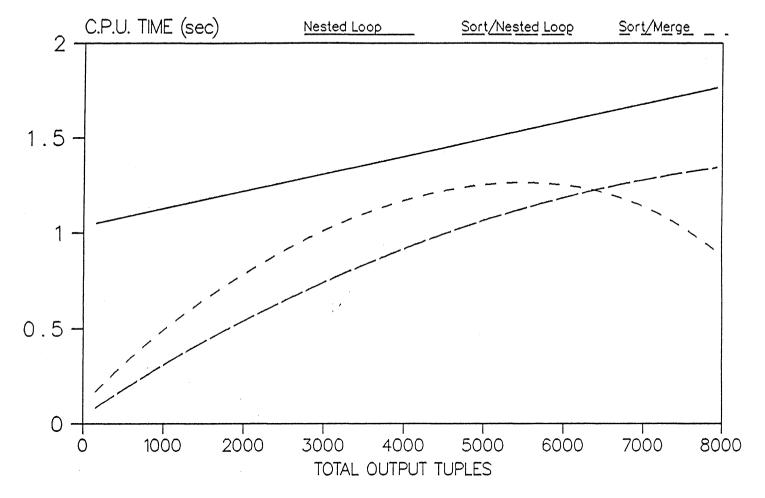


Figure 8. C.P.U Time per Tuple (Block Inner Relation)

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 pos-

sibility 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:

- 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:

- 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, <u>The Theory of Joins</u> 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., <u>An Introduction to Database Systems</u>, 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., <u>Query Pro-</u> cessing in Database Systems, Springer-Verlag, 1984.
- [18] Kim, W., D. Reiner, and D. Batory, eds., <u>Query Pro-</u> <u>cessing in Database Systems</u>, Springer-Verlag, 1985.
- [19] Knuth, D.E. <u>The Art of Computer Programming</u> vol. 3: <u>Sorting and Searching</u>. Addison-Wesley, Reading, Mass., 1973.
- [20] Levy, L.S. <u>Discrete Structures of Computer Science</u>, 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., <u>The Theory of Relational Databases</u>, 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. <u>Principles of Database Systems.</u> 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.

**.**. ,

4

# APPENDIXES

. .

.

# APPENDIX A

# TABLE1 AND TABLE2 RELATIONS

. .

.

.

# RELATION TABLE1

F	E	D	с	8	A
975	457	852	71	970	452
811	723	548	567	925	179
80	577	763	34	835	980
624	557	373	264	817	394
394	500	476	917	88	38
448	679	646	51	619	806
274	67	673	61	29	463
813	666	812	262	56	26
128	960	263	538	195	571
27	185	972	639	838	739
525	376	362	768	37	68
232	72	471	922	34	350
815	922	804	935	509	219
166	136	82	125	322	140
42	629	697	852	849	658
160	560	75	904	122	95
417	489	70	432	334	773
411	640	941	924	169	58
143	446	130	424	294	45
- 53	729	70	ô26	535	368
879	203	168	571	157	267
108	540	145	311	648	632
602	803	641	840	822	40
668	452	541	936	331	798
173	345	468	885	120	458
661	6	521	535	392	174
372	772	56	404	667	378
402	203	489	667	797	558
695	117	110	536	868	196
161	806	99	806	964	129
682	131	734	957	379	521
207	265	495	63	196	259
63	684	878	840	774	661
385	449	459	162	799	903
512	854	637	326	737	330
732	818	756	73	900	718
269	108	459	661	531	473 -
184	209	452	200	518	572
437	.477	770	665	34	620
900	887	491	537	892	603
215	143	68	678	131	683
228	11	303	697	623	466

	E	D	с	8	<b>A</b>
			 664	 .7	604
4	532	983	227	685	503
	969 892	665 109	66	827	897
-	428	- 677	692	650	256
3	854	674	737	635	. 101
2	143	0/4	366	235	291
7	454	229	563	652	842
1	133	462	822	603	64
9	725	850	164	281	505
9	271	162	36	851	526
. 8	803	590	426	182	178 344
5	86	516	318	379	552
1	469	552	306	593 740	596
4	768	234	640	695	706
6	531 536	606 39	285	461	612
4	955	285	459	284	163
9	355	371	202	896	104
2 2	156	891	480	561	139
2	749	899	991	631	248
	335	337	555	284	373
6	651	85	916	471	879
•	302	766	119	93	469
6	278	605	851	541	445
	. 655	468	748	278	130
2	863	904	572	536	679
7	748	134	185	765	207 51
6	101	652	293	650 617	968
6	343	977	868	- 207	953
4	384	408	859 137	814	489
3	353	246	497	543	268
1	650 578	145	757	604	565
5	493	642	537	149	491
3	858	793	936	208	73 <sup>-</sup>
2	697	400	6	623	240
7	522	51	472	150	161
5	838	895	493	455	940
7	563	973	194	0	196
7	129	410	215	175	505
2	435	761	368	814	862
2	143	865	660	. 900	201 317
6	874	813	189	45 . 808	331
9	590	698	737 58	211	819
5	664	895	58 824	\$ 569	899 .
1	789	194	761	100	725
7	733	447	636	272	920
2	1	573	241	795	272
5	507 274	374	297	865	522
6	2/4 915	743	84	296	33
8	915	152	545	652	494
8	957	726	411	23	481
1	542	39	466	. 880	449
2	207	804	305	987	. 536
9	991	685	- 654	929	246
8	281	787	441	890	525
8	• 192	617	224	158	836 .

•.

71

•

# **RELATION TABLE2**

.

.

.

604         7         664         913         512         413           603         627         226         663         969         77           256         650         627         664         913         77           256         650         627         677         464         160           231         233         366         0         454         160           64         603         622         453         725         980           505         281         164         850         725         980           505         281         164         850         603         815           344         379         318         516         666         524           536         740         640         234         768         460           76         691         44         606         531         682           104         296         232         235         933         168           103         286         631         601         891         156         201           373         284         555         337         733         128 <td< th=""><th>U</th><th>Ý</th><th>*</th><th>x</th><th>•</th><th>z</th></td<>	U	Ý	*	x	•	z
503         665         227         665         109         622         64           256         650         652         677         428         172           101         655         737         674         654         161           291         235         366         0         143         290           64         603         622         462         133         138           505         281         164         650         723         960           526         831         36         162         271         941           344         379         318         516         86         524           526         533         336         426         624         768           576         760         640         234         768         460           612         661         426         38         336         495           139         561         460         891         156         201           246         631         991         867         335         222           479         971         916         857         913         135 <t< td=""><td>604</td><td>7</td><td>664</td><td>983</td><td>537</td><td></td></t<>	604	7	664	983	537	
697         827         66         109         82         84           256         650         652         677         428         172           101         635         737         674         854         161           291         235         366         0         433         290           644         603         622         426         133         138           505         281         164         850         725         980           528         451         36         162         271         941           178         182         426         590         603         815           528         700         640         234         768         400           706         695         44         606         531         6492           163         284         459         285         513         632           163         284         459         285         513         632           163         284         459         285         513         632           163         284         451         651         633           164 <td< td=""><td></td><td>685</td><td></td><td></td><td></td><td></td></td<>		685				
236         630         692         677         428         172           101         635         737         674         654         163           291         233         366         0         143         290           65         603         622         462         133         138           505         211         164         850         723         980           505         211         164         850         723         980           505         211         164         850         723         980           505         211         164         850         723         980           515         66         531         66         524         665         665         460         234         469         167           526         706         605         248         636         331         682         460         167         788         400           104         896         702         201         236         936         493         201           373         284         555         537         937         315         222           469				109		
291225366 $10$ $14$ $16$ $16$ 64603822262133748646038232294621337485052811648527719805288513685260381434437931851686524552593306552460167556740640234768400566740640234768600706695446065316027066954460653160270765146093115620170874012237179331048962023717933124455337335122773284655651201273284555307335459931197665024455418516052787320776518446675714665316795320785940831413774035345854347714665360475714665465765965473208395833749314369834757 <t< td=""><td></td><td></td><td></td><td></td><td></td><td></td></t<>						
442         852         563         225         453         748           64         603         822         453         133         138           505         281         164         450         133         138           526         451         36         850         271         941           778         182         428         390         271         941           344         379         318         590         271         941           522         593         306         552         460         167           596         700         660         531         682         953         913           612         461         285         395         913         156         201           733         284         453         391         156         201         248         651         633           733         284         531         991         156         201         248         651         631         632           733         284         531         991         156         202         94         455         651         633         122						
64         603         822         465         133         138           505         281         154         850         723         980           178         182         438         390         803         813           344         379         308         516         86         524           596         740         640         234         768         400           706         695         44         606         531         682           516         284         459         285         913         682           163         284         459         285         913         682           164         896         202         371         7         7           139         561         480         891         156         265           139         561         480         891         156         201           373         284         555         337         335         22           469         93         119         766         302         94           469         93         119         766         302         94           469 </td <td></td> <td></td> <td></td> <td></td> <td></td> <td></td>						
505         281         164         905         722         940           778         182         428         390         271         940           778         182         428         390         201         941           344         379         318         590         201         941           552         593         306         552         466         167           706         695         44         606         531         682           706         695         44         606         531         682           104         896         202         371         7         231           103         551         460         891         156         201           248         551         991         899         749         12           373         284         455         337         335         222           499         93         119         766         302         94           453         541         855         134         748         655           459         931         196         656         302         94           4						
526         451         36         122         271         101           344         379         318         516         603         812           552         593         306         552         469         167           586         740         640         234         768         167           586         740         640         234         768         167           706         695         444         606         531         682           163         284         525         39         536         493           104         896         202         371         7         231           133         561         480         891         156         201           373         284         553         337         335         22           373         284         553         337         335         22           469         93         119         766         302         94           445         541         655         105         17         653         165           130         278         748         468         655         17         6						
$\begin{array}{c c c c c c c c c c c c c c c c c c c $		451				
344         379         318         516         66         524           552         593         306         552         469         167           566         740         640         234         768         400           706         695         44         606         531         682           612         461         285         39         536         495           163         284         459         285         913         104         896         202         371         7         231           139         561         480         691         155         201         237         233         221           373         284         551         897         739         212         237         335         222           373         284         551         651         631         211         246         651         631           469         93         119         766         302         94         445         541         851         632         167           207         765         185         134         740         737         565         605         177						
5967.06.02.27.06.06.0706695446065316821632844592859559131048962023717231139551480891156201373284555337335224699311976630294445541615134748665278469931197663029444554165518513474846993119766302944455416551347486652077651851347487482077651851347487379686176689773436059532078594083644282685434972466501945656047571455785277320893679385833774035331666530297732089367938583377320893679385833774035331666475714573208936733858337740353316664757145 <tr< td=""><td></td><td></td><td></td><td></td><td></td><td></td></tr<>						
706 $695$ $44$ $606$ $531$ $682$ $612$ $611$ $261$ $225$ $395$ $913$ $104$ $896$ $202$ $371$ $7$ $231$ $104$ $896$ $202$ $371$ $7$ $231$ $139$ $551$ $480$ $891$ $158$ $201$ $248$ $631$ $991$ $899$ $749$ $12$ $373$ $284$ $555$ $337$ $335$ $22$ $879$ $471$ $916$ $85$ $651$ $635$ $469$ $93$ $119$ $766$ $302$ $84$ $445$ $541$ $851$ $605$ $278$ $659$ $479$ $536$ $572$ $904$ $663$ $267$ $79$ $536$ $572$ $904$ $663$ $267$ $207$ $765$ $185$ $134$ $746$ $737$ $968$ $617$ $868$ $977$ $343$ $605$ $953$ $207$ $859$ $408$ $354$ $428$ $266$ $543$ $477$ $246$ $353$ $318$ $266$ $604$ $757$ $145$ $578$ $527$ $491$ $149$ $537$ $642$ $493$ $627$ $73$ $208$ $936$ $793$ $853$ $778$ $240$ $623$ $6$ $400$ $697$ $278$ $940$ $455$ $493$ $895$ $638$ $530$ $196$ $0$ $194$ $973$ $563$ $716$ <td< td=""><td></td><td></td><td></td><td></td><td></td><td></td></td<>						
612         425         33         536         495           163         284         459         285         955         913           104         896         202         371         7         231           139         551         480         891         156         201           373         284         555         337         335         22           879         471         916         85         651         633           469         93         119         766         302         94           445         541         851         605         278         659           130         278         744         466         653         267           707         755         185         134         746         767           51         650         293         652         101         647           953         207         859         408         364         428           469         814         137         740         353         318           565         604         757         145         578         527           73         20						
163         24         450         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         651         605         278         658           479         536         572         904         863         267           207         765         185         134         746         737           968         617         868         977         343         605           953         207         859         408         364         428           469         814         137         740         353         318           268         543         497         246         650         194           9						
104 $996$ $202$ $371$ $7$ $7231$ $139$ $561$ $480$ $891$ $156$ $201$ $373$ $284$ $555$ $337$ $335$ $22$ $379$ $471$ $916$ $855$ $651$ $638$ $469$ $93$ $119$ $766$ $302$ $94$ $445$ $541$ $851$ $605$ $278$ $659$ $479$ $278$ $748$ $468$ $655$ $17$ $879$ $536$ $572$ $904$ $863$ $267$ $707$ $765$ $185$ $134$ $746$ $737$ $51$ $650$ $293$ $652$ $101$ $647$ $968$ $617$ $868$ $977$ $343$ $665$ $953$ $207$ $859$ $408$ $364$ $428$ $486$ $543$ $497$ $246$ $650$ $194$ $565$ $604$ $757$ $145$ $578$ $527$ $73$ $208$ $936$ $793$ $853$ $710$ $505$ $175$ $215$ $410$ $129$ $765$ $940$ $455$ $493$ $895$ $838$ $530$ $940$ $455$ $493$ $895$ $143$ $270$ $311$ $806$ $737$ $698$ $590$ $915$ $899$ $569$ $824$ $194$ $729$ $139$ $73$ $208$ $936$ $793$ $853$ $710$ $505$ $175$ $215$ $410$ $129$ $765$ <td></td> <td></td> <td></td> <td></td> <td></td> <td></td>						
139       561       480       891       156       201         248       631       991       899       749       12         373       284       555       337       335       22         879       471       916       85       651       633         445       541       851       605       278       659         130       278       746       468       655       17         679       536       572       904       863       2657         207       765       185       134       746       737         968       617       868       977       343       605         953       207       859       408       364       428         469       814       137       740       353       318         268       543       497       246       650       194         555       604       757       145       578       527         73       208       936       793       858       337         240       623       6       400       657       278         73       208						
248       631       991       809       749       12         373       284       555       337       335       22         379       471       916       85       651       633         469       93       119       766       302       94         4435       541       851       605       278       653         130       278       748       468       655       17         207       765       185       134       748       767         207       765       185       134       748       767         968       617       868       977       343       605         953       207       859       408       354       428         459       814       137       740       353       318         268       543       497       246       650       194         565       604       757       145       578       527         73       208       936       793       858       337         491       149       537       642       493       627         733       208 <td></td> <td></td> <td></td> <td></td> <td></td> <td></td>						
373       284       555       337       335       22         879       471       916       85       651       633         469       93       119       766       302       94         445       541       851       605       278       659         130       278       748       466       655       17         879       536       572       904       863       267         207       765       185       134       748       746         968       617       868       977       343       605         953       207       8659       406       364       428         469       814       137       740       353       318         266       543       497       246       650       194         565       604       757       145       578       527         73       208       936       793       853       377         240       623       6       400       697       278         161       150       472       51       522       784         940       455			991			
#79       471       916       #5       651       635         469       93       119       766       302       94         130       278       748       468       655       17         879       536       572       904       863       267         207       765       185       134       748       748         968       617       868       977       343       605         953       207       859       408       364       428         268       543       497       246       650       194         565       604       757       145       578       522       78         73       208       936       793       858       377       246       650       194         565       604       757       145       578       522       784         940       455       493       895       838       530       106         161       150       472       51       522       784         940       455       493       895       638       530         161       150       472				337	335	
445 $541$ $851$ $605$ $322$ $942$ $130$ $278$ $748$ $468$ $655$ $17$ $879$ $536$ $572$ $904$ $863$ $267$ $207$ $765$ $185$ $134$ $746$ $737$ $968$ $617$ $868$ $977$ $343$ $605$ $953$ $207$ $859$ $408$ $364$ $428$ $268$ $543$ $497$ $246$ $650$ $194$ $565$ $604$ $757$ $145$ $578$ $527$ $73$ $208$ $936$ $793$ $858$ $377$ $240$ $6223$ $6$ $400$ $637$ $278$ $73$ $208$ $936$ $793$ $858$ $377$ $240$ $623$ $6$ $400$ $637$ $278$ $940$ $455$ $493$ $895$ $838$ $530$ $161$ $150$ $472$ $51$ $522$ $784$ $940$ $455$ $493$ $895$ $838$ $530$ $196$ $0$ $194$ $973$ $563$ $710$ $505$ $175$ $215$ $410$ $129$ $765$ $862$ $814$ $368$ $761$ $435$ $230$ $311$ $806$ $737$ $698$ $590$ $915$ $819$ $211$ $58$ $895$ $664$ $545$ $899$ $569$ $824$ $194$ $789$ $139$ $725$ $100$ $761$ $447$ $733$ $772$ <						
130         278         748         468         655         17           879         536         572         904         863         267           207         765         185         134         748         748         748           968         617         868         977         343         605           953         207         859         408         364         428           469         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						
879 $536$ $572$ $904$ $863$ $267$ $207$ $765$ $185$ $134$ $748$ $737$ $968$ $617$ $868$ $977$ $343$ $605$ $953$ $207$ $859$ $408$ $364$ $428$ $469$ $814$ $137$ $740$ $353$ $318$ $268$ $543$ $497$ $246$ $650$ $194$ $565$ $604$ $757$ $145$ $578$ $527$ $491$ $149$ $537$ $642$ $493$ $627$ $73$ $208$ $936$ $793$ $858$ $377$ $240$ $623$ $6$ $400$ $697$ $278$ $161$ $150$ $472$ $51$ $522$ $784$ $940$ $455$ $493$ $895$ $838$ $530$ $196$ $0$ $194$ $973$ $563$ $710$ $505$ $175$ $215$ $410$ $129$ $765$ $862$ $814$ $368$ $761$ $435$ $230$ $201$ $900$ $660$ $865$ $143$ $270$ $311$ $808$ $737$ $698$ $590$ $915$ $819$ $211$ $58$ $895$ $664$ $545$ $899$ $596$ $824$ $194$ $789$ $139$ $725$ $100$ $761$ $447$ $723$ $772$ $920$ $272$ $636$ $33$ $1$ $260$ $772$ $795$ $241$ $573$ $507$ $567$ <						
$\begin{array}{cccccccccccccccccccccccccccccccccccc$						
51 $650$ $293$ $652$ $101$ $647$ $968$ $617$ $868$ $977$ $343$ $605$ $953$ $207$ $859$ $408$ $364$ $428$ $469$ $814$ $137$ $740$ $353$ $318$ $268$ $543$ $497$ $246$ $650$ $194$ $565$ $604$ $757$ $145$ $578$ $527$ $491$ $149$ $537$ $642$ $493$ $627$ $73$ $208$ $936$ $793$ $858$ $377$ $240$ $623$ $6$ $400$ $697$ $278$ $161$ $150$ $472$ $51$ $522$ $784$ $940$ $455$ $493$ $895$ $838$ $530$ $196$ $0$ $194$ $973$ $563$ $710$ $505$ $175$ $215$ $410$ $129$ $765$ $862$ $614$ $368$ $761$ $435$ $230$ $201$ $900$ $660$ $865$ $143$ $270$ $311$ $808$ $737$ $698$ $590$ $915$ $819$ $211$ $58$ $895$ $664$ $545$ $899$ $569$ $824$ $194$ $789$ $139$ $725$ $100$ $761$ $447$ $733$ $772$ $520$ $2772$ $636$ $38$ $1$ $260$ $272$ $865$ $297$ $374$ $274$ $659$ $33$ $295$ $84$ $743$ $915$ $652$ <tr< td=""><td></td><td></td><td></td><td></td><td></td><td></td></tr<>						
968         617         868         977         343         603           953         207         859         408         364         428           469         814         137         740         353         318           268         543         497         246         650         194           565         604         757         145         578         527           491         149         537         642         493         627           73         206         936         793         853         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 <td< td=""><td>51</td><td></td><td></td><td></td><td></td><td></td></td<>	51					
953       207       859       408       364       428         469       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       657       278         161       150       472       51       522       784         940       455       493       895       838       530         196       0       194       973       553       710         505       175       215       410       129       765         862       614       368       761       435       230         201       900       660       865       143       270         317       45       189       813       874       660         331       808       737       698       590       915         899       569						
469         814         137         740         353         318           268         543         497         246         650         194           565         604         757         145         578         527           73         208         936         793         853         377           240         623         6         400         697         278           940         455         493         895         838         530           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			859			
365 $604$ $757$ $145$ $578$ $527$ $491$ $149$ $537$ $642$ $493$ $627$ $73$ $208$ $936$ $793$ $858$ $377$ $240$ $623$ $6$ $400$ $697$ $278$ $161$ $150$ $472$ $51$ $522$ $784$ $940$ $455$ $493$ $895$ $838$ $530$ $196$ $0$ $194$ $973$ $563$ $710$ $505$ $175$ $215$ $410$ $129$ $765$ $862$ $814$ $368$ $761$ $435$ $230$ $201$ $900$ $660$ $865$ $143$ $270$ $317$ $45$ $189$ $813$ $874$ $660$ $331$ $808$ $737$ $698$ $590$ $915$ $819$ $211$ $58$ $895$ $664$ $545$ $899$ $569$ $824$ $194$ $789$ $139$ $725$ $100$ $761$ $447$ $723$ $772$ $920$ $272$ $636$ $33$ $1$ $260$ $272$ $865$ $297$ $374$ $274$ $659$ $33$ $295$ $545$ $152$ $957$ $857$ $481$ $23$ $411$ $726$ $950$ $114$ $449$ $880$ $466$ $39$ $542$ $184$ $536$ $987$ $305$ $804$ $207$ $276$ $494$ $652$ $545$ $152$ $950$ $114$ <						
$\begin{array}{c ccccccccccccccccccccccccccccccccccc$						
$\begin{array}{c ccccccccccccccccccccccccccccccccccc$						
$\begin{array}{cccccccccccccccccccccccccccccccccccc$						
161150 $472$ 51522784940455493895838530196019497356371050517521541012976586281436876143523020190066086514327031745189813874660331808737698590915819211538956645458995698241947891397251007614477337729202726363812602727952415735075893329584743915632494652545152957857481234117269501144598804663954218536987305804207276246929654685991917						
9404554938956385301960194973563710505175215410129765862614368761435230201900660865143270317451898138746603318087376985909158192115889566454589956982419478913972510076144772377292027279524157350758933295 $297$ 374274659494652545152957857481234117269501144398804663954218536987305804207276246929654685991917						
1960 $194$ $973$ $563$ $710$ $505$ $175$ $215$ $410$ $129$ $765$ $862$ $814$ $368$ $761$ $435$ $230$ $201$ $900$ $660$ $865$ $143$ $270$ $317$ $45$ $189$ $813$ $874$ $660$ $331$ $808$ $737$ $698$ $590$ $915$ $819$ $211$ $58$ $895$ $664$ $545$ $899$ $569$ $824$ $194$ $789$ $139$ $725$ $100$ $761$ $447$ $733$ $772$ $520$ $272$ $636$ $38$ 1 $260$ $272$ $795$ $241$ $573$ $507$ $589$ $522$ $865$ $297$ $374$ $274$ $659$ $33$ $295$ $64$ $743$ $915$ $632$ $494$ $652$ $545$ $152$ $957$ $857$ $481$ $23$ $411$ $726$ $950$ $114$ $536$ $987$ $305$ $804$ $207$ $276$ $246$ $929$ $654$ $685$ $991$ $917$	940					
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         632           494         652         545         152         957         857           481         23         411         726         950         114           4		0				
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       64       743       915       632         494       652       545       152       957       857         481       23       411       726       950       114         439       880       466       39       542       18         536       987       305       804       207       276         449       880				410		
$\begin{array}{cccccccccccccccccccccccccccccccccccc$				761		
$\begin{array}{cccccccccccccccccccccccccccccccccccc$						270
819       211       58       635       564       545         899       569       824       194       729       139         725       100       761       447       733       772         920       272       636       38       1       260         277       795       241       573       507       589         522       865       297       374       274       659         33       295       84       743       915       632         494       652       545       152       957       857         481       23       411       726       950       114         536       987       305       804       207       276         654       685       991       917       917       917						
899         569         824         194         789         139           725         100         761         447         723         772           920         272         636         38         1         260           272         795         241         573         507         589           33         295         24         743         915         632           494         652         545         152         957         857           481         23         411         726         950         114           536         987         305         804         207         276           266         929         654         685         991         917						
$\begin{array}{cccccccccccccccccccccccccccccccccccc$						
\$20         272         636         38         1         260           272         795         241         573         507         589           522         865         297         374         274         659           33         295         84         743         915         632           494         652         545         152         957         857           481         23         411         726         950         114           449'         880         466         39         542         18           536         987         305         804         207         276           245         929         654         685         991         917						
272       795       241       573       507       589         522       865       297       374       274       659         33       295       84       743       915       632         494       652       545       152       957       857         481       23       411       726       950       114         449'       880       466       39       542       18         536       987       305       804       207       276         245       929       654       685       991       917						
522         865         297         374         274         659           33         295         84         743         915         632           494         652         545         152         957         857           481         23         411         726         950         114           449         880         466         39         542         18           536         987         305         804         207         276           245         929         654         685         991         917						
33         295         ê4         743         915         ê32           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						
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						
449         880         466         39         542         18           536         987         305         804         207         276           246         929         654         685         991         917						
536         987         305         804         207         276           246         929         654         685         991         917					950	
<b>246</b> 929 654 685 991 917						
	525	890	054 441	685 787		
876 168 004 201 072						
630 138 224 617 192 890				017	192	620

.

30 $80$ $917$ $476$ $500$ $300$ $463$ $29$ $61$ $673$ $677$ $26$ $26$ $56$ $222$ $812$ $666$ $673$ $677$ $26$ $739$ $838$ $639$ $9772$ $185$ $868$ $377$ $768$ $3622$ $376$ $9272$ $185$ $68$ $37$ $768$ $362$ $925$ $922$ $471$ $722$ $270$ $489$ $42$ $922$ $904$ $75$ $560$ $173$ $334$ $422$ $904$ $75$ $560$ $1773$ $334$ $422$ $904$ $75$ $560$ $1773$ $334$ $422$ $904$ $75$ $560$ $1772$ $267$ $517$ $571$ $1668$ $203$ $626$ $770$ $729$ $267$ $517$ $571$ $166$ $203$ $632$ $648$ $311$ $1455$ $203$ $677$ <t< th=""><th></th><th><b>.</b> . <b>.</b></th><th>×</th><th>W</th><th>V</th><th>. U</th></t<>		<b>.</b> . <b>.</b>	×	W	V	. U
179       925 $567$ $568$ $777$ $777$ $394$ $817$ $264$ $373$ $557$ $68$ $38$ $88$ $917$ $478$ $500$ $257$ $806$ $619$ $51$ $646$ $879$ $257$ $463$ $225$ $616$ $673$ $677$ $677$ $276$ $556$ $262$ $812$ $666$ $673$ $739$ $838$ $639$ $9772$ $185$ $566$ $376$ $362$ $376$ $52$ $66$ $37$ $768$ $362$ $376$ $52$ $376$ $56$ $560$ $772$ $378$ $639$ $972$ $185$ $560$ $773$ $376$ $576$ $560$ $772$ $777$ $575$ $560$ $773$ $3734$ $422$ $970$ $489$ $424$ $130$ $446$ $424$ $130$ $446$ $366$ $525$ $626$ $770$ $729$ $267$ $577$ $571$ $168$ $567$					970	452
980         835         34         763         977           39         817         264         373         557           38         88         917         476         500           463         29         61         673         67           26         56         262         812         666           571         195         538         263         960           739         838         639         972         185           68         37         758         362         376         53           350         34         922         417         72         5           350         34         922         417         72         5           658         849         852         697         529         5           95         122         904         75         560         7           364         55         626         70         729         7           364         55         626         70         729         7           363         313         936         641         603         6           667         404						179
33         8         817         264         373         557         6           806         619         51         646         879         2           463         29         61         673         67         2           26         56         262         812         666         6           739         838         639         972         185         6           350         34         922         471         72         5           219         509         935         804         922         6           68         37         768         82         697         629           140         322         125         82         136         629           95         122         904         75         560         629         629           773         334         432         70         489         45         294         424         941         646         45           368         535         636         70         723         632         648         111         145         540         46           40         822         840         641 </td <td></td> <td></td> <td></td> <td>34</td> <td></td> <td></td>				34		
32 $86$ $917$ $476$ $500$ $32$ $463$ $29$ $61$ $673$ $677$ $26$ $26$ $56$ $222$ $812$ $666$ $666$ $677$ $185$ $68$ $37$ $768$ $362$ $376$ $960$ $739$ $350$ $34$ $922$ $471$ $72$ $219$ $509$ $935$ $804$ $922$ $216$ $5136$ $249$ $852$ $637$ $629$ $95$ $122$ $904$ $75$ $560$ $1773$ $334$ $432$ $70$ $489$ $46$ $366$ $535$ $626$ $70$ $729$ $667$ $420$ $267$ $157$ $5711$ $166$ $203$ $667$ $420$ $822$ $840$ $641$ $803$ $667$ $40$ $822$ $840$ $641$ $803$ $667$ $420$ $826$ $772$ $558$ $7777$	6		373.			
463       29       61       673       673       66         26       56       262       812       666       66         571       195       538       263       960       1         66       37       768       362       376       5         350       34       922       471       72       2         140       322       125       82       136       1         658       849       852       697       629       9         95       122       904       75       560       1         773       334       432       70       489       4         45       294       424       130       446       2         368       535       626       70       729       2         632       648       311       145       540       4         40       822       840       641       803       6         778       331       936       541       452       6         174       392       535       521       6       6         578       797       647       499 <td>3</td> <td></td> <td></td> <td></td> <td></td> <td></td>	3					
26       56       252       815       666       667         571       195       538       263       960       1         66       37       768       362       376       5         350       34       922       471       72       5         219       509       935       804       922       6         400       322       125       82       136       1         658       849       852       697       639       9         95       122       904       75       560       1         773       334       432       70       469       4         45       294       424       130       446       4         368       166       535       626       70       739       5         267       157       571       168       168       203       6         798       331       936       541       452       6       6         714       392       535       521       6       6       772       2         58       797       667       489       203       6	4					
5711       195 $538$ $263$ $960$ $1739$ $68$ $37$ $7788$ $362$ $376$ $5736$ $972$ $185$ $350$ $34$ $922$ $471$ $72$ $272$ $219$ $219$ $509$ $935$ $804$ $922$ $471$ $72$ $219$ $440$ $3222$ $122$ $904$ $75$ $560$ $773$ $334$ $4322$ $70$ $499$ $45$ $95$ $122$ $904$ $75$ $560$ $773$ $334$ $4322$ $70$ $499$ $46$ $279$ $677$ $70$ $493$ $45$ $294$ $424$ $941$ $640$ $279$ $67$ $70$ $493$ $460$ $279$ $67$ $70$ $493$ $466$ $772$ $79$ $797$ $636$ $711$ $166$ $67$ $798$ $331$ $996$ $541$ $452$ $66$ $772$ $558$ $777$ $667$ $736$ $734$ $1311$ $675$ $6$	2					
7398386339721856837768362376835034922471727221950993580492281403221258821361658849652697629995122904755601773334432704994529442413044636853562670729632648311145540408228406418037983319365414526481208654683457743925355216786674045677255879766748920312996480653611017439253552165213799577341316337373266374893307371266378685725162004522096331316674918676431316786811437490073737685475516200452209661774866733181673531667303<	8					
68       37       768       362       376         219       509       935       804       922       2         140       322       135       804       922       2         658       649       852       697       629       136         95       122       904       75       560       1         773       334       432       70       489       4         368       535       626       70       729       2         632       648       311       145       540       4         40       622       648       311       145       540       4         40       622       648       311       145       540       4         40       622       648       311       145       540       4         174       392       535       521       6       6       6         378       667       404       56       772       5       5       5       5         174       392       535       521       6       6       6       6       6       6       6       6       6	1					
350 $34$ $922$ $471$ $72$ $140$ $322$ $125$ $82$ $936$ $140$ $322$ $125$ $82$ $936$ $95$ $122$ $904$ $75$ $560$ $773$ $334$ $432$ $70$ $489$ $45$ $294$ $424$ $130$ $446$ $45$ $294$ $424$ $130$ $446$ $45$ $294$ $424$ $130$ $446$ $368$ $535$ $626$ $70$ $729$ $267$ $157$ $571$ $168$ $203$ $40$ $822$ $840$ $641$ $803$ $40$ $822$ $840$ $641$ $803$ $40$ $822$ $840$ $641$ $803$ $40$ $822$ $840$ $641$ $803$ $40$ $822$ $840$ $641$ $803$ $40$ $822$ $840$ $641$ $803$ $78$ $667$ $404$ $56$ $772$ $558$ $797$ $667$ $489$ $203$ $714$ $392$ $535$ $521$ $6$ $513$ $495$ $2655$ $225$ $558$ $797$ $563$ $495$ $259$ $964$ $806$ $998$ $806$ $521$ $379$ $957$ $734$ $131$ $259$ $964$ $806$ $995$ $203$ $310$ $737$ $226$ $637$ $654$ $903$ $799$ $162$ $459$ $449$ $313$ $657$ <td>5</td> <td></td> <td></td> <td>768</td> <td></td> <td></td>	5			768		
140       322       125       82       136       1         658       699       852       697       629       1         95       122       904       75       560       1         773       334       432       70       489       4         45       294       424       130       446       46         368       535       626       70       729       267       157       571       168       203       6         632       648       311       145       540       6       6       6       6         798       331       936       541       452       6       6       6       6         774       667       404       56       772       5       6       772       5       6       7       6 <t< td=""><td>2</td><td></td><td></td><td></td><td></td><td></td></t<>	2					
658 $0.000$ $0.52$ $0.57$ $62.9$ 95       122       904       75       560       1         53       169       924       941       640       4         366       535       626       70       729       729         267       157       571       168       203       6         40       822       840       641       803       6         40       822       840       641       803       6         788       331       936       541       452       6         478       120       885       468       345       1         174       392       535       521       6       6         378       667       404       56       772       2         588       797       667       499       203       2         196       868       536       110       117       6         259       196       633       495       265       2         196       868       536       100       117       6         251       379       957       734	8					
95122904755607733344327048945294424130446462944241304463685356267072926715757116820363264831114554040822840641803788331936541452408225355216778331936541452408654683451743925355216378667404567725587976674992031868685361101175299648069980652137995773413125919663765426617748408786843007373266378544735316614591085725182004522096038925374918876331316786814363373465199833163373465199833163365730622024263365730622025576379778461797 <td>1</td> <td></td> <td></td> <td></td> <td></td> <td></td>	1					
773 $334$ $432$ $70$ $489$ $58$ $169$ $924$ $941$ $640$ $446$ $366$ $535$ $626$ $70$ $729$ $267$ $157$ $571$ $168$ $203$ $66$ $632$ $648$ $311$ $145$ $540$ $788$ $40$ $822$ $840$ $641$ $803$ $66$ $798$ $331$ $936$ $541$ $452$ $66$ $778$ $331$ $936$ $541$ $452$ $66$ $778$ $667$ $404$ $56$ $772$ $277$ $558$ $797$ $667$ $499$ $203$ $265$ $378$ $667$ $404$ $56$ $772$ $277$ $558$ $797$ $637$ $499$ $203$ $265$ $521$ $379$ $957$ $734$ $131$ $661$ $523$ $196$ $63$ $495$ $2665$ $272$ $330$ $737$ $326$ $637$ $854$ $331$ $572$ $518$ $2200$ $452$ $209$ $11$ $572$ $518$ $2200$ $452$ $209$ $11$ $563$ $734$ $651$ $998$ $331$ $265$ $677$ $216$ $733$ $181$ $276$ $633$ $892$ $537$ $491$ $887$ $265$ $770$ $777$ $226$ $770$ $477$ $286$ $661$ $774$ $633$ $892$ $537$ $854$ $143$ $265$ $572$ </td <td></td> <td></td> <td></td> <td></td> <td></td> <td></td>						
58       169       924       941       640       446 $45$ 294       424       130       446       446 $368$ 535       626       70       729       729 $267$ 157       571       168       203       8 $40$ $822$ $840$ 641       803       6 $40$ $822$ $840$ 641       803       6 $478$ 331       936       541       452       6 $478$ 320       535       521       6       6 $378$ 667       404       56       772       5 $558$ 797       667       489       203       2 $129$ 964       806       10       117       6 $521$ 379       957       734       131       6 $521$ 379       957       734       449       3 $521$ 379       126       637       854       3 $533$ 737       326       637       854       3 $572$	1					
45 $294$ $424$ $130$ $446$ $130$ $368$ $535$ $626$ $70$ $729$ $267$ $157$ $571$ $168$ $203$ $646$ $632$ $648$ $311$ $1455$ $540$ $641$ $40$ $822$ $840$ $641$ $803$ $661$ $798$ $331$ $936$ $541$ $452$ $66$ $456$ $120$ $885$ $468$ $345$ $677$ $174$ $392$ $535$ $521$ $6$ $67$ $378$ $667$ $404$ $56$ $772$ $57$ $558$ $797$ $667$ $489$ $203$ $203$ $129$ $964$ $806$ $99$ $8066$ $774$ $131$ $661$ $253$ $196$ $633$ $495$ $265$ $275$ $265$ $275$ $265$ $275$ $265$ $275$ $3776$ $684$ $330$ $737$ $3266$ $637$ $654$ $449$						58
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	-					
632 $648$ $311$ $145$ $540$ $40$ $822$ $640$ $641$ $803$ $663$ $798$ $331$ $936$ $541$ $452$ $663$ $458$ $120$ $885$ $468$ $345$ $663$ $174$ $392$ $535$ $521$ $6$ $378$ $667$ $404$ $56$ $772$ $772$ $558$ $797$ $667$ $489$ $203$ $667$ $196$ $868$ $536$ $110$ $117$ $667$ $129$ $964$ $806$ $99$ $806$ $110$ $521$ $379$ $957$ $734$ $131$ $678$ $259$ $196$ $63$ $495$ $265$ $265$ $661$ $774$ $840$ $878$ $684$ $718$ $903$ $799$ $162$ $459$ $449$ $333$ $310$ $737$ $326$ $637$ $854$ $718$ $900$ $73$ $756$ $818$ $772$ $633$ $892$ $537$ $491$ $887$ $933$ $666$ $623$ $697$ $303$ $11$ $777$ $733$ $663$ $131$ $678$ $681$ $1433$ $797$ $633$ $892$ $537$ $933$ $311$ $676$ $677$ $200$ $511$ $579$ $238$ $659$ $576$ $713$ $769$ $836$ $283$ $471$ $336$ $933$ $275$ $93$ $979$ $200$ $736$ $933$				626		
40 $822$ $840$ $641$ $803$ $647$ $798$ $331$ $936$ $541$ $452$ $6456$ $456$ $120$ $885$ $468$ $345$ $174$ $392$ $535$ $521$ $6$ $378$ $667$ $404$ $56$ $772$ $196$ $868$ $536$ $110$ $117$ $129$ $964$ $806$ $99$ $806$ $129$ $964$ $806$ $99$ $806$ $129$ $964$ $806$ $99$ $806$ $129$ $964$ $806$ $99$ $806$ $129$ $964$ $806$ $99$ $806$ $129$ $964$ $806$ $99$ $806$ $129$ $964$ $806$ $99$ $806$ $13379$ $957$ $734$ $131$ $663$ $495$ $265$ $265$ $265$ $661$ $774$ $840$ $878$ $684$ $903$ $799$ $162$ $459$ $449$ $778$ $330$ $737$ $326$ $637$ $854$ $776$ $572$ $516$ $200$ $452$ $209$ $777$ $603$ $892$ $537$ $491$ $887$ $96$ $633$ $131$ $678$ $68$ $143$ $774$ $663$ $651$ $996$ $331$ $784$ $651$ $996$ $511$ $579$ $738$ $461$ $797$ $777$ $713$ $769$ $836$ $283$ $471$ $777$ $7777$ <	8	203				
798 $331$ $936$ $541$ $452$ $633$ $458$ $120$ $885$ $468$ $345$ $1345$ $174$ $392$ $535$ $521$ $6$ $378$ $667$ $404$ $56$ $772$ $235$ $558$ $797$ $667$ $469$ $203$ $449$ $129$ $964$ $806$ $99$ $806$ $667$ $521$ $379$ $957$ $734$ $1311$ $6661$ $521$ $379$ $957$ $734$ $1311$ $6661$ $529$ $196$ $63$ $495$ $2655$ $265$ $259$ $196$ $63$ $495$ $2655$ $265$ $330$ $737$ $326$ $637$ $854$ $684$ $903$ $799$ $162$ $459$ $108$ $273$ $718$ $900$ $73$ $756$ $818$ $772$ $633$ $892$ $537$ $491$ $887$ $5663$ $683$ $131$ $678$ $68$ $143$ $27566637$ $663$ $623$ $697$ $303$ $111$ $774$ $6666$ $623$ $697$ $303$ $111$ $77466377$ $27662996697667676676766777900511579238659977777777777777777777777777777777777$	· · · · 1					
458       120 $885$ $468$ $345$ 1 $174$ $392$ $535$ $521$ 6       6 $378$ $667$ $404$ $56$ $772$ 5 $558$ $797$ $667$ $489$ $203$ 4 $196$ $868$ $536$ $110$ $117$ 6 $521$ $379$ $957$ $734$ $131$ 6 $521$ $379$ $957$ $734$ $131$ 6 $521$ $379$ $957$ $734$ $131$ 6 $651$ $774$ $840$ $878$ $684$ 9 $903$ $799$ $162$ $459$ $4499$ 3 $330$ $737$ $326$ $637$ $854$ 9 $473$ $531$ $661$ $459$ $108$ 3 $572$ $516$ $200$ $452$ $209$ 1 $633$ $131$ $678$ $68$ $143$ 2 $633$ $734$ $651$	6					
174 $392$ $535$ $521$ $6$ $378$ $667$ $404$ $56$ $772$ $772$ $558$ $797$ $667$ $409$ $203$ $469$ $196$ $868$ $536$ $110$ $117$ $667$ $129$ $964$ $806$ $99$ $806$ $110$ $259$ $195$ $633$ $4955$ $2655$ $2665$ $259$ $195$ $633$ $4955$ $2655$ $2661$ $903$ $799$ $162$ $459$ $4499$ $330$ $737$ $3266$ $6377$ $854$ $854$ $903$ $799$ $162$ $459$ $108$ $772$ $516$ $2200$ $452$ $209$ $108$ $772$ $516$ $2200$ $452$ $209$ $108$ $772$ $603$ $892$ $537$ $491$ $887$ $966$ $633$ $131$ $678$ $68$ $143$ $276$ $666$ $623$ $697$ $303$ $11$ $276$ $633$ $734$ $651$ $998$ $331$ $876$ $954$ $645$ $226$ $733$ $181$ $876$ $620$ $562$ $93$ $76$ $210$ $976$ $90$ $511$ $579$ $238$ $659$ $576$ $379$ $778$ $481$ $797$ $276$ $713$ $769$ $836$ $283$ $471$ $777$ $275$ $93$ $279$ $200$ $976$ $933$ $275$ $93$ <	- 6					
378 $667$ $404$ $56$ $772$ $378$ $558$ $797$ $667$ $489$ $203$ $489$ $196$ $868$ $536$ $110$ $117$ $489$ $129$ $964$ $806$ $99$ $806$ $521$ $379$ $957$ $734$ $131$ $684$ $259$ $196$ $63$ $495$ $265$ $265$ $661$ $774$ $840$ $878$ $684$ $903$ $903$ $799$ $162$ $459$ $449$ $330$ $330$ $737$ $326$ $637$ $854$ $933$ $473$ $531$ $661$ $459$ $108$ $743$ $572$ $516$ $200$ $452$ $209$ $162$ $620$ $34$ $665$ $770$ $477$ $463$ $633$ $131$ $678$ $68$ $143$ $2666$ $633$ $131$ $678$ $68$ $143$ $2766$ $666$ $623$ $697$ $303$ $11$ $2766$ $576$ $379$ $778$ $486$ $797$ $2766$ $576$ $379$ $778$ $481$ $797$ $2766$ $713$ $769$ $636$ $283$ $471$ $5766$ $310$ $685$ $492$ $242$ $302$ $5766$ $310$ $685$ $492$ $242$ $302$ $5766$ $310$ $685$ $492$ $242$ $302$ $5766$ $310$ $685$ $492$ $242$ $302$ $5766$ <t< td=""><td></td><td></td><td></td><td></td><td></td><td></td></t<>						
558 $797$ $667$ $489$ $203$ $203$ $196$ $868$ $536$ $110$ $117$ $661$ $129$ $964$ $806$ $99$ $806$ $521$ $379$ $957$ $734$ $131$ $661$ $259$ $196$ $63$ $495$ $265$ $661$ $774$ $840$ $878$ $684$ $903$ $799$ $162$ $459$ $449$ $330$ $330$ $737$ $326$ $637$ $854$ $718$ $900$ $73$ $756$ $818$ $76$ $473$ $531$ $661$ $459$ $108$ $3572$ $516$ $200$ $452$ $209$ $3666$ $633$ $572$ $516$ $200$ $452$ $209$ $3663$ $572$ $516$ $200$ $452$ $209$ $3766$ $633$ $892$ $537$ $491$ $887$ $9668$ $633$ $131$ $678$ $68$ $1433$ $2766$ $633$ $734$ $651$ $996$ $331$ $3766$ $666$ $623$ $697$ $303$ $111$ $2766$ $583$ $734$ $651$ $996$ $331$ $676$ $620$ $562$ $93$ $766$ $210$ $176$ $90$ $511$ $579$ $238$ $659$ $576$ $379$ $778$ $481$ $797$ $238$ $666$ $657$ $306$ $22$ $602$ $2933$ $933$ $275$ $93$ $679$						378
196       868       536       110       117       6         129       964       806       99       806       1         259       196       63       495       265       2         661       774       840       878       684         903       799       162       459       449       2         330       737       326       637       854       8         718       900       73       756       818       7         473       531       661       459       108       2         572       516       200       452       209       1         620       34       665       770       477       4         683       131       678       68       143       2         683       131       678       68       143       2         683       131       678       68       143       2         583       734       651       998       331       6         637       210       529       456       916       6         620       562       93       76	4					
521 $379$ $957$ $734$ $131$ $600$ $259$ $196$ $63$ $495$ $265$ $265$ $661$ $774$ $840$ $878$ $684$ $903$ $799$ $162$ $459$ $449$ $330$ $737$ $326$ $637$ $854$ $718$ $900$ $73$ $756$ $818$ $473$ $531$ $661$ $459$ $108$ $572$ $516$ $2200$ $452$ $209$ $620$ $34$ $665$ $770$ $477$ $603$ $892$ $537$ $491$ $887$ $683$ $131$ $678$ $68$ $143$ $666$ $623$ $697$ $303$ $11$ $583$ $734$ $651$ $998$ $331$ $954$ $645$ $226$ $733$ $181$ $637$ $210$ $529$ $456$ $916$ $620$ $562$ $93$ $76$ $210$ $90$ $511$ $579$ $238$ $659$ $576$ $379$ $778$ $481$ $90$ $511$ $579$ $238$ $659$ $576$ $379$ $778$ $481$ $713$ $769$ $836$ $222$ $604$ $310$ $685$ $492$ $242$ $302$ $933$ $275$ $93$ $979$ $200$ $933$ $275$ $93$ $979$ $200$ $933$ $275$ $93$ $979$ $200$ $933$ $275$ $933$ $979$ $200$ <td>6</td> <td></td> <td></td> <td></td> <td></td> <td></td>	6					
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	1					
661 $774$ $840$ $878$ $684$ $903$ $799$ $162$ $459$ $449$ $330$ $330$ $737$ $326$ $637$ $854$ $718$ $900$ $73$ $756$ $818$ $7$ $473$ $531$ $661$ $459$ $108$ $572$ $516$ $200$ $452$ $209$ $108$ $572$ $516$ $200$ $452$ $209$ $108$ $633$ $892$ $537$ $491$ $887$ $9887$ $683$ $131$ $678$ $68$ $143$ $2383$ $466$ $623$ $697$ $303$ $11$ $23883$ $466$ $623$ $697$ $303$ $11$ $23883$ $954$ $645$ $226$ $733$ $181$ $666$ $620$ $562$ $93$ $76$ $210$ $11$ $90$ $511$ $579$ $238$ $659$ $576$ $976$ $379$ $778$ $481$ $797$ $238$ $576$ $379$ $778$ $481$ $797$ $238$ $866$ $657$ $306$ $22$ $604$ $887$ $310$ $685$ $492$ $242$ $602$ $933$ $3275$ $93$ $579$ $200$ $92$ $933$ $275$ $331$ $82$ $2541$ $666$ $576$ $725$ $331$ $82$ $2541$ $576$ $576$ $94$ $4444$ $71$ $667$	. 6					
903 $799$ $162$ $459$ $449$ $531$ $330$ $737$ $326$ $637$ $654$ $718$ $900$ $73$ $756$ $818$ $473$ $531$ $661$ $459$ $108$ $572$ $516$ $200$ $452$ $209$ $620$ $34$ $665$ $770$ $477$ $603$ $892$ $537$ $491$ $887$ $683$ $131$ $678$ $68$ $143$ $466$ $623$ $697$ $303$ $11$ $583$ $734$ $651$ $996$ $331$ $954$ $645$ $226$ $733$ $181$ $637$ $210$ $529$ $456$ $916$ $620$ $562$ $93$ $76$ $210$ $90$ $511$ $579$ $238$ $659$ $576$ $379$ $778$ $481$ $797$ $713$ $769$ $836$ $283$ $471$ $114$ $168$ $858$ $578$ $229$ $866$ $657$ $306$ $22$ $604$ $310$ $685$ $492$ $242$ $802$ $933$ $275$ $93$ $879$ $200$ $972$ $272$ $480$ $725$ $331$ $82$ $541$ $866$ $94$ $444$ $71$	2					
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	2					
718 $900$ $73$ $756$ $818$ $773$ $473$ $531$ $661$ $459$ $108$ $572$ $516$ $200$ $452$ $209$ $620$ $34$ $665$ $770$ $477$ $603$ $892$ $537$ $491$ $887$ $683$ $131$ $678$ $68$ $143$ $466$ $623$ $697$ $303$ $11$ $583$ $734$ $651$ $998$ $331$ $954$ $645$ $226$ $733$ $181$ $637$ $210$ $529$ $456$ $916$ $620$ $562$ $93$ $76$ $210$ $'90$ $511$ $579$ $238$ $659$ $576$ $379$ $778$ $481$ $797$ $713$ $769$ $836$ $283$ $471$ $114$ $168$ $858$ $578$ $229$ $866$ $657$ $306$ $22$ $604$ $310$ $685$ $492$ $242$ $802$ $933$ $275$ $93$ $879$ $200$ $272$ $480$ $725$ $331$ $82$ $541$ $868$ $94$ $444$ $71$	5					
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	7					
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$	2	108				
603       892       537       491       887       537         683       131       678       68       143       537         466       623       697       303       11       537         583       734       651       998       331       537         954       645       226       733       181       537         637       210       529       456       916       66         620       562       93       76       210       537         90       511       579       238       659       55         576       379       778       481       797       57         713       769       836       283       471       57         114       168       858       578       229       9         866       657       306       22       604       8         310       685       492       242       802       5         933       275       93       879       200       5         541       868       94       444       71       4	1					
683       131       678       68       143       2466         466       623       697       303       11       2466         583       734       651       996       331       2466         954       645       226       733       181       2466         637       210       529       456       916       66         620       562       93       76       210       14         90       511       579       238       659       55         576       379       778       481       797       27         713       769       836       283       471       37         114       168       858       578       229       99         866       657       306       22       604       86         310       685       492       242       602       93         933       275       93       879       200       93       97         272       480       725       331       82       22       31       82       32         541       856       94       4444       71       4	4					
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	9					
583       734       651       998       331       531         954       645       226       733       181       537         637       210       529       456       916       66         620       562       93       76       210       16         90       511       579       238       659       55         576       379       778       481       797       57         713       769       636       283       471       57         114       168       858       578       229       93         866       657       306       22       604       8         310       685       492       242       802       200       54         933       275       93       879       200       54       541       863       94       4444       71       444						
954       645       226       733       181       537         637       210       529       456       916       66         620       562       93       76       210       1         90       511       579       238       659       55         576       379       778       481       797       57         713       769       836       283       471       53         114       168       858       578       229       59         866       657       306       22       604       8         310       685       492       242       802       542         933       275       93       979       200       52         272       480       725       331       82       541         541       868       94       4444       71       444					734	
637       210       529       456       916       6         620       562       93       76       210       1         90       511       579       238       659       5         576       379       778       481       797       5         713       769       836       283       471       5         114       168       858       578       229       9         866       657       306       22       604       8         310       685       492       242       802       9         933       275       93       979       200       9         272       480       725       331       82       2         541       868       94       444       71       4	3					
'90         511         579         238         659         576           576         379         778         481         797         2           713         769         836         283         471         3           114         168         858         528         229         9           866         657         306         22         604         8           310         685         492         242         802         933         275         93         979         200         9           272         480         725         331         82         2         2         444         71         4	6					
576       379       778       481       797       2         713       769       836       283       471       2         114       168       858       528       229       9         866       657       306       22       604       8         310       685       492       242       802       9         933       275       93       979       200       9         272       480       725       331       82       2         541       868       94       444       71       4	1					
713       769       836       283       471       32         114       168       858       578       229       9         866       657       306       32       604       8         310       685       492       242       802         933       275       93       979       200       9         272       480       725       331       82       32         541       868       94       444       71       4	5					
114       168       858       578       229       9         866       657       306       32       604       8         310       685       492       242       604       8         933       275       93       879       200       9         272       480       725       331       82       33         541       868       94       444       71       444	2					
866         657         306         22         604         8           310         685         492         242         802           933         275         93         879         200         92           272         480         725         331         82         22           541         868         94         444         71         444						
310     665     492     242     802       933     275     93     879     200     931       272     480     725     331     82     231       541     868     94     444     71     444					657	
933         275         93         979         200         93           272         480         725         331         82         23           541         868         94         444         71         444	•			492	685	
272         480         725         331         82         331	ê		879			
175 522 444 71 4	2					
	4	71				
455 120 400 555	2	692	936			
751 921 012 000 144 8	87					

•

•

. U	· <b>v</b>	<b>W</b>	x	" <b>¥</b>	z
576	403				
389	-03 646	310 842	472	465	16
930	522	304	187	721	994
494	914	50 <b>4</b>	168	165	666
63	621	10	195	529	939
266	262		639	306	355
748	646	849	977	110	612
562	725	924	885	0	5
405	889	805	418	26	106
980		398	115	717	494
147	255 301	483	684	130	333
843		739	- 550	185	634
858	882	110	385	321	327
216	871	154	142	786	300
200	642	995	494	57	658
	864	661	258	857	14
634 966	810	783	799	795	995
	181	648	853	804	743
683	463	478	181	4	341
681	637	220	166	429	210
95	912	161	485	910	98 <del>6</del>
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	405
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
			•••	004	- 30

•

### 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'
       - Relation for the 'SECOND CONDITION'
 * RR
 * 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;
    D\overline{O} I = 1 TO blksize WHILE( not eor(outer) );
       n_outer = n_outer + 1;
       u\overline{s}ed(I) = f\overline{a}lse;
       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</pre>
        R is 1>1 or 1>=1 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'
       - Relation for the 'SECOND CONDITION'
* RR
* 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;
 D\overline{O} 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 '>=' \overline{T}HEN
         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;
```

T

### 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'
        - Relation for the 'SECOND CONDITION'
 * RR
* 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;
  D\overline{O} 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;
 D\overline{O} 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 1>7 or 1>=1 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 + l 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'
        - Relation for the 'SECOND CONDITION'
 * RR
 * 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;
  D\overline{O} 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 *\overline{/}
/* 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 '>=' \overline{T}HEN
         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;
    ΙF
       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-l) > start inner ) THEN DO;
            IF WHERE(FIRST) on lookahd THEN DO;
              p inner = start inner + 1;
              \overline{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
<ol> <li>Equal join conditions</li> <li>Unique key with one predicate.</li> </ol>	sort/merge outer-equal-join sort/nested loop
3. Non-unique key with one predicate.	sort/merge
<ol> <li>Unique key with multiple predicates.</li> </ol>	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
<ol> <li>Equal join conditions</li> <li>Unique key with one predicate and θ</li> </ol>	sort/merge outer-equal-join nested loop
condition. 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 (Note: make the smaller r	either relations. elation the inner relation)
Conditions	Recommeded Methods
<ol> <li>Equal join conditions</li> <li>Unique key with '=' as the FIRST CONDITION.</li> </ol>	sort/merge outer-equal-join 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 (Tl,T2) where Tl.a > T2.a

Let Tl and T2 be the following relations:

Tl	т2
1 2 3 4	1 2 3 4 5
	6

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

(Tl and T Tl.a	2 sorted in T2.a	ascending	order)	
1 2 3 4	1 2 3 4 5 6			
T2.a	Tl.a	===>	Tl.a >	> T2.a
$\begin{array}{c}1\\2\\3\\4\\5\\6\end{array}$	$ \begin{array}{c} & & & \\ & & & & \\ & & & \\ & & & \\ & & & \\ & & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & $		1 2 3 4	? 1 1 1

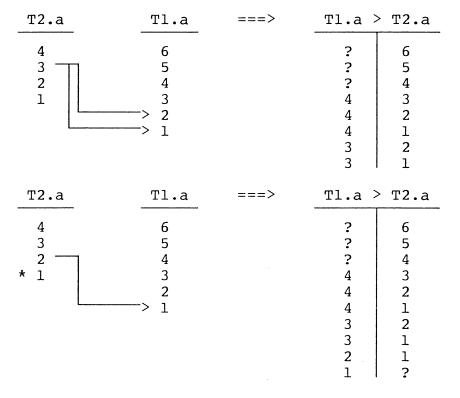
87

T2.a	Tl.a	===>	Tl.a >	> T2.a
$ \begin{array}{c} 1\\2\\3\\4\\5\\6\end{array} $	$\begin{array}{c}1\\2\\->3\\->4\end{array}$		1 2 3 4 3 4	? 1 1 2 2
T2.a	Tl.a	===>	Tl.a >	> T2.a
1 2 3 * 4 * 5 * 6	1 2 3 -> 4		1 2 3 4 3 4 4 ? ?	? 1 1 2 2 3 4 5 6

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

(Tl and T2 	2 are sorted T2	in descen	ding or	der)
4	6			
2 1	4 3 2			
	ī			
T2.a	T1.a	===>	Tl.a >	> T2.a
4 3 2 1	> 6 * > 5 * > 4 * > 3 > 2 > 1		? ? 4 4	6 5 4 3 2 1

88



\* 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:

STOREL				STOP	RE2
	S#	P#		S#	P#
(1) (5) (9) (14) (19)	s3 s5 s5 s8 s9	p1 p3 p7 p5 p2	(2) (3) (4,10,15) (6,11,16) (7,12,17) (8,13,18)	s1 s2 s5 s5 s5 s7	p2 p6 p4 p9 p8 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	Res	ults of	operations
<pre>(1) and (2) (1) and (3) (1) and (4) (5) and (4) (5) and (6) (5) and (7) (5) and (8) (9) and (10) (9) and (11) (9) and (12) (9) and (13)</pre>	? \$5 \$5 \$5 \$5 \$5	p3 s5 p3 s5 p3 s5 p3 s5 p7 s5 p7 s5	? p4 p9 p8 p4 p9

(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#
?	?	sl	p2
?	?	s2	p6
s3	pl	?	?
s5	р3	s5	p4
s5	р3	s5	р9
s5	р3	s5	p8
s5	p7	s5	p4
s5	p7	s5	р9
s5	p7	s5	p8
?	?	s7	p8
s8	р5	?	?
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:

	STO	ORE1	_	STO	RE2	_
	S#	Р#		S#	P#	
(1) [ (5) [ (6) [ (8) [	s3 s5 s5 s8 s9	p1 p3 p7 p5 p2	]       (2) [         ]       (3) [         ]       (4) [         ]       (7) [	sl s2 s5 s5 s5 s5	p2 p6 p4 p9 p8 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	of the operatic			
(1) (1) (1)	&	(3)	==>		?	? ? pl	s2	p6	

(5)	&	(4)	==>	s5	р3	s5	p4
				s5	p3	s5	p9
				s5	3	s5	8
				s5	p7	s5	p4
				s5	p7	s5	p9
				s5	p7	s5	p8
(6)	&	(7)	==>	?	?	s7	р9
(6)			==>	s8	p5	?	?
(8)			==>	s9	p2	?	?

The resulted output:

STORE1.s#	STORE1.p#	STORE2.s#	STORE2.p#
?	?	sl	p2
?	?	s2	p6
s3	pl	?	?
s5	р3	s5	p4
s5	р3	s5	р9
s5	р3	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(filel) INTO(tuplel); DO WHILE( NOT eofl); 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 tuplel and tuple2 did not mark used THEN DO; WRITE FILE(file3) FROM(nulls1 | tuple2); END; READ FILE(file2) INTO(tuple2); END; IF not tuplel 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.