1998

An Experimental Study Into the Effect of Varying the Join Selectivity Factor on the Performance of Join Methods in Relational Databases

Ada Mallet
Edith Cowan University

Recommended Citation

Mallet, A. (1998). *An Experimental Study Into the Effect of Varying the Join Selectivity Factor on the Performance of Join Methods in Relational Databases*. https://ro.ecu.edu.au/theses_hons/784

Edith Cowan University

Copyright Warning

USE OF THESIS

An Experimental Study into the Effect of Varying the Join Selectivity Factor on the Performance of Join Methods in Relational Databases

by

Ada Mallet

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Award of

Bachelor of Science Honours (Computer Science)

Faculty of Science, Technology and Engineering
Edith Cowan University

Date of Submission: 12 February 1998

ABSTRACT

Relational database systems use join queries to retrieve data from two relations. Several join methods can be used to execute these queries. This study investigated the effect of varying join selectivity factors on the performance of the join methods. Experiments using the ORACLE environment were set up to measure the performance of three join methods: nested loop join, sort merge join and hash join. The performance was measured in terms of total elapsed time, CPU time and the number of I/O reads. The study found that the hash join performs better than the nested loop and the sort merge under all varying conditions. The nested loop competes with the hash join at low join selectivity factor. The results also showed that the sort merge join method performs better than the nested loop when a predicate is applied to the inner table.

DECLARATION

I certify that this thesis does not, to the best of my knowledge and belief:

a) incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education;
b) contain any material previously published or written by another person except where due reference is made in the text; or
c) contain any defamatory material

[redacted]

Signature
Date 12/02/98 .

ACKNOWLEDGEMENTS

I wish to thank my supervisor, Jean Hall, who devoted hours of her time rigorously checking the work. Jean is friendly and easy to work with. Her deep interests in that area of study have been a great encouragement.

I also wish to thank Ms Penny Cookson of SAGE Professional Services for the helpful Oracle database performance tips. Special thanks to Steve Schupp of Winthrop Technology for his assistance in the UNIX language. I also wish to thank the Oracle International Support Analysts for providing solutions to the software problems encountered in that study.

I extend my gratitude to my dear husband, Désiré, who provided a shelter of love, friendship and encouragement throughout my studies. Désiré has been a great support through his presence while I worked till late at night.

TABLE OF CONTENTS

LIST OF FIGURES

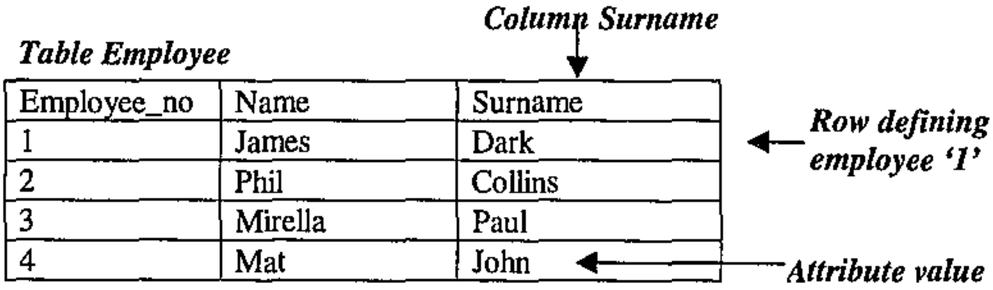
LIST OF TABLES

DEFINITION OF TERMS

1. Block	Unit of transfer between the secondary and primary memory.
2. Cartesian product	Consider two relations R and S each with n and m number of tuples respectively. The cartesian product of these two relations will concatenate each tuple producing a resulting relation with $(n * m)$ tuples.
3. Degree of Relationship	A degree of relationship of 'n' implies that a tuple from one relation relates to a minimum of zero and a maximum of 'n' tuples from the other relation at any point in time.
4. Join selectivity factor	The ratio of the number of tuples participating in the join to the total number of tuples present in the Cartesian product of the relations (Mishra & Eich, 1992). For example, consider the join of two relations consisting of 100 and 1000 tuples respectively. Assuming that 100 tuples satisfy condition 'x'. A cartesian product of these two relations will consist of 100,000 tuples. If the join condition 'x' is applied to the cartesian product, then only 100 tuples will be returned. Hence, the join selectivity factor is 100 / 100000.
4.1 Low join selectivity Factor	Number of tuples participating in the join is less than 10% of the maximum number of tuples that could participate in the join.
4.2 High join selectivity Factor	Number of tuples participating in the join is greater than 60% of the maximum number of tuples that could participate in the join.
5. Predicate	A relational operation that applies a condition so that only tuples satisfying this condition are returned. A predicate is used in the WHERE clause of a SQL statement.
6. Relation	The relational model treats a set as a relation. The relation is a logical view of the data. It is a set consisting of a number of tuples.
6.1 Small relation	Size of relation is less than 400Kb.

6.2 Large relation	Size of relation is greater than 400Kb.
6.3 Inner relation	The inner relation refers to the larger relation in the join relationship.
6.4 Outer relation	Outer relation refers to the smaller relation in the join relationship.
6.5 Result Relation	The join operation is used to combine related tuples from two relations into single tuples that are stored in the result relation.
7. Table	The relational database system models the relational set as a table. The table is also referred as the relation. A table is a group of related data and is made up of rows and columns.
7.1 Attribute	Smallest unit of data in the relational model.
7.2 Column	The column contains a particular field value.
7.3 Row	The row is a unique entry in a table. A row consists of all the data that identifies an entry in a table.
8 Tuple	The collection of values that compose one row of a relation.

Table 1: Different terms used to define a table



Employee_no	Name	Surname
1	James	Dark
2	Phil	Collins
3	Mirella	Paul
4	Mat	John

Chapter one: Introduction

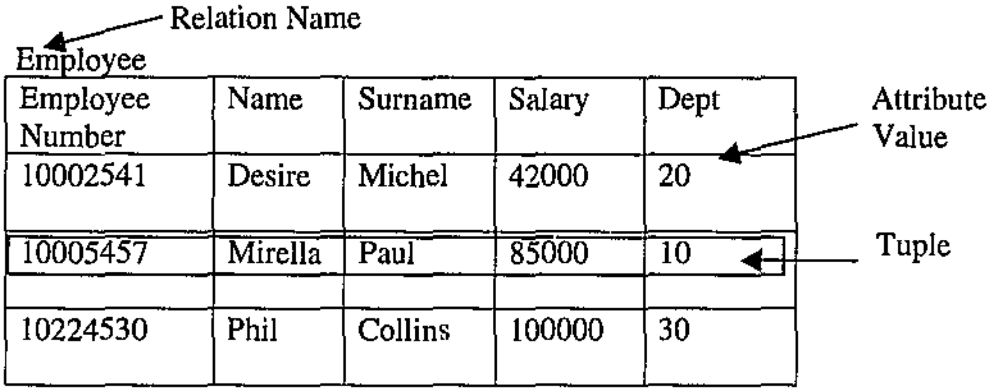
The Background to the Study

The Relational Model

In 1970, E. F. Codd, a researcher at IBM, published a seminal paper on the relational data model (Codd, 1970). The model described in this paper was based on mathematical set theory and it offered an enormous advancement over previous database models. The relational model differed from other database models because the logical view of data was completely independent from the physical view. This independence meant that programs manipulating data were not affected by changes to the internal data representations, such as changes to file organisation or access paths. In traditional systems, the program is dependent on the data files as the description of the data and the way to access the data is built in the application system (Mc Fadden & Hoffer, 1991).

Data in the relational model are organised as units of data storage known as relations or tables. A relation consists of a collection of similar pieces of information (Bennett, Ferris & Ioannidis, 1991). It is a set consisting of a number of tuples (also known as records or rows). A tuple comprises of a number of attributes and the values of these attributes are based on a domain. The attribute is the smallest unit of data in the relational model. For example, consider a relation named Employee. This relation consists of the attributes such as employee number, name, surname and salary. The tuple refers to the collection of data that defines an employee.

Table 2: Terms used in the relational model



Employee

Employee Number	Name	Surname	Salary	Dept
10002541	Desire	Michel	42000	20
10005457	Mirella	Paul	85000	10
10224530	Phil	Collins	100000	30

The relational model provides mathematical operations and constraints that can be applied to tables in databases. Codd (cited in Topor, n.d.) proposed two languages to access data from the relational database system: the relational calculus and the relational algebra. However, these languages did not provide facilities for database definition or database update. In the late 1970's, Structured Query Language (SQL) was developed to add some necessities lacking in the previous languages. SQL provided facilities for querying the database as well as facilities for defining the database, manipulating and controlling the data in a relational database (Date, 1989).

Query Optimisation

The great power and capability of the relational model have enabled the emergence of commercial Relational DataBase Management Systems (RDBMS) such as Oracle, Ingres, Sybase and DB2. A RDBMS is a controlled collection of programs based on a single relational data model allowing authorised access to data queries, additions, deletions and modifications in a reliable, efficient and flexible way (Topor, n.d.). Relational applications may contain large volumes of data, and the retrieval of data needs to be efficient especially for on-line transaction processing.

According to Date (1986, p. 67), the performance of a transaction is determined by the number of I/O (Input/Output) operations and the amount of CPU (Central Processing Unit) processing. During execution of a query statement such as a SQL statement, the query optimiser will select the strategy with the least processing cost from the many execution strategies. The optimal strategy is usually determined by calculating the cost of different available strategies in terms of some combination of processing load and disk I/O accesses. The selection of the most efficient strategy to access the data and answer the query is known as 'query optimisation' (Bennett et al., 1991).

Access Path

The JOIN operator is used to retrieve data when at least two relations are involved in a query statement. It "permits two relations with at least one comparable attribute to be combined into one" (Jarke, Koch & Schimdt, 1985, p. 11). For example, a join between the relations 'Department' and 'Employee' is possible using the join attributes 'dept no' present in both relations.

The JOIN operator is a costly operator because of the many alternative strategies that must be analysed during join query processing. The optimal execution strategy is dependent on factors such as the order of the operations defined on the relations as well as the access path used. The access path refers to the "data structures and the algorithms that are used to access the data" (Meechan, 1988, p. 4). There are three main types of access path used in relational systems: indexed, sequential, hashed access paths.

Indexed Scan

The indexed scan uses a B-tree structure to read the values of the indexes. The node of the tree represents the pages of the index. Each leaf page consists of an index key value and the physical address of the row in the table where that value for that key is stored. A search through the tree always starts at the root and descends through the leaves until the required value is found. If the value is not found in a terminal node, then that value does not exist in the tree. Figure 1 is a schematic representation of how an indexed scan works.

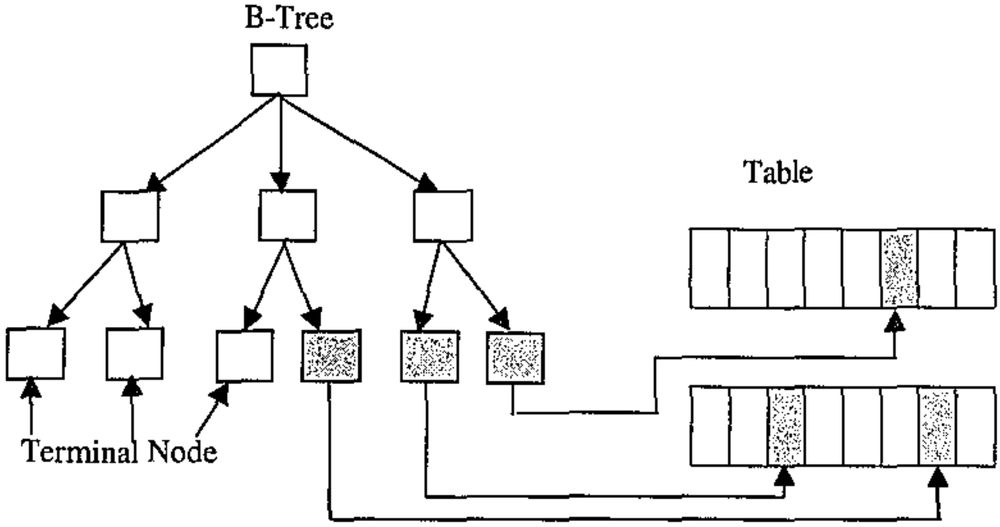


Figure 1: Indexed Scan

Sequential Scan

A sequential scan reads one row of a table at a time until the required value is found or the end of the table is reached.

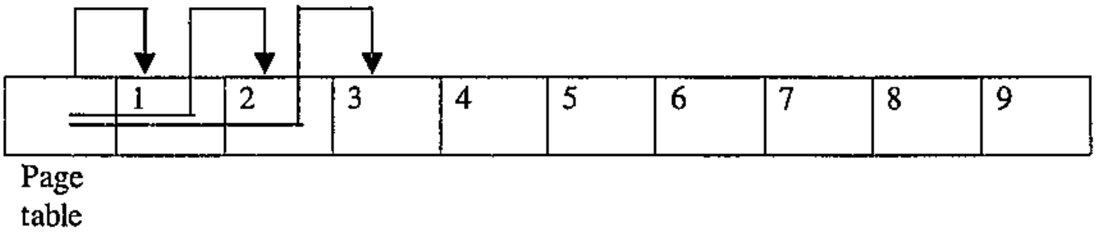


Figure 2: Sequential Scan

Hash Scan

A hash scan provides direct access to the data block containing the record by applying a hash function or transformation operation to the record key value and the number of primary pages (Gardarin & Valduriez, 1989). A set number of pages (called the primary pages) and overflow pages are defined for the hash structure. A hash function is used to compute the physical address for the primary page on which the row in the table should be stored. During a hash scan, the same algorithm that was used to store the row in the table is used to get the physical address of the primary page. This page is searched for the row with the matching hash key. If the row is not found, then the overflow page or pages associated with that primary page are examined. The figure below illustrates the workings of the hash scan.

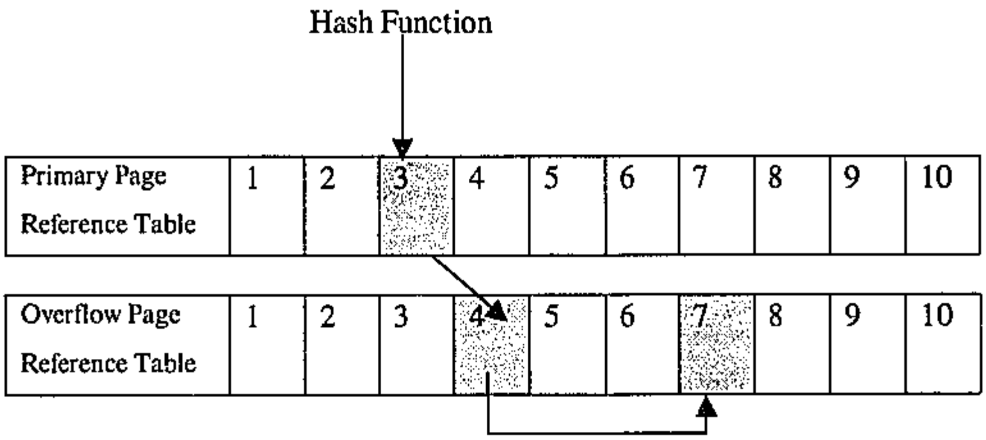


Figure 3: Hash Scan

Join Method

Data are retrieved from two or more relations using a join method. There are three main join methods: nested loop, sort merge and hash join. The nested loop join performs an indexed scan on one of the relations, usually the larger relation. The sort merge join method sorts both relations and then merges the two relations using the matching tuples as the selection criteria to produce the resulting relation. The hash join applies a hash function to the key columns of one relation and store these hash values in the hash table. The record key value of the other relation is then hashed using the same hash function and a hashed scan is then performed on the hash table. The table below summarises the type of access path used by each join method.

Table 3: Access paths used by different join methods

Join Method	Nested Loop	Sort Merge	Hash Join
Access Path	Index Scan	Sequential Scan	Hash Scan

Each join method performs differently depending on factors such as the size of the relations, the number of rows retrieved from the relations and the degree of relationship (a degree of relationship 10 implies that a tuple in one relation relates to a maximum of 10 tuples from the other relation). The choice of the optimum join method for a particular set of conditions can significantly reduce the join query processing time.

Join Query Processing

The following example illustrates the importance of query optimisation:

Consider the case where a customer can have many orders and an order is for one customer. Assuming that there are 100 customers and 1000 orders.

Customer(<u>cust_id</u>, cust_name)

Order(<u>order_no</u>, order_desc, *cust_id*)

Consider the execution of the following query where there are 20 tuples with a customer id of > 1000:

```
SELECT cus.cust_id, ord.order_desc
FROM Customer cus, Order ord
WHERE cus.cust_id = ord.cust_id
AND cus.cust_id > 1000
```

There are two ways to process this query:

1. The two relations are joined first over 'cust_id' and a resulting relation of (100 * 1,000) tuples created. The selection is then done against the resulting relation. In this case, 100,000 comparisons are required.
2. The join condition is applied to the customer table. In this case, 20 tuples with 'cust_id' > 1000 are returned as a temporary relation. The join over 'cust_id' is

then performed between the temporary relation and the order relation.

Therefore (20 * 10,000) or 20,000 comparisons are required.

The second alternative is the preferred strategy providing a quicker way to process the query.

The Significance of the Study

The projected increase in database applications and the volume of transactions to be processed (Database Market, 1997) have accentuated the need to consider performance issues carefully. The recent introduction of the hash join method in commercial database systems such as Oracle has also triggered the need to investigate the performance of the hash join compared to the two common join methods: nested loop and sort merge.

This research has provided relevant information concerning the performance of the join methods under varying join selectivity factors (Refer Definition of Terms - 4) and for different degrees of relationship (Refer Definition of Terms - 3). This study also considered the behaviour of the join methods when a predicate is applied to the inner table.

The Purpose of the Study

This study considered the effect of the join selectivity factor on the performance of the join methods in relational database systems when the number of rows satisfying a join condition varies. A set of experiments was designed to capture the time taken for a query using different join methods to retrieve data. The study also examined the sensitivity of the elapsed time, CPU time and logical I/O reads when the number of tuples being retrieved from the outer relation varied (See Definition of Terms – 6.4). The sensitivity of the elapsed time to the join selectivity factor when the degree of the relationship varies was also examined.

Research Questions

Main Question

There are several factors that impact on the performance of the join methods. This study examines the effect of the join selectivity factor on the performance of the nested loop, sort merge and hash join methods when the degree of relationship varies and a predicate is applied to the inner table.

How do the nested loop, sort merge and hash join perform when the join selectivity factor varies under certain conditions?

Sub Question 1

What is the effect of the join selectivity factor on the performance of the nested loop, sort merge and hash join methods for a one-to-one and a one-to-many relationship?

Hypotheses

Note: The response time is the total time taken by a query statement to retrieve data from the database.

H_1 For a one-to-many relationship with a low join selectivity factor, the nested loop has a faster response time than the sort merge join method.

H_2 For a one-to-many relationship with a low join selectivity factor, the hash join has a faster response time than the sort merge join method.

H_3 For a one-to-many relationship with a high join selectivity factor, the sort merge has a faster response time than the nested loop join method.

H_4 For a one-to-many relationship with a high join selectivity factor, the hash join has a faster response time than the sort merge join method.

H_5 For a one-to-one relationship with a low join selectivity factor, the nested loop has a faster response time than the sort merge join method.

H_6 At high join selectivity factor, the nested loop join method has a faster response time for a one-to-one relationship than for a one-to-many relationship.

Sub Question 2

What is the effect of applying a predicate to the inner relation on the performance of the join methods when the number of tuples selected from the outer relation varies?

Note: The inner relation refers to the larger relation in the join relationship and the outer relation refers to the smaller relation. A predicate is basically an operation (e.g., equality operator) that can be applied to attributes in a relation so that the tuples being retrieved from the relation are selective.

H_7 The sort merge join method with low selectivity of the outer relation gives a faster response time when a predicate is applied to the inner relation than when no predicate is applied.

H_8 The sort merge join method with high selectivity of the outer relation gives a faster response time when a predicate is applied to the inner relation than when no predicate is applied

Assumptions:

The study is based on the following assumptions:

- A small relation is assumed to be a table that fits into the buffer cache and therefore can be read in one physical read.

- A large relation is assumed to be larger than the buffer cache.
- An index is defined on the join column of the inner (large) relation.

Chapter Two: Literature Review

Query Optimisation

Join query processing has been studied from several different points of view: (Jarke, M. & Koch,J., 1984, Kim et al. 1985, Yu, P. & Cornell, W., 1991, Harris, E. 1995)

a) query optimisation

b) optimising I/O and buffer space

c) hardware support such as a join processor

d) parallel processing

e) physical database design

Join query optimisation in relational database systems attempts to find the optimal execution strategy for a join query. Query processing has two main phases: compilation and execution. Compilation consists of operations such as parsing the statement, checking its syntax and mapping the logical-level names to physical-level address. Execution consists of tasks such as retrieval and manipulation of data. The operations involved in execution are choosing an access strategy, checking access to data and generating machine code.

When a query is executed, there are many possible execution strategies that can be considered. The cost of each execution strategy is calculated and the strategy with the least cost is chosen (Li, Kitigawa & Ohbo, 1994). The cost is the sum of the costs of processing each individual operator and is measured in terms of CPU time and/or I/O time. During query optimisation, factors such as the ordering of database operations, the access paths and the algorithm used to perform database operations are considered (Kuznetsov, 1989).

Query Optimiser - Cost-based v/s Rule-based

Early query optimisers developed for the System R Database Management System (DBMS) used a simple cost function to estimate the best execution strategy based on CPU operation and number of I/O accesses.

Cost Function = Time to perform CPU operation x Number of CPU operations
+
Time to perform I/O operation x Number of I/O operations.
(Meechan, 1988)

The strategy resulting in the least value of the cost function was selected as the best execution strategy. Today's DBMS systems make use of the rule-based or cost-based optimiser. The rule-based optimiser bases the execution plan on some pre-defined rules. These rules allow the optimiser to determine whether to perform an indexed scan or a full table scan. The cost-based optimiser chooses the optimal execution plan based on flexible rather than on rigid rules. It considers database variables such as the relation size, the selectivity of the index, the amount of clustering of data to find the best execution path. The rule-based optimiser is sensitive to the order in which the tables are specified in a query. It does not consider the statistical distribution of data in the tables being accessed and therefore performs poorly with complex queries involving many tables (Roti, 1996). The query optimiser needs to have access to the relevant statistics about the tables and the join condition to determine the right join method. The ratio of the number of tuples to be retrieved from a relation to the total number of tuples that exists in that relation, that is the selectivity factor is an important factor that is considered by both types of optimisers for selection of the optimum execution strategy.

Join Operator

The join operator is provided by the relational algebra as defined by Codd (1970). It is used to combine data from two relations. A more precise definition is given by Stanczyk (1991), who defines the join as the combination "of tuples from two operand relations that are related via a common attribute(s)". When more than two relations are involved, the join is said to be a multiway join. A multiway join processes the join as a series of joins between two relations.

Relational algebra is useful to define new relations as it offers a wide range of operations. In the relational algebra, the expression R(A1,A2, ..., An) denotes a relation named R with attributes A1, A2,...,An. The attribute value is based on a domain. The domain defines the set of possible values that an attribute can contain (Atzeni & De Antonellis, 1993). The relation maps to a table in the database. The rows of the table correspond to the tuples $<a_{1,k}, a_{2,k},\ldots,a_{n,k}>$ in the relation.

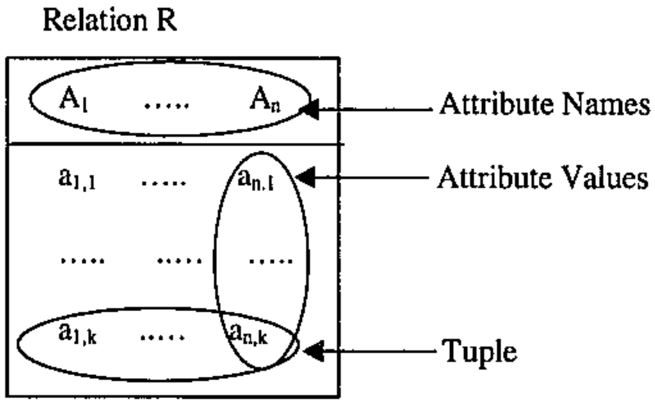


Figure 4: Attributes and tuples as expressed in relational algebra.

The relational algebra contains operations such as union, difference, product (Cartesian operation), theta-selection, projection, intersection, division and join. The join operation is an essential operation of the relational algebra. The most common join between two relations is the natural join. The natural join is implemented using the Cartesian product. For example, when a relation R with n tuples is joined to another relation S with m tuples, a result relation with (n x m) tuples is built. The theta join is a natural join that allows for operators to be defined on the relations (Pascal, 1993). If the equality operator is applied between two attributes, then the join can be further defined as the equality join.

The theta join of two relations R and S is written as:

$$R \bowtie_{r(a)\,\theta\,s(b)} S$$

where r(a) θ s(b) defines the join condition between two relations R and S. The figure below shows the result of joining the relations R and S with the following join condition:

$$R \bowtie_{r(level) > s(level)} S$$

Relation R

Employee No	Emp Name	Level
10000201	Mirella Paul	5
10000245	Phil Collins	3
10002441	Desire Lyn	4
10000287	Anu Hall	2

Relation S

Level	Description
2	Clerical
3	Valuation
4	Marketing
5	Management

Employee No	Emp Name	Level	Level	Dept Name
10000201	Mirella Paul	5	2	Clerical
10000201	Mirella Paul	5	3	Valuation
10000201	Mirella Paul	5	4	Marketing
10000245	Phil Collins	3	2	Clerical
10002441	Desire Lyn	4	2	Clerical
10002441	Desire Lyn	4	3	Valuation

Figure 5: Resulting relation from applying a theta join to R and S

The join operator is the most important and expensive operation in relational database systems (Harris, 1995). This view is also shared by Li, Kitawaga and Ohbo (1994) who state that the join operator is "indispensable in processing many ad-hoc queries" (p. 648). The join operator needs to perform efficiently as it is used extensively in relational query processing (Mishra & Eich, 1992). The join operator is also the most difficult to process and optimise because of the number of possible factors affecting this operator (Bennett et al., 1991). The number of tables to be joined, the access paths and the join method used are some of the factors that need to be considered in join-type query optimisation (Bennett et al., 1991). Indeed, the choice of the right join method can offer a significant reduction in the cost of the query (Cheng et al., 1991).

Join Methods

The join method determines the way that the individual joins are processed when a query is optimised. The three types of join method considered in this study were: the nested loop, sort merge and hash join.

Nested Loop

The nested loop join is the simplest join method. It exploits the use of an index in the inner relation (i.e., the larger relation). Each tuple of the outer relation, that is, the smaller relation, is read and compared with all tuples in the inner relation that satisfy the join condition to produce a result relation. The algorithm is as follows:

```
While there are unread tuples in the outer relation
    read tuples from the outer relation into buffer $B_1$
    seek to the beginning of the inner relation
    while there are unread tuples in the inner relation
        read tuples from the inner relation into buffer $B_2$
        inner loop
        for each tuple $r_1$ in $B_1$
            for each tuple $r_2$ in $B_2$
               if $r_1$ and $r_2$ satisfy the join condition
                   place the resulting tuple in buffer $B_R$
                   if the buffer $B_R$ is full, write it to the result relation.
```

(Harris, 1995, p. 25)

In order to increase its performance, the nested loop join is usually implemented as a block read for the outer relation instead of a tuple read. This implementation helps to minimise the number of physical I/O accesses. The nested loop join takes advantage of the indexed inner relation. Blasgen and Eswaran (cited in Harris, 1995, p. 25) have implemented a nested loop algorithm that holds as many records as possible from the outer relation in main memory. 'Rocking' was introduced to improve the efficiency of the nested loop (Kim, 1980). Rocking refers to when the inner relation is read from top to bottom for an outer relation and from bottom to top for the next outer relation. This technique reduces the number of physical I/O accesses on the inner relation since the blocks that have been read from the inner relation are still in memory when the next pass through the outer relation occurs.

The cost of the nested loop is O(n x m) time where n and m are the number of tuples in each relation for a simple implementation of the nested loop.

Sort Merge

The sort merge join makes use of sequential access. It works in two phases: a sorting phase and a merging phase. Both relations are sorted first in order of the join attributes, then the relations are scanned and finally tuples with matching join attributes are merged. The algorithm below applies for equijoin.

```
Sort Phase

Sort tuples in relation R on join attribute r(a)
Sort tuples in relation S on join attribute s(b)

Merge Phase
Read first tuple from relation R
Read first tuple from relation S
For each record of relation R do
        {While s(b)< r(a) then
        read next record of relation S
        If r(a) = s(b) then
                join r and s
                place record in resulting relation Q }
```

(Mishra & Eich, 1992, p. 73)

The performance of this join method is sensitive to whether the join column contains unique values or not. Non-uniqueness means that several passes through the inner relation are needed and consequently additional input output accesses are required (Yu & Cornell, 1991, p. 624).

Consider the case where relation R contains two tuples r1 and r2 with a join attribute value 'x' and similarly, relation S contains three tuples s1, s2 and s3 with the same join attribute value 'x'. Using the above algorithm, tuple r1 is first read and tuples s1, s2 and s3 are then read from the inner relation. When tuple r2 is read, then the tuple following s3 will be read. In this case, the resulting relation will not include the join of tuple r2 with s1, s2 and s3 (Mishra & Eich, 1992).

The above algorithm can be modified so as to record the position where the read to the inner loop started. Non unique join attribute values can then be accommodated in the join algorithm. When a duplicate value is found, backtracking to the recorded position occurs. If the buffer size is small and the sorted data does not fit in the buffer size, then more I/O is required as data will be fetched from disk to memory frequently.

In the late 1970's, investigation by Blasgen and Eswaran (cited in Graefe et al., 1994), concluded that the sort merge join was the most efficient join when large tables were involved. They noted that the time required to perform a sort merge was mainly dependent on the sorting time rather than the merging time. Mishra & Eich (1992) also confirmed that the sorting time determined the overall execution time. Therefore, if the relations are already sorted, the time to process a sort merge join can be minimised. The complexity of this method is based on the sort time and is given as O(n log n) time for each relation where n is the number of tuples in the relation.

The performance of the sort merge join is dependent on the number of passes required during the merge phase. "Each additional pass means reading in and writing out the relation one more time" (Yu & Cornell, 1991, 624).

The sort merge sorts both relations on the join attribute and then merges the results using the matching tuples as the selection criteria. Reducing the number of passes required to merge the pages can increase the performance of this join method. The number of passes depends on the 'number of way merge' (number of pages that can be merged in a pass) provided by the sort merge algorithm. An 8-way sort merge algorithm with 16 pages to be merged will be merged in 3 passes: one pass to merge the first eight pages, another pass to merge the next eight pages and a final pass to merge the two eight pages. Alternatively, if the sort merge uses a 16-way merge, then the number of passes can be reduced to a single pass.

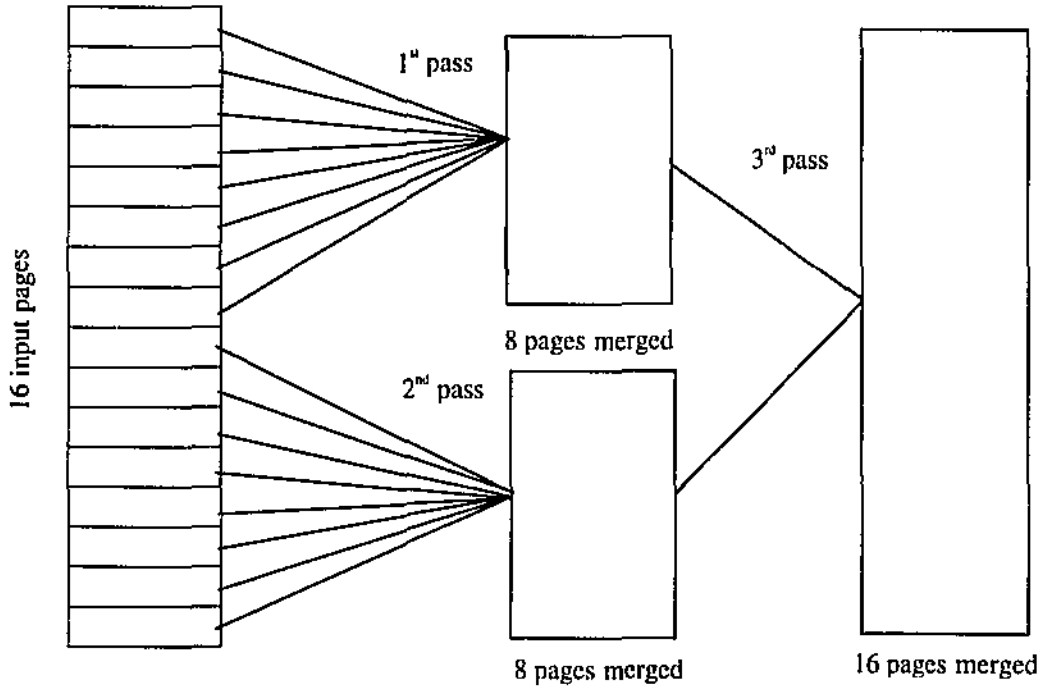


Figure 6: Normal Merging

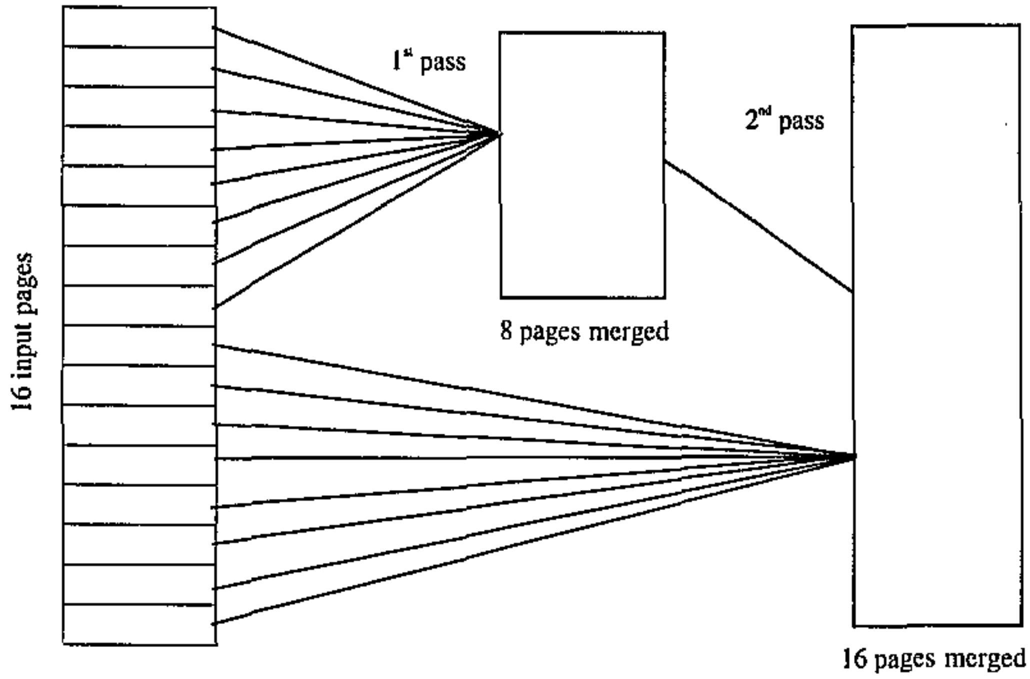


Figure 7: Delayed Merging

Similarly, the number of passes can be optimised by delaying the merge until all the pages are read (Graefe et al. 1994). For example, if a delayed merge was considered for sixteen input pages and using an 8-way merge algorithm, then only two passes would be required: one pass to merge the first eight pages, and the next pass to merge the output with the remaining eight pages (See Figure 6 and Figure 7).

Hash Join

The simple hash join works in two phases. During the first phase, tuples from one relation (the smaller relation) are read and a hashing function is applied to the join attributes to form a hash key. The hashing function considers the page location and the join attribute(s) to form the hash key. This key can then provide direct access to the required page. A hash table containing these hash keys is kept in main memory. In the second phase, tuples from the other relation are "hashed on the join attribute and the hash table is probed for matches" (Graefe et al., 1994, p. 935). When a match is found, tuples from the two relations are concatenated and added to the resulting relation (Yu & Cornell, 1991). A simple algorithm is as follows:

```
For each tuple in relation S do
        {hash on join attributes s(b)
        place hash value in hash table}
For each tuple in relation R do
        {hash on join attributes r(a)
        if r hashes to a nonempty bucket of hash table for S then
                {if r matches any s in bucket
                join r and s
                place in resulting relation Q}}
```

(Mishra & Eich, 1992)

The complexity of this method is found to be O(n+m) time where n and m are the number of tuples in each relation. The performance of this method is also dependent on the hashing function used. Other authors describe several flavours of the hash join, for example, GRACE hash join (Harris, 1995) and hybrid hash join (Cheng et al, 1991). The hybrid hash join makes use of an index to read the values. Each of these methods was implemented with the aim of improving the performance of the hash join. The GRACE hash join method takes O(n+m)/k time where k is the number of partitions in memory and (2 x k) processors are used (Kitsuregawa, cited in Mishra and Eich, 1992). If the hash table fits in main memory, then the hash join can compete with the sort merge and the nested loop (Aronoff, Loney & Sonawalla., 1997; Gaede & Gunther, 1994; Graefe et al., 1994;

Harris, 1995).

The hash join can have the advantage over the nested loop: a "single scan of the input relations is required if one of the two relations can be completely contained in memory" (Harris, 1995, p. 28). A hashing function is applied to the join attributes of each tuple of the outer relation. The hash key formed is placed in a hash table or 'bucket'. For each tuple of the inner relation, the join attribute value is hashed using the same hashing function. If the values hash to a bucket that contains values, that is, a non-empty bucket, then the tuples satisfy the join condition.

Selectivity Factor

The selectivity factor refers to the ratio of the number of tuples retrieved from a relation to the total number of tuples in that relation. Similarly, the join selectivity factor refers to the proportion of tuples retrieved from the Cartesian product of two relations that satisfy the join condition (Gardarin & Valduriez, 1988). The query optimiser uses the selectivity factor to estimate the size of a query and consequently plan the execution of the query effectively (Lipton, Naughton & Schneider, 1990). Research is continuing on efficiently estimating a query size. Both parametric and non-parametric methods have been proposed (Lipton & Naughton, 1990). A high selectivity factor requires a large number of tuples to be compared and hence produces a large result relation. The large amount of space required by the result relation implies that a high number of blocks are needed. Consequently, a high number of I/O accesses is expected.

Literature on Previous Findings

In an attempt to derive heuristic rules for query optimisation, Meechan (1988) investigated the effect of the join selectivity factor and the buffer availability on the response time and CPU time for Nested Loop and Sort Merge joins. He conducted the experiments using R* (an extension of System R DBMS) and suggested that further investigation using other system configurations was necessary. He concluded that the nested loop was more efficient than sort merge at low join selectivity factor.

Some authors have alternate views. Mishra & Eich (1992) considered the nested-loop to be the most inefficient join method at low join selectivity factor. They also noted that the performance of hash join decreases as the selectivity factor increases. These conflicting views suggest that the performance of join methods at low join selectivity factor need to be further investigated.

Researchers at the Database Technology Institute at IBM compared the performance of hybrid join, nested loop join and the sort merge join in a DB2 environment by varying the selectivity of outer table (Cheng et al., 1991). They concluded that "merge join is most often the best when qualifying rows of inner and outer table are large and the join predicate does not offer much filtering" (Cheng et al., 1991, p. 171).

The current research aimed to investigate how the join methods perform at varying join selectivity factor. The performance of the join methods using a different relational environment (Oracle database system) than the experiments described above (system R and DB2) was considered.

Summary

The join operation is a very important and expensive operation. The join method is influenced by a number of variables such as the selectivity, the size of the tables, the clustering of data in the table and the distribution of data in the table (Pascal, 1993). Many authors have indicated that the join selectivity factor is a key component in join-query optimisation. Reports in the literature investigating join methods have focused on the sort merge and the nested loop join methods. Hash joins were seldom considered in previous studies as large main memories were required for optimal performance. There is disagreement over which join method is the best at low join selectivity.

Chapter Three: Method

This chapter describes the model that was used to carry out the current experiments, data collection procedure and analysis. Throughout this chapter the term table and relation are used interchangeably.

Experiments conducted by Lu and Carey (1985) considered how distributed join algorithms performed in a local network. The effects of varying relation sizes, join selectivities and join column value distributions on the performance of eight different distributed join algorithms were investigated. Furthermore, the methodologies used by the researchers at the Database Technology Institute (1992) were noted. The following issues were noted:

- The relation sizes used in the experiments were 1000 tuples and 10,000 tuples.
- Enforcement of random values in join columns. This is necessary to ensure a fair comparison of the join methods. Sort merge join algorithm performs an internal sort and therefore the sort processing time is less for unsorted join columns than sorted join columns.

The current experiments were designed in light of the above considerations.

Experimental Environment

The experimental environment consisted of a workstation running Personal Oracle (version 7.3.2.2.1) for Windows NT 4.0 with a single 486 processor, 32 MB RAM and 1GB hard disk space. The environment was used to compare the performance of the three common join methods: nested loop, sort merge and hash join, under varying selectivities. The following timings were recorded when a join query statement was executed for varying selectivities using different join methods:

- total elapsed time (response time)
- time spent in memory (CPU time)
- number of disk accesses

Database Setting

A limited buffer size was necessary to ensure that the large relation could not be contained in the buffer cache. The database buffer was limited to 200 blocks where each block occupied 2 KB. The hash join algorithm performs well if both relations can fit in memory. In order to ensure an unbiased treatment of the join methods, the size of the buffer cache was limited so as to ensure that the large relation could not fit in the buffer cache. The number of blocks to be transferred in one physical read was limited to 16 blocks or 32 KB of data. If more blocks were to be fetched from disk to memory in one physical read, then less I/O would have been required.

Tables and Columns Settings

A join always involves two relations and the experiments considered the join between a small and a large relation. The small relation (or the outer relation) was defined as occupying less than 32 KB and the larger relation (or the inner relation) as occupying more than 32Kb. The experiments consisted of a small relation of 1000 tuples requiring a storage space of 30Kb. The large relation consisted of 10,000 tuples and occupied 500Kb. Both tables consisted of four columns.

Table 4: Table and Column Settings

Table	Column	Data Type	Key	Domain	Special values
Outer	Column1	Number (5)	Primary key	1 – 1000	Unique random values
	Column2	Char (10)			
	Column3	Char (10)			
	Column4	Number (4)		6001 – 7000	Unique random values
Inner	Column1	Number (6)	Primary Key	1000 – 11000	Unique values
	Column2	Char (30)			
	Column3	Number (7)		1 – 50000	
	Column4	Number (5)	Foreign Key	1 – 1000	Random values

The table above shows the values contained in the columns. An index was defined on the column4 on the inner table as the nested loop join performs an index scan on the inner relation.

Procedure

Initialisation of Variables

In order to obtain performance timings, several variables were initialised both at the database level and session level. The database initialisation file (Appendix A) was modified so that statistics were collected when a query statement was run. This was achieved by adding the following lines to the database initialisation file.

- TIMED_STATISTICS set to TRUE to enable collection of timed statistics such as CPU and elapsed time.
- USER_DUMP_DEST specifies the directory name on the file system where the trace files are generated. This was set to c:\amallet\trace.

Tracing was switched on for the session so as to obtain the access path and the join method used by the query statement as well as other information such as the number of rows retrieved from the database.

- ALTER SESSION SET SQL_TRACE = TRUE;
- ALTER SESSION SET SQL_TRACE = FALSE;

Database Creation

The tables were created and populated using SQL statements (Appendix B). The creation of unique random values for the join attribute was achieved through the use of a program written by Windy Weaver & Mike Raulin (1994). This program generates unique random numbers for a given range and outputs the random numbers to a text file (Refer Appendix G). Unix commands were executed to convert the text file to a format that could be read by the PL/SQL procedure (Refer Appendix D). After formatting, each line contained a single number instead of a string of numbers. The Oracle built-in package 'UTL_FILE' was used to read data from this file.

Optimiser hints

Three different experiments were set up to test the hypotheses. Each experiment will be described in the following section.

In order to force the optimiser to use a particular join method, hints were specified in the join query statement. In the Oracle environment, hints are specified after the SELECT statement. For example, the query below forces the optimiser to use a sort merge join method.

```
SELECT /*+ USE_MERGE(QUO, CUS) */
    cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND cus.postcode < 6101;
```

Selectivity Factor

The join selectivity factor is computed as follows:

Consider a join between a small and large relation.

The small relation consists of 100 tuples.

The large relation consists of 1000 tuples.

The result relation consists of 10 tuples.

The join selectivity factor is:

10/(100*1000) = 0.0001.

The selectivity of a table was calculated as follows:

The number of tuples in outer table is 100.

The number of tuples to be retrieved from database is 10.

The selectivity of outer table is 10/100 or 0.1.

The selectivity factor was varied by changing the condition value defined against the attribute. For example, consider the following query:

```
SELECT cus.cust_id, quo.quote_no
from quotes quo, customers cus
WHERE cus.cust_id = quo.quo_id
AND cus.postcode > n;
```

The value of 'n' was changed to vary the number of rows retrieved from the database.

Experiments

Three experiments were conducted to consider the performance of three different join methods under varying conditions:

1. Varying the join selectivity factor for a 1-to-10 relationship.
2. Varying the join selectivity factor for 1-to-1 relationship.
3. Applying a filter condition to the inner table for changing selectivity of the outer table on a one-to-many relationship.

Each experiment was run fifteen times for each join method. Each join method considered twelve different selectivities each requiring a unique query statement. The order of the run of the join methods was varied to ensure consistency. Before each run of the join method, the database was shutdown and restarted to ensure that the database buffer cache was cleared and that a join method did not use data present in the cache from the previous run.

The same outer table was used for these experiments. The outer table, in this case, the CUSTOMERS table contained 1000 rows and the inner table, the QUOTES table contained 10,000 rows.

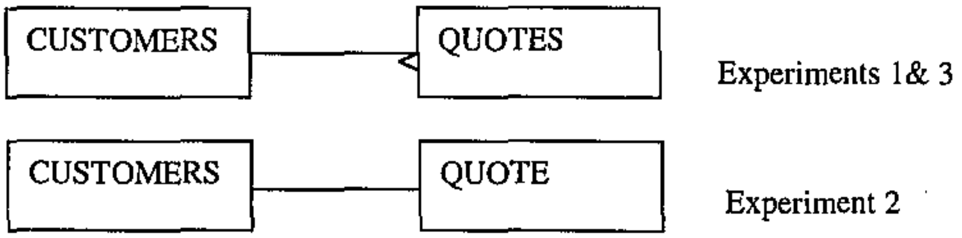


Figure 8: Entity-Relation Diagram showing the relation between the tables in the experiments

Set up of Experiment 1

The two relations were joined by a one-to-many relationship. Each tuple in the outer relation was related to ten tuples in the inner relation. The large (inner) relation was populated by adding a thousand tuples at a time until ten thousand tuples were added. The process of adding a thousand tuples at a time ensured that the foreign key value consisted of random values ranging from 1 to 1000. The random program generator program was run ten times to generate ten files consisting of unique random values ranging from 1 to 1000. This process was repeated ten times and a tuple from the outer table was always related to 10 tuples from the inner table.

The following query statement was executed:

General query statement	Actual query statement
SELECT TAB1.C2, TAB2.C1 FROM TAB1, TAB2 WHERE TAB1.C1 = TAB2.C4 AND TAB1.C4 < n;	SELECT cus.name, quo.quote_no FROM quotes quo, customers cus WHERE cus.cust_id = quo.cust_id AND cus.postcode < 6101;

Set up of Experiment 2

The join between the two relations in this experiment was a one-to-one relationship. A join attribute value from the outer relation could thus only exist once in the inner relation. The large (inner) relation was populated from the large relation used in experiment 1 with a null value set for the foreign key value (also the join attribute value). A thousand tuples were then selected at random from the large relation and their foreign key values were updated with a unique random value ranging from 1-1000. This process ensured that only 1000 tuples contained a join attribute value and that these values were from the domain defined for the primary column of the inner relation.

The same query statement as in experiment 1 was executed:

General query statement	Actual query statement
SELECT TAB1.C2, TAB2.C1 FROM TAB1, TAB2 WHERE TAB1.C1 = TAB2.C4 AND TAB1.C4 < n;	SELECT cus.name, quo.quote_no FROM quotes quo, customers cus WHERE cus.cust_id = quo.cust_id AND cus.postcode < 6101;

Set Up of Experiment 3

The inner relation was populated in such a way that half of the values contained in column3 had a value of 50000. The other half contained unique random numbers ranging from 1 to 10000. A filter condition was applied to the inner table so that for 50% of the tuples satisfied the condition when the outer table selectivity varied.

The following query statement was considered:

General query statement	Actual query statement
SELECT TAB1.C2, TAB2.C1 FROM TAB1, TAB2 WHERE TAB1.C1 = TAB2.C4 AND TAB1.C4 < n AND TAB2.C3< 50000;	SELECT cus.name, quo.quote_no FROM quotes quo, customers cus WHERE cus.cust_id = quo.cust_id AND cus.postcode < 6001 AND quo.amount < 1000001;

Data Conversion to SPSS Data File

For every run of the join method, a trace file was generated. The Oracle utility TKPROF was used to format the generated trace file (15 files per join method or 45 files per experiment) into a text file. The formatted file provided useful information such as the execution plan of the join query statement as well as statistics about the CPU time, the response time and the number of data blocks read. The execution plan provided details such as the access paths and join methods used.

A UNIX script (detailed in Appendix D) was then run against the formatted text files to extract the required data into a SPSS readable format. The extraction of data worked in two phases:

1. The formatted files were scanned one at a time for the lines containing the performance data and these lines were then written to separate text files.
2. These text files were scanned to extract selected fields (such as response time, CPU time and number of disk reads) and these fields were then stored in separate data files.

The data files were loaded directly into SPSS. This prevented unnecessary typing or data entry error.

Pilot Study

The experimental and recording procedures were tested in a pilot study. The pilot study considered the performance of the nested loop, sort merge and hash join methods for a small and a large relation and focused on:

- eleven distinct selectivity factors
- a 1-to-10 relationship.

The experiment was run 10 times for each type of join method.

Main Study

The main study measured the performance of the nested loop, sort merge and hash join methods for a small and a large relation and considered the following:

- twelve distinct selectivities for the outer table,
- twelve distinct join selectivity factors,
- a one-to-one relationship, one-to-many relationship, and
- a predicate applied to inner table for a one-to-many relationship.

Three set of experiments were run:

- Response time v/s join selectivity factor for a one-to-one relationship,
- Response time v/s join selectivity factor for a one-to-many relationship,
- Response time v/s outer table selectivity when a predicate was applied to the inner relation for a one-to-many relationship.

The CPU time, the response time and the number of I/O reads were measured. However, only the response time was required to test the hypotheses. The other data collected was used to graphically show the effect of the join method on the selectivity factor.

Data Analysis

The response time was classified as low, medium and high (See Table 5).

Table 5: Classification of response time

Response time classification	Percentage of number of tuples retrieved for a join condition to the total number of tuples retrieved if all tuples satisfies the condition	Join Selectivity Factor for a one-to-one relationship	Join Selectivity Factor for a one-to-many relationship
Low	0 - 10	<= 0.00001	<= 0.0001
Medium	11 -59	> 0.00001 and < 0.00006	> 0.0001 and < 0.0006
High	60 - 100	>= 0.00006	>= 0.0006

The hypotheses were initially tested using a t-test. A t-test is used for independent samples of sample size less than twenty and when the data is normally distributed. This research dealt with three independent samples each with a sample size of 15, that is, three experiments with 15 runs each. However, the test of normality showed that the data was not normally distributed for two cases (at low join selectivity factor for the nested loop join for a one-to-one relationship and at high join selectivity factor for the nested loop join for a one-to-many relationship). Therefore, the Mann-Whitney test was used instead of the t-test. The Mann-Whitney test is used for small sample size of less than 20 and when the data is not normally distributed.

The first and second experiments considered two independent measures: join method (nested loop, sort merge and hash join) and selectivity (low, medium and high) The dependent measure was the response time and was measured in seconds. The third experiment considered the effect of applying a filter condition on the inner table when the number of rows retrieved from the outer table varied. The independent variables were the selectivity (low, medium and high) of the outer table and the predicate on the inner table (with, without). The dependent factor was the response time.

Hypotheses

An alpha level of .05 was used for all statistical tests. The following hypotheses were tested:

H_1: For a one-to-many relationship with a low join selectivity factor, the nested loop has a faster response time than the sort merge join method.

X1 = response time for the nested loop

X2 = response time for the hash join

H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

The data collected in experiment 2 were used to test this hypothesis. A Mann-Whitney test was applied to the response time of the nested loop and hash join at low join selectivity factor. If the probability value obtained from the test was less than or equal to 0.05, then the null hypothesis was rejected.

H_2: For a one-to-many relationship with a low join selectivity factor, the hash join has a faster response time than the sort merge join method.

X1 = response time for the hash join

X2 = response time for the sort merge

H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

The data collected in experiment 2 was used to test this hypothesis. A Mann-Whitney test was applied to the response time of the hash join and sort merge at low join selectivity factor. If the probability value obtained from the test was less than or equal to 0.05, then the null hypothesis was rejected.

H_3: For a one-to-many relationship with a high join selectivity factor, the sort merge has a faster response time than the nested loop join method.

X1 = response time for the sort merge

X2 = response time for the nested loop

H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

The data collected in experiment 1 was used to test this hypothesis. A Mann-Whitney test was applied to the response time of the sort merge and nested loop at low join selectivity factor. If the probability value obtained from the test was less than or equal to 0.05, then the null hypothesis was rejected.

H_4: For a one-to-many relationship with a high join selectivity factor, the hash join has a faster response time than the sort merge join method.

X1 = response time for the hash join

X2 = response time for the sort merge

H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

The data collected in experiment 1 was used to test this hypothesis. A Mann-Whitney test was applied to the response time of the sort merge and nested loop at low join selectivity factor. If the probability value obtained from the test was less than or equal to 0.05, then the null hypothesis was rejected.

H_5: For a one-to-one relationship with a low join selectivity factor, the nested loop has a faster response time than the sort merge join method.

X1 = response time for the nested loop

X2 = response time for the sort merge

H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

The data collected in experiment 2 was used to test this hypothesis. A Mann-Whitney test was applied to the response time of the nested loop and sort merge at low join selectivity factor. If the probability value obtained from the test was less than or equal to 0.05, then the null hypothesis was rejected.

H_6: At high join selectivity factor, the nested loop join method has a faster response time for a one-to-one relationship than for a one-to-many relationship.

X1 = response time for the nested loop for a one-to-one relationship

X2 = response time for the nested loop for a one-to-many relationship

H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

The data collected in experiment 2 was used to test this hypothesis. A Mann-Whitney test was applied to the response time of the nested loop at low and high join selectivity factor. If the probability value obtained from the test was less than or equal to 0.05, then the null hypothesis was rejected.

H_7: The sort merge join method with low selectivity of the outer relation gives a faster response time when a predicate is applied to the inner relation than when no predicate is applied.

X1 = response time for the sort merge loop with a predicate on inner table at low selectivity.

X2 = response time for the sort merge with no predicate on inner table at low selectivity.

H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

The data collected in experiment 3 was used to test this hypothesis. A Mann-Whitney test was applied to the response time of the sort merge at low selectivity with and without a predicate on the inner table. If the probability value obtained from the test was less than or equal to 0.05, then the null hypothesis was rejected.

H_8: The sort merge join method with high selectivity of the outer relation gives a faster response time when a predicate is applied to the inner relation than when no predicate is applied.

X1 = response time for the sort merge loop with a predicate on inner table at high selectivity.

X2 = response time for the sort merge with no predicate on inner table at high selectivity.

H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

The data collected in experiment 3 was used to test this hypothesis. A Mann-Whitney test was applied to the response time of the sort merge at low selectivity with and without a predicate on the inner table. If the probability value obtained from the test was less than or equal to 0.05, then the null hypothesis was rejected.

Limitations

This research has some limitations:

- Whenever an Oracle instance is started, a number of processes are also started. These processes communicate with each other via the shared memory known as the Shared Global Area (SGA). The SGA consists of the shared pool, the data block buffer cache and the redo log buffer.

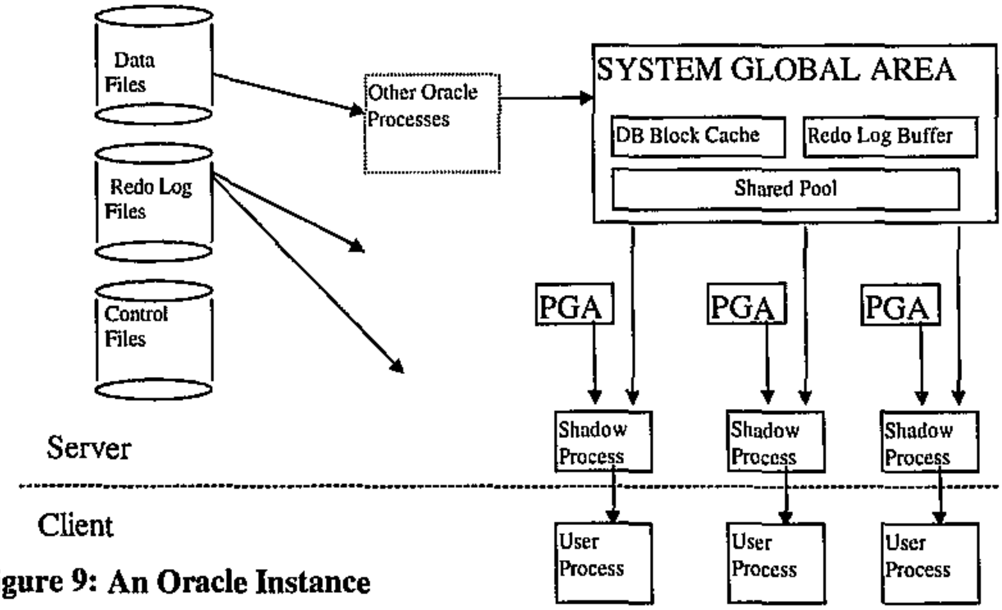


Figure 9: An Oracle Instance

The shared pool contains parsed SQL statements. Whenever a SQL statement is executed, the statement is parsed and stored in the shared pool. Before a SQL statement is parsed, the shared pool is first checked to see if the parsed statement already exists. If the parsed statement is found, then the cost of executing that statement will be reduced. It is therefore necessary to ensure that the shared pool is empty before each run of the experiment so that the elapsed time better reflect the time taken to parse the statement. When the shared pool becomes full, objects are removed from the pool on a least recently used (LRU) basis (Urman, 1996, p. 476). Additions and deletions of objects cause the shared pool area to become fragmented. Consequently, to prevent fragmentation and to ensure a clean environment for every run, the shared pool need to be refreshed. The following command was executed before each run of the experiment:

```
ALTER SYSTEM FLUSH SHARED_POOL.
```

- The database block (DB) buffer cache in the SGA stores copies of the database blocks. Blocks are loaded in the DB buffer cache when a process reads data from the database The database buffer processes data that in a LRU fashion. To ensure that the buffer cache is empty for every run, the database was shutdown and restarted after each run.

- The execution of the experiments could have been automated in such a way that a batch job executed all the runs for the different join methods. However, since the NT operating system provides for parallel processing and therefore allocates processing time to each processes, the experiments would not have reflected the relevant time. A single run of the experiment was executed at a time in order to ensure an unbiased treatment of the runs.

- The sort merge method first sorts both tables on the join columns and then performs a merge using the join column. If the columns are already sorted, then the time taken to process a join using the sort merge method will be reduced. Consequently, to ensure an unbiased treatment of the join method, the join column consisted of random generated values.

- It was found that the NT operating system crashed when the TKPROF utility was run against the generated trace files. After investigation of this unexpected behaviour, it was found that TKPROF did not support the word 'APPNAME' found in the trace files. The problem was fixed by removing that word from the generated trace files.

- Under Windows 95 environment, the generated trace files did not record the CPU time. Consequently, Windows NT environment was used.

- Random values were generated in a text file using the random generator program. During creation of the tables, this text file was read from a SQL procedure using a built-in Oracle package. However, it was found that this package could not be used under Windows NT version 4.0 but could be successfully used under Windows NT version 3.5. Therefore, the creation of the database was done under NT 3.5 and the database was later exported to NT 4.0.

- To ensure a fair treatment of the join methods, the order of the runs for the join methods was varied.

Summary

It was found that the design of this experiment was a lengthy activity as there were several essential conditions to be satisfied before setting up the database. The limitations of this research also added to the complexity of the set up. The solving of the problems encountered with the software and hardware consumed a considerable amount of time.

Chapter Four: Results

Hypothesis 1 - Nested Loop v/s Sort Merge at low join selectivity for 1-to-10

H_1: For a one-to-many relationship with a low join selectivity factor, the nested loop has a faster response time than the sort merge join method.

X1 = response time for the nested loop

X2 = response time for the sort merge

H_0: $\mu_1=\mu_2$

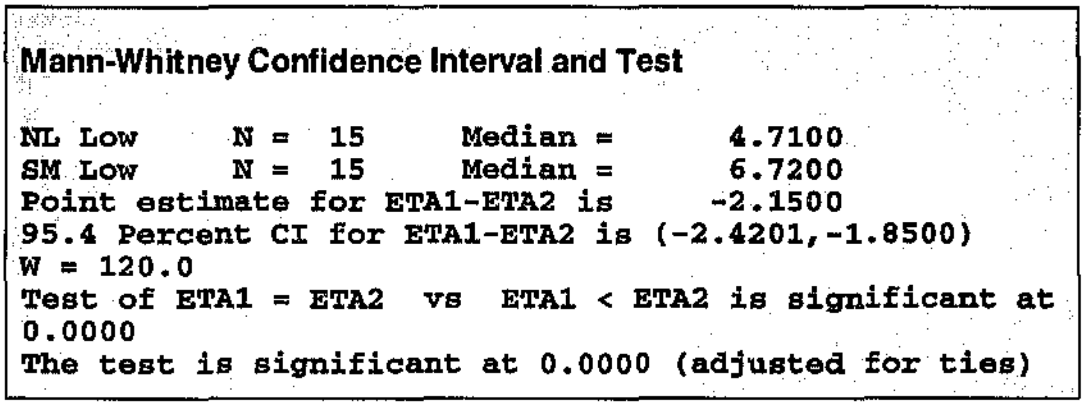
H_A: $\mu_1<\mu_2$

Table 6: Response times for the nested loop and sort merge at low join selectivity factor for a one-to-many relationship

Runs	NL Low	SM Low
1	4.59	6.70
2	4.82	6.47
3	4.58	6.71
4	4.87	6.16
5	4.85	6.57
6	4.56	7.07
7	4.76	7.03
8	4.63	6.44
9	4.76	6.05
10	4.71	7.11
11	4.21	7.20
12	4.62	7.45
13	4.87	6.72
14	4.77	7.13
15	4.65	7.20

NL Low – Response time of Nested Loop at low join selectivity factor
SM Low - Response time of Sort Merge at low join selectivity factor

Table 6 shows the data used to test the above hypothesis. The Mann-Whitney test was applied to the data



```
Mann-Whitney Confidence Interval and Test

NL Low     N =  15     Median =       4.7100
SM Low     N =  15     Median =       6.7200
Point estimate for ETA1-ETA2 is      -2.1500
95.4 Percent CI for ETA1-ETA2 is (-2.4201,-1.8500)
W = 120.0
Test of ETA1 = ETA2  vs  ETA1 < ETA2 is significant at
0.0000
The test is significant at 0.0000 (adjusted for ties)
```

Figure 10: Minitab output showing the test for Hypothesis 1 using the Mann-Whitney test

The figure above shows that, at an alpha level of 0.05, the response time of the nested loop is significantly less than the response time of the sort merge at low join selectivity factor for a one-to-many relationship. Since the probability value 0.0000 is less than 0.05, therefore the null hypothesis is rejected. As a result, the nested loop performs better than the sort merge at low join selectivity factor for a one-to-many relationship.

Hypothesis 2 – Hash Join v/s Sort Merge at low join selectivity for 1-to-10

H_2: For a one-to-many relationship with a low join selectivity factor, the hash join has a faster response time than the sort merge join method.

X1 = response time for the hash join

X2 = response time for the sort merge

H_0: $\mu_1=\mu_2$

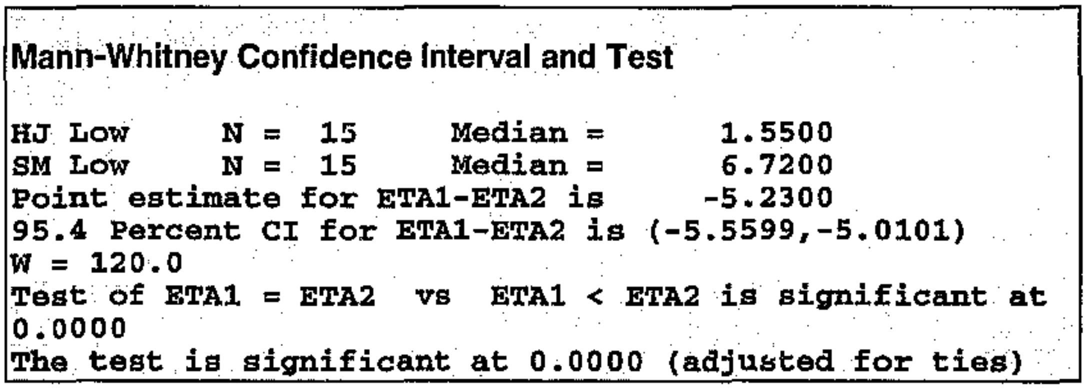
H_A: $\mu_1<\mu_2$

Table 7: Response times for the sort merge and hash at low join selectivity factor for a one-to-many relationship

Runs	SM Low	HJ Low
1	6.70	4.96
2	6.47	4.98
3	6.71	4.74
4	6.16	4.78
5	6.57	4.71
6	7.07	4.73
7	7.03	4.91
8	6.44	4.84
9	6.05	4.87
10	7.11	5.00
11	7.20	4.88
12	7.45	4.97
13	6.72	4.94
14	7.13	5.13
15	7.20	4.75

SM Low - Response time of Sort Merge
at low join selectivity factor
HJ Low –Response time of Hash Join
at low join selectivity factor

Table 7 shows the data used to test the above hypothesis. The Mann-Whitney test was applied to the data.



```
Mann-Whitney Confidence Interval and Test

HJ Low      N =  15      Median =       1.5500
SM Low      N =  15      Median =       6.7200
Point estimate for ETA1-ETA2 is     -5.2300
95.4 Percent CI for ETA1-ETA2 is (-5.5599,-5.0101)
W = 120.0
Test of ETA1 = ETA2  vs  ETA1 < ETA2 is significant at
0.0000
The test is significant at 0.0000 (adjusted for ties)
```

Figure 11: Minitab output showing the test for Hypothesis 2 using the Mann-Whitney test

The figure above shows that, at an alpha level of 0.05, the response time of the hash join is significantly less than the response time of the sort merge at low join selectivity factor for a one-to-many relationship. Since the probability value 0.0000 is less than 0.05, therefore the null hypothesis is rejected. As a result, the hash join performs better than the sort merge at low join selectivity factor for a one-to-many relationship.

Hypothesis 3 - Nested Loop v/s Sort Merge at high join selectivity for 1-to-10

H_3: For a one-to-many relationship with a high join selectivity factor, the sort merge has a faster response time than the nested loop join method.

X1 = response time for the sort merge

X2 = response time for the nested loop

H_0: $\mu_1=\mu_2$

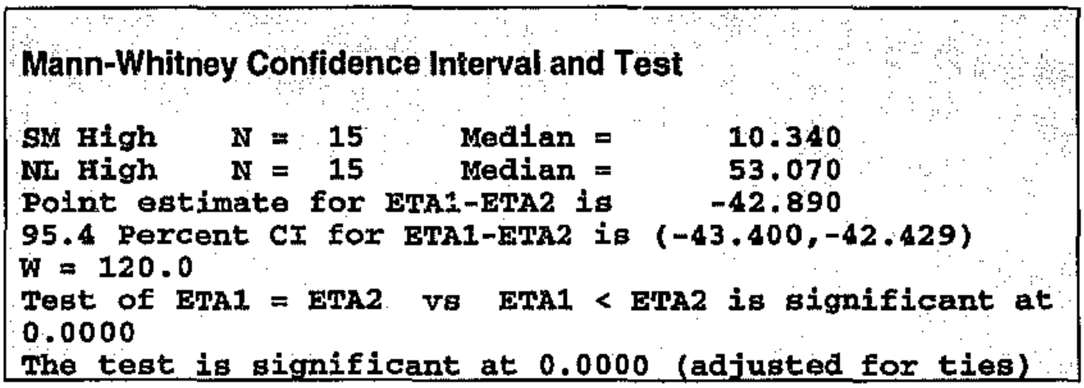
H_A: $\mu_1<\mu_2$

Table 8: Response times for the sort merge and nested loop at high join selectivity for a one-to-many relationship

Runs	SM High	NL High
1	10.52	53.07
2	10.64	52.85
3	10.74	52.58
4	9.26	53.12
5	9.29	59.37
6	10.03	52.66
7	9.79	52.22
8	10.34	52.79
9	9.91	53.54
10	10.47	53.63
11	10.12	52.77
12	10.79	52.60
13	10.49	53.52
14	10.21	57.59
15	10.53	53.54

SM High - Response time of Sort Merge at high join selectivity factor
NL High –Response time of Nested Loop at high join selectivity factor

Table 8shows the data used to test the above hypothesis. The Mann-Whitney test was applied to the data.



```
Mann-Whitney Confidence Interval and Test

SM High    N =  15     Median =       10.340
NL High    N =  15     Median =       53.070
Point estimate for ETA1-ETA2 is      -42.890
95.4 Percent CI for ETA1-ETA2 is (-43.400,-42.429)
W = 120.0
Test of ETA1 = ETA2  vs  ETA1 < ETA2 is significant at
0.0000
The test is significant at 0.0000 (adjusted for ties)
```

Figure 12: Minitab output showing the test for Hypothesis 3 using the Mann-Whitney test

The figure above shows that, at an alpha level of 0.05, the response time of the sort merge is significantly less than the response time of the nested loop at high join selectivity factor for a one-to-many relationship. Since the probability value 0.0000 is less than 0.05, therefore the null hypothesis is rejected. As a result, the sort merge performs better than the nested loop at high join selectivity factor for a one-to-many relationship.

Hypothesis 4 – Hash Join v/s Sort Merge at high join selectivity for 1-to-10

H_4: For a one-to-many relationship with a high join selectivity factor, the hash join has a faster response time than the sort merge join method.

X1 = response time for the hash join

X2 = response time for the sort merge

H_0: $\mu_1=\mu_2$

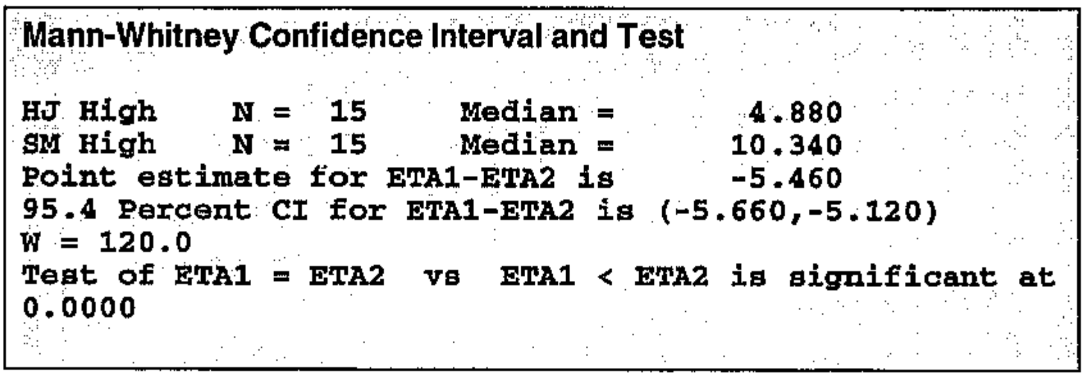
H_A: $\mu_1<\mu_2$

Table 9: Response times for the hash join and sort merge at high join selectivity for a one-to-many relationship

Runs	HJ High	SM High
1	4.96	10.52
2	4.98	10.64
3	4.74	10.74
4	4.78	9.26
5	4.71	9.29
6	4.73	10.03
7	4.91	9.79
8	4.84	10.34
9	4.87	9.91
10	5.00	10.47
11	4.88	10.12
12	4.97	10.79
13	4.94	10.49
14	5.13	10.21
15	4.75	10.53

HJ High - Response time of Hash Join
at high join selectivity factor
SM High –Response time of Sort Merge
at high join selectivity factor

Table 9 shows the data used to test the above hypothesis. The Mann-Whitney test was applied to the data.



```
Mann-Whitney Confidence Interval and Test

HJ High     N =  15     Median =        4.880
SM High     N =  15     Median =       10.340
Point estimate for ETA1-ETA2 is        -5.460
95.4 Percent CI for ETA1-ETA2 is (-5.660,-5.120)
W = 120.0
Test of ETA1 = ETA2  vs  ETA1 < ETA2 is significant at
0.0000
```

Figure 13: Minitab output showing the test for Hypothesis 4 using the Mann-Whitney test

The figure above shows that, at an alpha level of 0.05, the response time of the hash join is significantly less than the response time of the sort merge at high join selectivity factor for a one-to-many relationship. Since the probability value 0.0000 is less than 0.05, therefore the null hypothesis is rejected. As a result, the hash join performs better than the sort merge at high join selectivity factor for a one-to-many relationship.

Hypothesis 5 - Nested Loop v/s Sort Merge at low join selectivity for 1-1

H_5: For a one-to-one relationship with a low join selectivity factor, the nested loop has a faster response time than the sort merge join method.

X1 = response time for the nested loop

X2 = response time for the sort merge

H_0: $\mu_1=\mu_2$

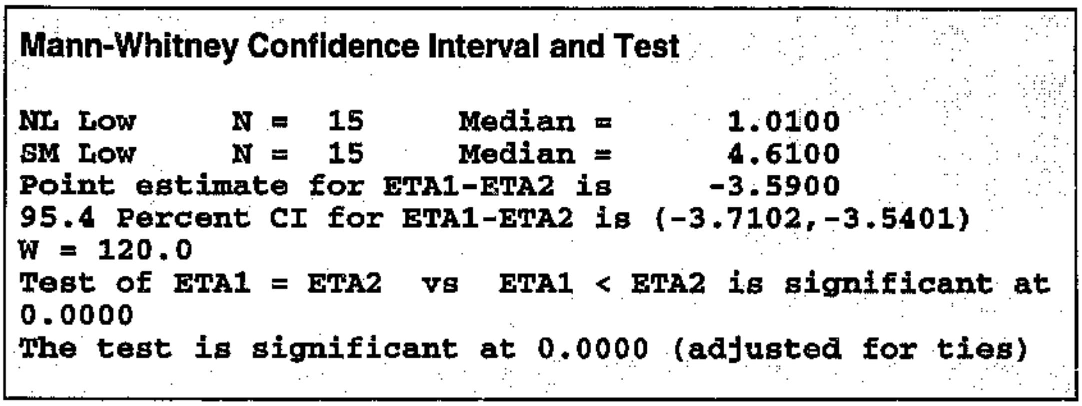
H_A: $\mu_1<\mu_2$

Table 10: Response times for the nested loop and sort merge at low join selectivity for a one-to-one relationship

Runs	NL Low	SM Low
1	0.92	4.53
2	1.02	4.74
3	1.05	4.95
4	0.81	5.45
5	1.01	4.46
6	1.01	4.98
7	0.83	4.96
8	0.96	5.25
9	1.01	5.04
10	1.03	5.08
11	1.05	5.00
12	0.93	5.38
13	1.04	4.97
14	1.06	5.20
15	1.04	4.91

NL Low - Response time of Nested Loop at low join selectivity factor
SM Low –Response time of Sort Merge at low join selectivity factor

Table 10 shows the data used to test the above hypothesis. The Mann-Whitney test was applied to the data.



```
Mann-Whitney Confidence Interval and Test

NL Low     N =  15     Median =        1.0100
SM Low     N =  15     Median =        4.6100
Point estimate for ETA1-ETA2 is       -3.5900
95.4 Percent CI for ETA1-ETA2 is (-3.7102,-3.5401)
W = 120.0
Test of ETA1 = ETA2  vs  ETA1 < ETA2 is significant at
0.0000
The test is significant at 0.0000 (adjusted for ties)
```

Figure 14: Minitab output showing the test for Hypothesis 5 using the Mann-Whitney test

The figure above shows that, at an alpha level of 0.05, the response time of the nested loop join is significantly less than the response time of the sort merge at low join selectivity factor for a one-to-one relationship. Since the probability value 0.0000 is less than 0.05, therefore the null hypothesis is rejected. As a result, the nested loop join performs better than the sort merge at low join selectivity factor for a one-to-one relationship.

Hypothesis 6 – At high join selectivity, 1-to-1 v/s 1-to-10 for Nested Loop

H_6: At high join selectivity factor, the nested loop join method has a faster response time for a one-to-one relationship than for a one-to-many relationship.

X1 = response time for the nested loop for a one-to-one relationship

X2 = response time for the nested loop for a one-to-many relationship.

H_0: $\mu_1=\mu_2$

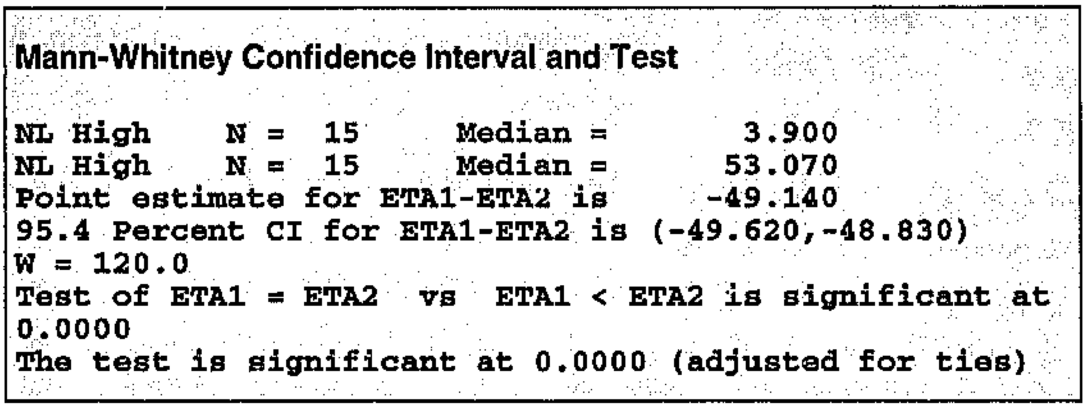
H_A: $\mu_1<\mu_2$

Table 11: Response times for the nested loop and sort merge at high join selectivity for a one-to-one relationship

Runs	NL High (1-1)	NL High (1-10)
1	3.82	53.07
2	3.98	52.85
3	3.93	52.58
4	4.16	53.12
5	3.81	59.37
6	3.75	52.66
7	3.63	52.22
8	3.90	52.79
9	3.77	53.54
10	3.84	53.63
11	4.01	52.77
12	4.02	52.60
13	3.95	53.52
14	4.15	57.59
15	3.87	53.54

NL High (1-1) - Response time of Nested Loop at high JSF for a one-to-one relationship
NL High (1-10) - Response time of Nested Loop at high JSF for a one-to-many relationship

Table 11 shows the data used to test the above hypothesis. The Mann-Whitney test was applied to the data.



```
Mann-Whitney Confidence Interval and Test

NL High    N =  15     Median =        3.900
NL High    N =  15     Median =       53.070
Point estimate for ETA1-ETA2 is      -49.140
95.4 Percent CI for ETA1-ETA2 is (-49.620,-48.830)
W = 120.0
Test of ETA1 = ETA2  vs  ETA1 < ETA2 is significant at
0.0000
The test is significant at 0.0000 (adjusted for ties)
```

Figure 15: Minitab output showing the test for Hypothesis 6 using the Mann-Whitney test

The figure above shows that, at an alpha level of 0.05, the response time of the nested loop join for a one-to-one relationship is significantly less than the response time of the nested loop for a one-to-many relationship at high join selectivity factor. Since the probability value 0.0000 is less than 0.05, therefore the null hypothesis is rejected. As a result, the nested loop join for a one-to-one relationship performs better than the nested loop for a one-to-many relationship at high join selectivity factor.

Hypothesis 7 – At low join selectivity, SM with predicate v/s SM no predicate

H_7: The sort merge join method with low selectivity of the outer relation gives a faster response time when a predicate is applied to the inner relation than when no predicate is applied.

X1 = response time for the sort merge with a predicate applied to the inner relation

X2 = response time for the sort merge with no predicate applied to the inner relation

H_0: $\mu_1=\mu_2$

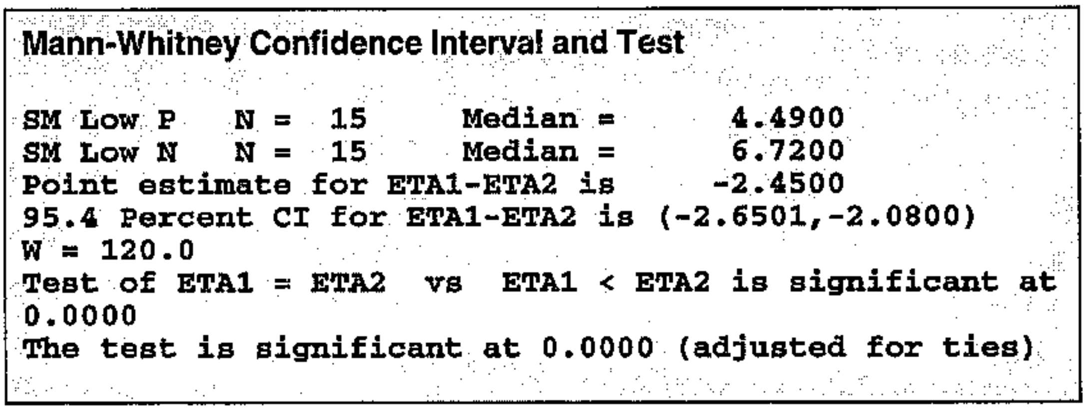
H_A: $\mu_1<\mu_2$

Table 12: Response times for the sort merge at low join selectivity for a one-to-many relationship with and without a predicate applied to the inner table

Runs	SM Low Pred	SM Low No Pred
1	3.79	6.70
2	3.98	6.47
3	4.21	6.71
4	4.41	6.16
5	4.12	6.57
6	4.49	7.07
7	4.54	7.03
8	4.49	6.44
9	4.61	6.05
10	4.60	7.11
11	4.49	7.20
12	4.75	7.45
13	4.62	6.72
14	4.70	7.13
15	4.46	7.20

SM Low Pred - Response time of Sort Merge at low JSF with a predicate on inner relation
SM Low No Pred - Response time of Sort Merge at low JSF with no predicate on inner relation

Table 12 shows the data used to test the above hypothesis. The Mann-Whitney test was applied to the data.



Mann-Whitney Confidence Interval and Test

```
SM Low P   N =  15      Median =        4.4900
SM Low N   N =  15      Median =        6.7200
Point estimate for ETA1-ETA2 is       -2.4500
95.4 Percent CI for ETA1-ETA2 is (-2.6501,-2.0800)
W = 120.0
Test of ETA1 = ETA2  vs  ETA1 < ETA2 is significant at
0.0000
The test is significant at 0.0000 (adjusted for ties)
```

Figure 16: Minitab output showing the test for Hypothesis 7 using the Mann-Whitney test

The figure above shows that, at an alpha level of 0.05, the response time of the sort merge with a predicate on the inner table is significantly less than the response time of the of the sort merge with no predicate on the inner table at low join selectivity factor. Since the probability value 0.0000 is less than 0.05, therefore the null hypothesis is rejected. As a result, the sort merge with a predicate on the inner performs better than the response time of the of the sort merge with no predicate on the inner table at low join selectivity factor.

Hypothesis 8 - At high join selectivity, SM with predicate v/s SM no predicate

H_8: The sort merge join method with high selectivity of the outer relation gives a faster response time when a predicate is applied to the inner relation than when no predicate is applied.

X1 = response time for the sort merge with a predicate applied to the inner relation

X2 = response time for the sort merge with no predicate applied to the inner relation

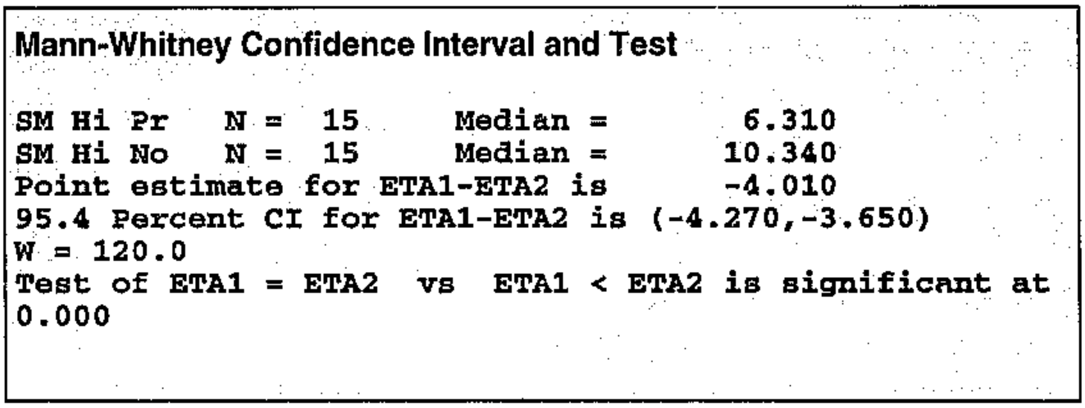
H_0: $\mu_1=\mu_2$

H_A: $\mu_1<\mu_2$

Table 13: Response times for the sort merge at high join selectivity for a one-to-many relationship with and without a predicate applied to the inner table

Runs	SM Hi Pred	SM Hi No Pred	
1	5.75	10.52	
2	5.61	10.64	
3	6.22	10.74	
4	6.18	9.26	
5	5.87	9.29	
6	6.58	10.03	
7	6.29	9.79	
8	6.15	10.34	
9	6.42	9.91	
10	6.51	10.47	
11	6.44	10.12	
12	6.59	10.79	SM Hi Pred - Response time of Sort Merge
13	6.73	10.49	at high JSF with a predicate on inner relation
14	6.43	10.21	SM Hi No Pred - Response time of Sort Merge
15	6.31	10.53	at high JSF with no predicate on inner relation

Table 13 shows the data used to test the above hypothesis. The Mann-Whitney test was applied to the data.



```
Mann-Whitney Confidence Interval and Test

SM Hi Pr   N =  15     Median =        6.310
SM Hi No   N =  15     Median =       10.340
Point estimate for ETA1-ETA2 is       -4.010
95.4 Percent CI for ETA1-ETA2 is (-4.270,-3.650)
W = 120.0
Test of ETA1 = ETA2  vs  ETA1 < ETA2 is significant at
0.000
```

Figure 17: Minitab output showing the test for Hypothesis 8 using the Mann-Whitney test

The figure above shows that at an alpha level of 0.05, the response time of the sort merge with a predicate on the inner table is significantly less than the response time of the of the sort merge with no predicate on the inner table at high join selectivity factor. Since the probability value 0.0000 is less than 0.05, therefore the null hypothesis is rejected. As a result, the sort merge with a predicate on the inner performs better than the response time of the of the sort merge with no predicate on the inner table at high join selectivity factor.

Chapter Five: Discussion

The sensitivity with respect to varying selectivity of the response time, CPU time and number of logical reads were studied for the following join methods: nested loop, sort merge and hash join. The effect of applying a filter condition on the inner table on the response time was also considered.

Initial Observations

The join selectivity factor refers to the ratio of the number of tuples that satisfy a join to the total number of tuples present in a Cartesian product of the relations. A high selectivity factor means that a large proportion of possible tuples from the Cartesian product satisfies the join condition. A low join selectivity factor means that a small proportion of tuples satisfies the join condition.

Testing of hypotheses H1 and H5 showed that at low join selectivity factor, the nested loop performed better than the sort merge join method for both a one-to-one and a one-to-many relationship. Figure 19 and Figure 20 (see page 72) show the response time measured for the join methods for varying join selectivity factors. It can be observed from these figures that the hash join performs better than the sort merge and nested loop at varying join selectivity factor. The testing of hypothesis H2 and H4 lead to the conclusion that the hash join had a better response time than the sort merge for a one-to-many relationship.

Testing Hypotheses H3 at a high join selectivity factor, the sort merge performed better than the nested loop for a 1-to-10 relationship. The nested loop algorithm would need to read 10 tuples from the inner relation for each tuple read from the outer relation for a 1-to-10 relationship. The sort merge would read only tuples that it can be joined to. Mishra and Eich (1992, p. 74) noted that "a tuple from the outer relation is not compared with those tuples in the second relation with which it cannot possibly join" for a sort merge join. This explains why the sort merge has a better response time than the sort merge at high join selectivity factor foı a one-to-many relationship.

It was also observed that the cost of the sort merge was not impacted by the degree of relationship. It can be seen from Figure 25 and Figure 26 (see page 75) that the time taken to perform a sort merge for a one-to-one relationship and a one-to-many relationship for a small and large relation is the same. Since the size of the small and large relations used in both relationships are the same, then the same amount of I/O and processing is required to sort and merge the relations.

Testing hypothesis H6 showed that the nested loop performed better for a one-to-one relationship than a one-to-many relationship. This is because a tuple from the outer relation need to access only one tuple from the inner relation instead of 10 tuples.

Testing hypotheses H7 and H8 showed that the sort merge performed much better when a predicate was applied to the inner table. Since only the tuples that satisfy the join condition are sorted and merged, if less tuples are to be sorted, hence merged, then the elapsed time is reduced.

Detailed Observations

It can be noted from Figure 19 (page 72) that the nested loop shows an approximately linear increase in elapsed time when the number of tuples retrieved from the database increases in contrast to the sort merge and hash join method.

The linear increase in response time for varying join selectivity for the nested loop join is expected because of the way that the algorithm is implemented (please refer to Algorithm on page 19). For each tuple of the outer table that satisfies the join condition, all tuples from the inner relation are read via an index. If more tuples from the outer relation satisfy the join condition, then more tuples from the inner relation are accessed. For example, consider the join between the two relations used in the one-to-many experiments. In this case, each tuple from the outer table is related to 10 tuples of the inner relation. Therefore, if x tuples satisfy the join condition for the outer relation, then 10x tuples from the inner relation will be read. This results in a linear increase in elapsed time.

Figure 23 and Figure 24 (page 74) show that the number of I/O accesses for the nested loop join method increases as the join selectivity factor increases. For every qualifying tuple of the outer relation, a search for a matching value is done though each level of index. For every match found, an I/O read is required.

In the experiments using a one-to-many relationship, an index was defined on the join column 'cust_id' for the large relation QUOTES. This index was a non-unique index, as there existed more than one tuple with that same 'cust_id'. Therefore, the number of reads through the B-tree indexes increases for non-unique indexes. The figure below (adapted from Aronoff et al., 1997) illustrates the workings of the indexed access. The root node is initially read. Then the leaves are accessed. If the value of the index matches the required value, then the QUOTES table is subsequently read. The figure below shows that if three tuples from the inner relation satisfy the join condition, then 8 logical reads are required.

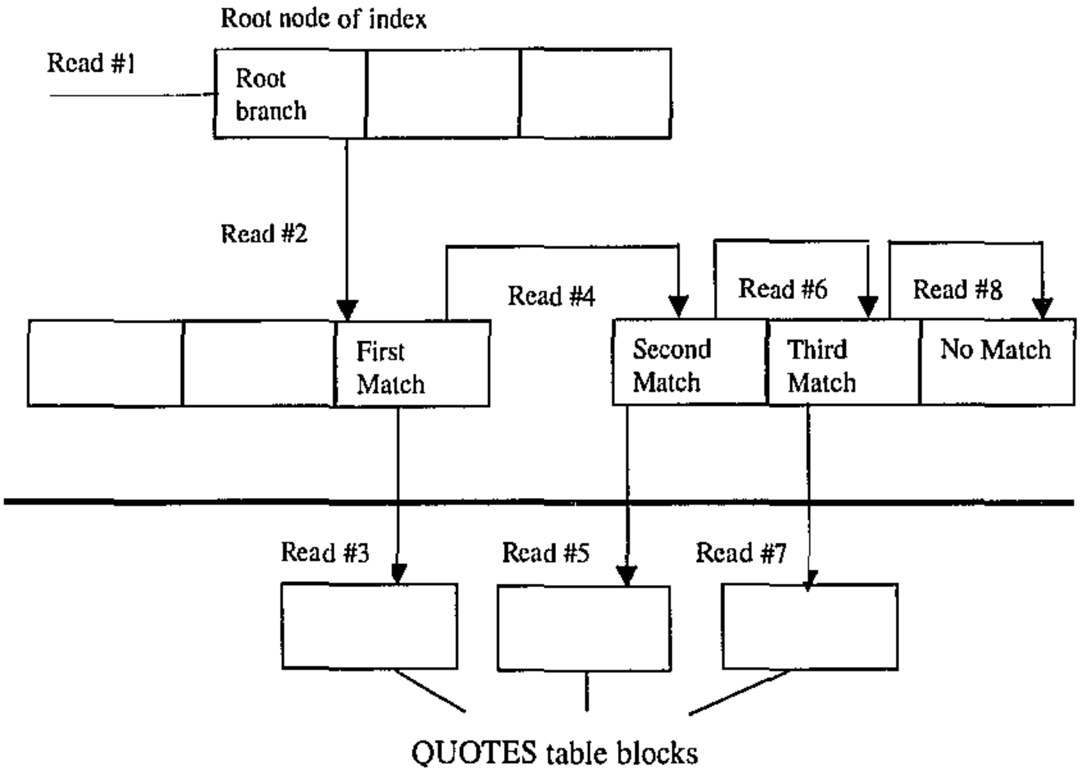


Figure 18: Reads for Indexed access

The experiments showed that the hash join performed well under the different conditions (See Figure 19 and Figure 20 on page 72). The performance of the hash join method depends on whether the smaller relation can fit into main memory. Harris (1995) noted that the hash join algorithm only required a single scan of the input relations if one of the two relations can be completely contained in memory.

An in-memory hash table of the join column value in the small relation is first built and if the relation is small enough to fit in memory, then the hash join competes well with the other join methods. The experiments considered a small relation that can be contained in memory. Therefore a single scan on the small relation is required.

Yu and Cornell (1991) mentioned that the CPU time for the sort merge was dependent on the size of the larger relation whereas the CPU time for the hash join was dependent on the size of the smaller relation. As mentioned in the literature review, the performance of the sort merge join is dependent on the number of passes required to merge the relations. A larger relation occupies more pages and therefore more passes may be required.

Figure 23 and Figure 24 (see page 74) show that the hash join and the sort merge have a similar profile with respect to varying join selectivity factor. Graefe et al. (1994) noted that the hash join and the sort merge have some similarities in the way that a data set is processed. Both the hash join and the sort merge makes use of an in-memory algorithm to process the data set. The sort merge performs an internal sort of the data (implemented by the quick sort or tournament tree algorithm) while the hash join employs a hashing technique.

> In the sort-based algorithms, a large data set is divided into subsets using a physical rule, namely, into chunks as large as memory. These chunks are later combined using a logical step, merging. In the hash-based algorithms, the large data set is cut into subsets using a logical rule, by hash values. The resulting

> partitions are later combined using a physical step, simply concatenating the subsets or result subsets. Graefe et al. (1994, p. 936)

In both join methods, the amount of memory determines the effectiveness of the merge or the hash. An increase in memory means that larger units of I/O can be allocated and therefore less paging and swapping occur. However, large pages cause internal fragmentation and therefore can impact on the processor.

The number of I/O reads depends on the number of different pages accessed during a join. Figure 23 and Figure 24 (see page 74) show the number of I/O reads for the join methods for varying join selectivity factors. It was noted that the sort merge has the same number of I/O accesses for varying join selectivity factor. This is because the sort merge makes use of "sequential access by prefetching multiple data pages, amortizing disk seek and latency overhead over multiple page transfers" (Cheng et al, 1991, p. 171). On the other hand, the nested loop requires more I/O than the other join methods because there are additional I/Os caused by the retrieval of index pages.

The linear increase in CPU time for the nested loop join method as shown in Figure 21 (on page 73) is due to an increase in paging and swapping. Paging occurs when data is being moved from disk to main memory and swapping occurs when data is being moved from memory to disk so as to release memory. Therefore, the CPU is busy moving data to and fro instead of processing requests. The page replacement strategy used by the data block buffer cache is the Least Recently Used (LRU) algorithm. This means that the page that has been unused for the longest time is replaced (Deitel, 1990). This has particular significance for the sort merge algorithm. The number of way merges used by the sort merge algorithm determines the efficiency of the sort merge. If the sort algorithm provides for a 16-way merge, then it means that 16 pages are loaded into 16 buffer frames. If more memory is required, then the first page loaded will be removed from memory. If the buffer size is small, there will be unnecessary page faults since the first page loaded will be swapped out and will then need to be accessed immediately in the next phase (Meechan, 1988). Also more passes will be required.

Predicate v/s No Predicate on the Inner Table

It can be observed from Figure 26 (page 75) that the sort merge join benefits the most from applying a predicate on the inner table. The result of applying a predicate on the inner table in the third experiment reduces the number of tuples eligible to satisfy the join condition by 50%. Because the number of tuples from the inner relation to be processed is halved, therefore the number of pages to be sorted and merged is also halved. Therefore, the CPU time as well as the number of I/O reads required to perform the sort merge with a predicate on the inner table is also reduced. The nested loop join and the hash join method are not affected by a predicate on the inner table as the same number of tuples from the inner table are read (See Table 20 and Table 23 in Appendix G on pages 153 and 154).

Critique

Mishra and Eich (1992) stated that the "nested-loops method is considered the most inefficient method to use in the case of low join selectivities" (p. 101). Mishra and Eich (1992) argued that the nested loop is inefficient "because most of the comparisons do not result in a match, and the effort is wasted" (p 101). Alternatively, it could be considered that because there is no match, then there is no need to access the block. This would imply that the number of logical reads required for the nested loop would be reduced and consequently the response time would be reduced This line of thought would therefore lead to the conclusion that the nested loop is an efficient method at low join selectivity factor.

The current research concluded that the nested loop has a faster response time than the sort merge join for both a one-to-many and a one-to-one relationship at low join selectivity factor (See hypotheses H1 and H5 on pages 48 and 54). The results therefore contradict the view of Mishra and Eich (1992) but agrees with the results obtained by Meechan (1988) and Cheng at al. (1991). They both concluded that the nested loop is better than the sort merge at low level of selectivity. The sort merge algorithm requires all tuples of the outer table to be accessed. Effort is therefore wasted in that case, as unqualified tuples for the join would still be accessed. The sort merge join method is, consequently, the worst join method to be used at low join selectivity factor.

It has further been affirmed that:

> The advantage that hash joins have over the nested-loops method diminishes as the selectivity factor increases. In this case, exhaustive comparison is useful because of the large number of tuples participating in the join. Furthermore, the nested-loops method does not have the overhead of doing hashing (Mishra

and Eich, 1992, p.101).

However, in the current experiment, hypotheses H3 and H4 led to the conclusion that at high selectivity factor, the hash join has a faster time than the sort merge and that the sort merge has a faster response time than the nested loop for a one-to-many relationship. Furthermore, Figure 19 and Figure 20 (see page 72) show that the hash join performs better than the sort merge and nested loop join methods as the join selectivity factor increases. This means that the performance of the hash join method is better than the nested loop join method at high selectivity factor and this again contradicts the writings of the above authors.

As the join selectivity factor increases, more tuples are qualified for the join. In the case of the nested loop join method, for each tuple from the outer relation that satisfies the join condition, all tuples from the inner relation are read Consequently, as the selectivity factor increases, more tuples need to be accessed and therefore more time is spent reading the blocks. Alternatively, the hash join method performs in-memory processing using the hash table to probe for matches. This type of processing is fast since the amount of input/output accesses is reduced considerably.

The result of the current research confirmed the results of Cheng et al's (1991) study and showed that the nested loop has a higher response time than the hybrid hash join as the selectivity increases.

Summary

The performance of the different join methods have been compared with respect to the total elapsed time, CPU time and number of I/O reads required to execute different query statements retrieving different number of tuples. The results obtained have been discussed with regards to the way that the different join algorithm was implemented. It has been found that the views raised by some authors are in contradiction with the results of this experiment as far as hypothesis H1, H3, H4, H5 were concerned. The current experiment agrees with Cheng et al's experiments (1991) and contradicts the views shared by Mishra and Eich (1992).

The figures have been placed at the end of this chapter as they are cross-referenced several times throughout this chapter.

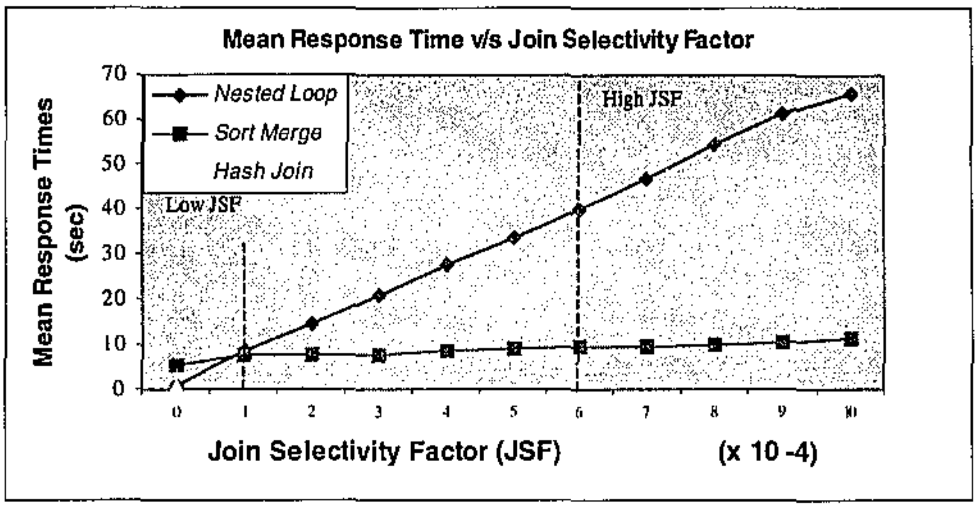


Figure 19: Effect of Join Selectivity on Response Time for a 1-to-10 relationship

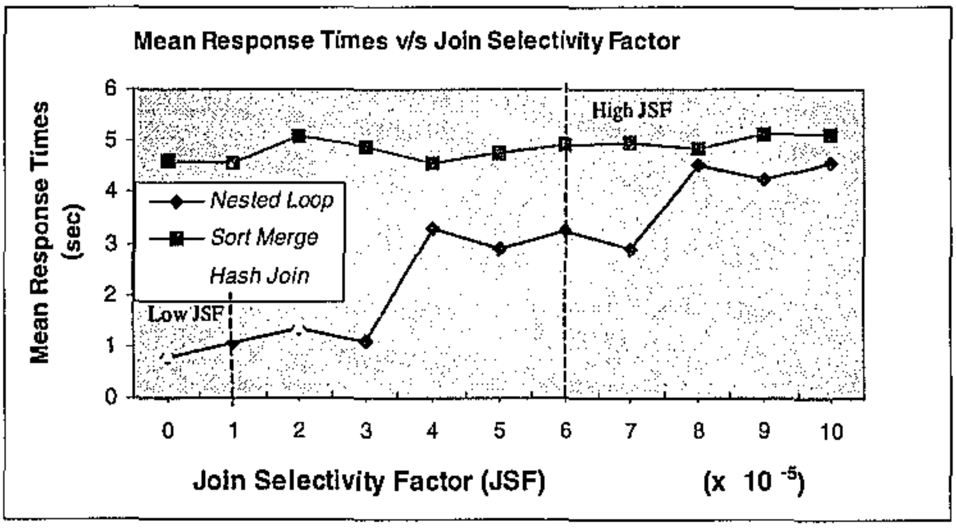


Figure 20: Effect of Join Selectivity on Response Time for a 1-to-1 relationship

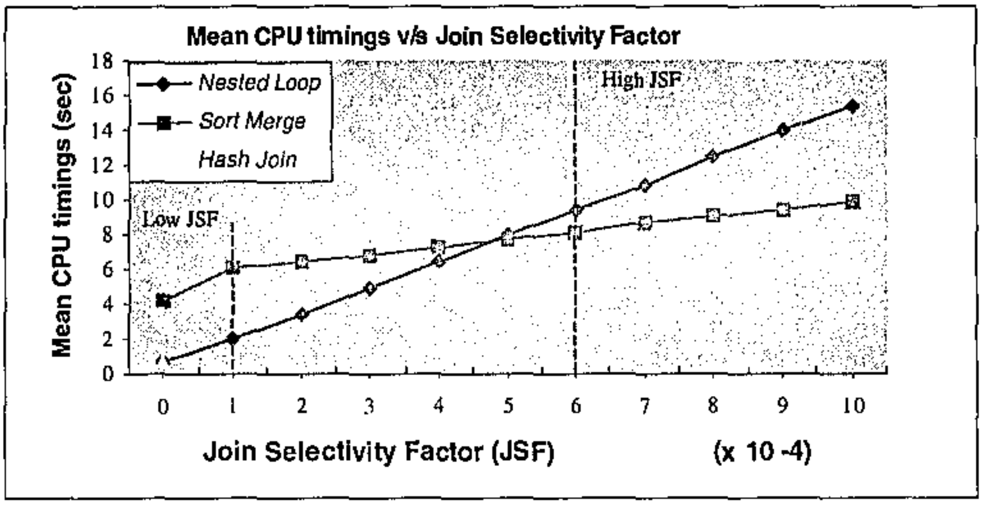


Figure 21: Effect of Join Selectivity on CPU Time for a 1-to-10 relationship

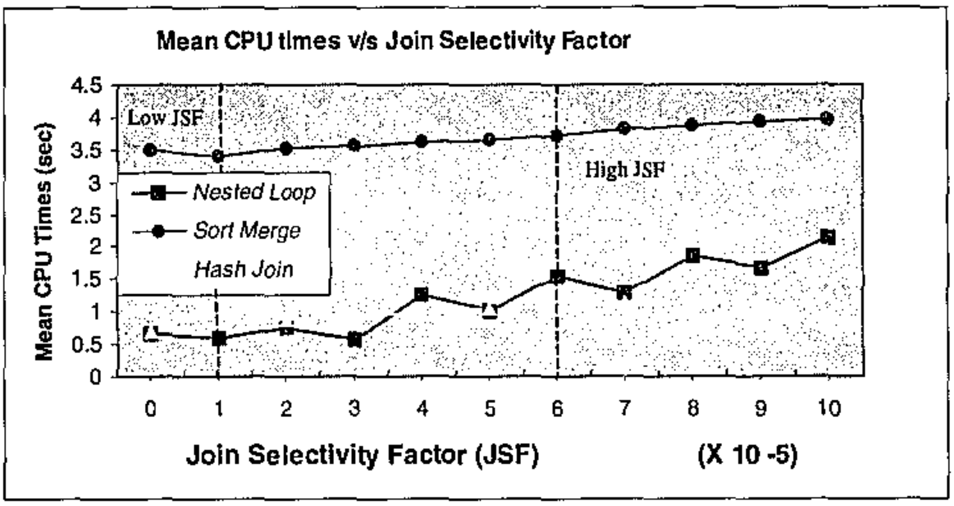


Figure 22: Effect of Join Selectivity on CPU Time for a 1-to-1 relationship

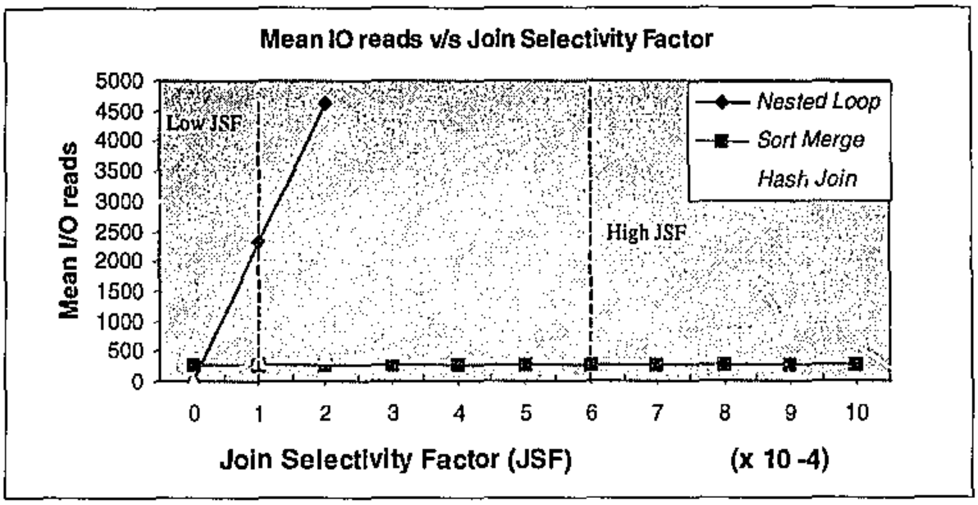


Figure 23: Effect of Join Selectivity on I/O Reads for a 1-to-10 relationship

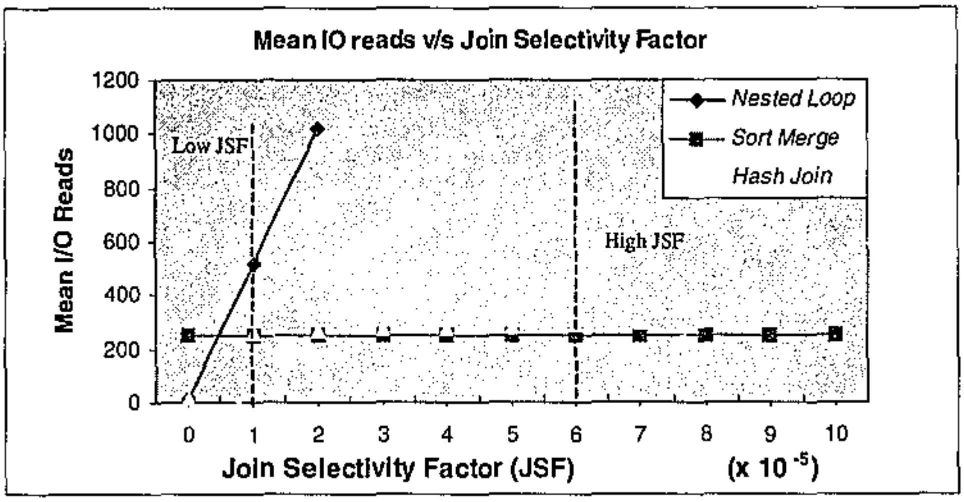


Figure 24: Effect of Join Selectivity on I/O reads for a 1-to-1 relationship

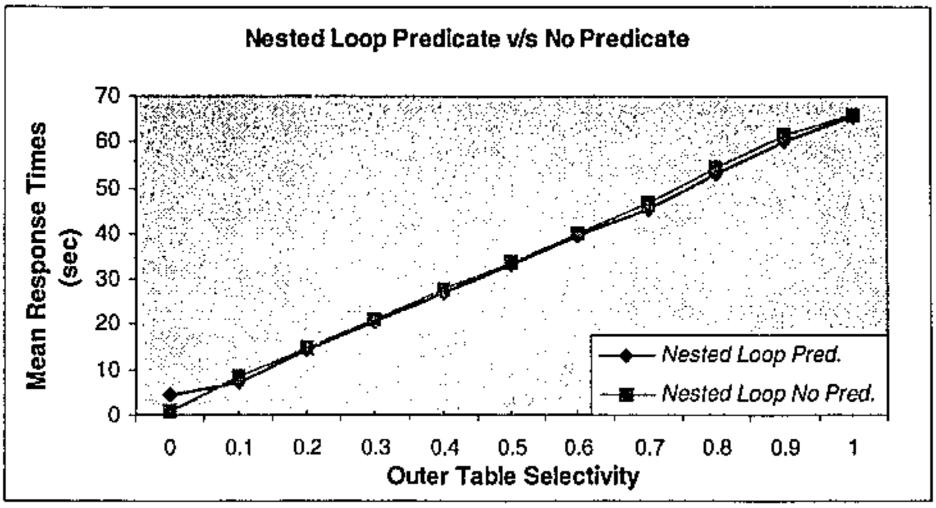


Figure 25: Effect of applying a predicate on inner table for the Nested Loop join method for a 1-to-10 relationship.

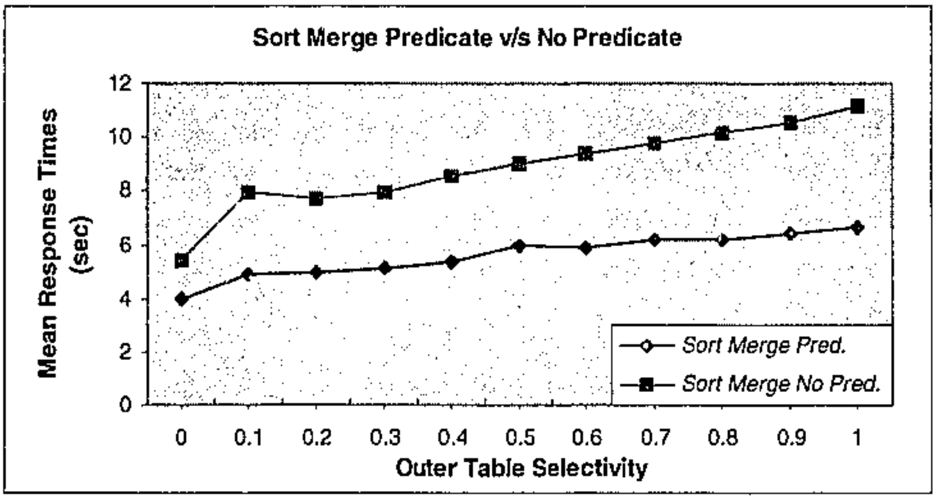


Figure 26: Effect of applying a predicate on inner table for the Sort Merge join method for a 1-to-10 relationship.

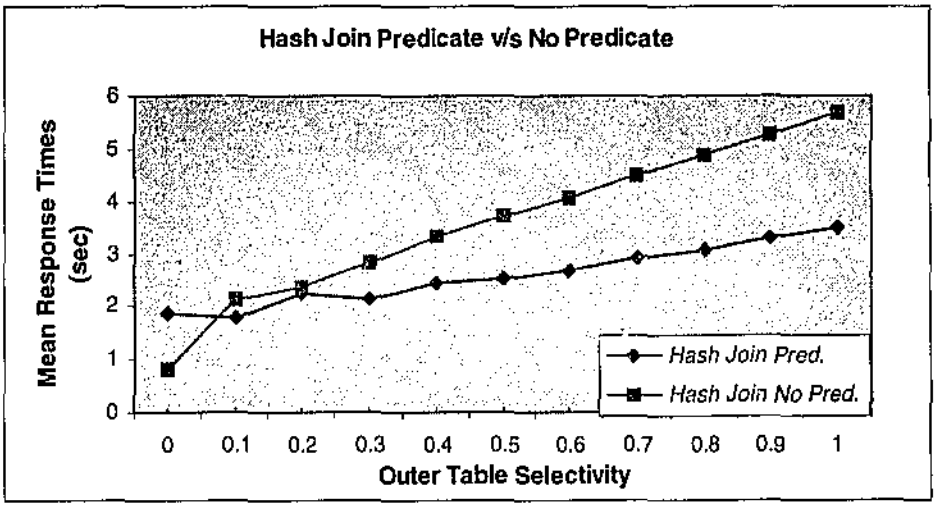


Figure 27: Effect of applying a predicate on inner table for the Hash Join method for a 1-to-10 relationship.

Chapter Six: Conclusion

The response time refers to the total time taken for a query statement to execute. The time includes the time taken by the CPU to process the query as well as the time taken by the data blocks to be retrieved from disk.

The access time on a disk consists of three parts:

- seek time – time to move the disk head to the proper cylinder
- latency time – time to wait for the data to move under the appropriate read/write head (March & Carlis, 1985).
- data transfer time – transfer data from disk to memory

These times depend on the location of data relative to the disk head. To ensure that the data collected is a valid representation of the time measured, the experiments were run several times and the mean of the individual data was used.

The experiments were also restricted to a single disk. The indexes and the tables were kept on the same disk. The use of two separate disks would have reduced the cost of the nested loop as indexes could have been read at the same time as the tables. The separation of disk drives allows the disk head to read the data tables while another disk head residing on the other disk reads the indexes.

Findings

The findings of this study are:

- Overall, the hash join performs better than the sort merge and the nested loop under all varying conditions. The hash join has an advantage over the sort merge in that hashing requires only one relation to be hold in memory whereas the sort merge requires both relations in memory. The hash join also competes well with the nested loop join method, as a single scan of the inner relation is required in the case of the hash join.

- The nested loop join method is I/O intensive. The nested loop is efficient when a small number of tuples participate in the join. It was found that the nested loop competes well with the hash join at low selectivity factor. However, at high join selectivity factor, the nested loop is the worst join method to be used. The results obtained from the experiments carried out by DataBase Technology Institute, IBM (1991) also showed the nested loop to be the worst join method at high selectivity factor even when two separate disks were used for the indexes and the tables.

- The sort merge and the hash join perform well with filtering present on the inner table. In both the above join methods, less tuples are retrieved from the database and consequently, less data are to be processed. The presence of a predicate on the inner table does not affect the nested loop join method as all the records from the inner table are read and processed (Refer algorithm on page 19).

Database Tuning

The experiments for this research were run in a controlled environment. In the real world, there are other factors that may impact on the performance of join methods. The costs of retrieving data from server to client can be significant. For example, the physical distance between the client and the server and the packet size play an important role in the network cost. In order to reduce the network costs, the nested loop join method is usually implemented as a block read instead of a tuple read.

The database system also needs to be carefully tuned to make optimum use of available memory and to reduce the number of disk input/output accesses (disk I/Os). The buffer cache, which holds copies of the table blocks, the sorted data and indexes is a critical area of memory. A small buffer cache means that data needs to be fetched constantly from disk to buffer cache. Alternatively, increasing the buffer cache reduces the number of disk I/Os required as less fetches are needed. Similarly, the number of disk I/Os can be reduced by increasing the size of the sort area. A small sort area may require several runs for the data to be sorted and therefore more I/O accesses are required. The sort area parameter is especially useful for joins involving the sort merge join method.

The number of disk I/Os can also be reduced by spreading the disk load across devices and controllers. For example, the use of two separate disks for storing the tables and the indexes can reduce the time required to access a block since both tables and indexes can be read at the same time.

Recommendations

In light of the above discussion on database tuning and the results of the current experiments, the following recommendations can be made for a join query processing using a large and a small table:

- The hash join method should be used for most cases. This join method has a good rating under the different conditions. The cost-based optimiser present in Oracle database system determines the join method to be used for a join query. However, the join method chosen by the optimiser can be changed by the use of hints in the query statement. For example, the following query statement uses a hash join method:

```
SELECT /* +USE_HASH(QUO,CUS) */
cus.name, quo.quote_no
FROM quotes quo, customers cus
WHERE cus.cust_id = quo.cust_id
AND postcode < 6401
```

- The nested loop join method can be used when the number of tuples participating in the join is less than 10% of the maximum number of tuples that could participate in the join. The nested loop join method requires an index to be present on the join column of the inner table. To ensure that the index of the inner table is used instead of the index of the outer table, the query hint 'USE_INDEX (inner table, outer table)' can be defined in the query statement. For example,

```
SELECT /* +USE_INDEX(QUO,CUS) */
cus.name, quo.quote_no
FROM quotes quo, customers cus
WHERE cus.cust_id = quo.cust_id
AND postcode < 6401
```

  When a large number of rows is retrieved from a join relation, the nested loop performs poorly. The sort merge or the hash join should be used instead.

- The sort merge join method is efficient when a filter condition is defined against the inner table. Consequently, the number of tuples retrieved from the inner table is reduced. For example, in the query statement below, a predicate is defined aginst the inner table:

```
SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
 AND amount < 50000  -- predicate applied to inner table
```

The hash join also competes well with the sort merge in this case.

Potential Future Research

Commercial database vendors are now marketing object-oriented database management systems as a solution to the requirements of modern business. This research could be extended to consider the effect of varying selectivity on the performance of join methods in an object-oriented database management system. An object-oriented database system consists of a set of objects that are connected through their attributes. The objects communicate with each other through methods. The main difference between the object-oriented model and the entity-relational model is that objects have methods as well as attributes.

> *Just remember – when you think all is thought out, the future remains.*
>
> (Based on the original idea of Bob Goddard).

APPENDIX A - Initialisation Files and Set Up

Database Initialisation File

```
#
# $Header: init.ora 1.2 94/10/18 16:12:36 gdudey Osd<desktop/netware> $
init.ora Copyr (c) 1991 Oracle
#
###################################################################
###
# Example INIT.ORA file
#
# This file is provided by Oracle Corporation to help you customize
# your RDBMS installation for your site. Important system parameters
# are discussed, and example settings given.
#
# Some parameter settings are generic to any size installation.
# For parameters that require different values in different size
# installations, three scenarios have been provided: SMALL, MEDIUM
# and LARGE. Any parameter that needs to be tuned according to
# installation size will have three settings, each one commented
# according to installation size.
#
# Use the following table to approximate the SGA size needed for the
# three scenarios provided in this file:
#
#              -------Installation/Database Size------
#               SMALL        MEDIUM        LARGE
#  Block     2K   4500K        6800K        17000K
#  Size     4K   5500K        8800K        21000K
#
# To set up a database that multiple instances will be using, place
# all instance-specific parameters in one file, and then have all
# of these files point to a master file using the IFILE command.
# This way, when you change a public
# parameter, it will automatically change on all instances. This is
# necessary, since all instances must run with the same value for many
# parameters. For example, if you choose to use private rollback segments,
# these must be specified in different files, but since all gc_*
# parameters must be the same on all instances, they should be in one file.
#
# INSTRUCTIONS: Edit this file and the other INIT files it calls for
# your site, either by using the values provided here or by providing
# your own. Then place an IFILE= line into each instance-specific
# INIT file that points at this file.
###################################################################
###
```

```
db_name = oracle
db_files = 20
control_files = (C:\ORANT\DATABASE\ctl1orcl.ora,
C:\ORANT\DATABASE\ctl2orcl.ora)

compatible = 7.3.0.0.0

db_file_multiblock_read_count =  16 # INITIAL
# db_file_multiblock_read_count = 8                       # SMALL
# db_file_multiblock_read_count = 16                       # MEDIUM
# db_file_multiblock_read_count = 32                       # LARGE

db_block_buffers =  200                  # INITIAL
# db_block_buffers = 200                              # SMALL
# db_block_buffers = 550                              # MEDIUM
# db_block_buffers = 3200                              # LARGE

shared_pool_size =  6500000                  # INITIAL
# shared_pool_size = 3500000                          # SMALL
# shared_pool_size = 6000000                          # MEDIUM
# shared_pool_size = 9000000                          # LARGE

log_checkpoint_interval = 10000

processes =  50                          # INITIAL
# processes = 50                                  # SMALL
# processes = 100                                  # MEDIUM
# processes = 200                                  # LARGE

dml_locks =  100                           # INITIAL
# dml_locks = 100                                  # SMALL
# dml_locks = 200                                  # MEDIUM
# dml_locks = 500                                  # LARGE

log_buffer =  8192                         # INITIAL
# log_buffer = 8192                                 # SMALL
# log_buffer = 32768                                 # MEDIUM
# log_buffer = 163840                                 # LARGE

sequence_cache_entries =  10          # INITIAL
# sequence_cache_entries = 10                           # SMALL
# sequence_cache_entries = 30                           # MEDIUM
# sequence_cache_entries = 100                           # LARGE

sequence_cache_hash_buckets =  10    # INITIAL
# sequence_cache_hash_buckets = 10                       # SMALL
# sequence_cache_hash_buckets = 23                       # MEDIUM
```

```
# sequence_cache_hash_buckets = 89                                  # LARGE

# audit_trail = true            # if you want auditing
timed_statistics = true        # if you want timed statistics
max_dump_file_size = 10240     # limit trace file size to 5 Meg each

# log_archive_start = true      # if you want automatic archiving

# define directories to store trace and alert files
background_dump_dest=%RDBMS73%\trace
user_dump_dest=c:\AMALLET\trace
db_block_size = 2048
hash_multiblock_io_count=16
optimizer_mode=RULE
UTL_FILE_DIR=c:\AMALLET\script
snapshot_refresh_processes = 1

remote_login_passwordfile = shared

text_enable = true
```

Creation of Tablespaces

```
CREATE TABLESPACE SMALL_TABLES
DATAFILE 'C:\ORANT\DBS\SMALL_TABLES.DBF'
SIZE 20M
/
CREATE TABLESPACE LARGE_TABLES
DATAFILE 'C:\ORANT\DBS\LARGE_TABLES.DBF'
SIZE 20M
/
CREATE TABLESPACE USER_INDEXES
DATAFILE 'C:\ORANT\DBS\USER_INDEXES.DBF'
SIZE 20M
/
CREATE TABLESPACE TEMP
DATAFILE 'C:\ORANT\DBS\TEMP.DBF'
SIZE 10M
/
 ALTER USER ada
    IDENTIFIED BY ada
    DEFAULT TABLESPACE large_tables
    TEMPORARY TABLESPACE temp
    QUOTA UNLIMITED ON temp
    QUOTA UNLIMITED ON small_tables
    QUOTA UNLIMITED ON large_tables
```

```
QUOTA UNLIMITED ON user_indexes
```

APPENDIX B - Program Coding

Creation of Packages, Procedures and Functions

Package Random

```
CREATE OR REPLACE PACKAGE Random AS
/* Random number generator. Uses the same algorithm as the
  rand() function in C. */

 -- Used to change the seed. From a given seed, the same
 -- sequence of random numbers will be generated.
PROCEDURE ChangeSeed(p_NewSeed IN NUMBER);

 -- Return a random integer between 1 and 32767.
FUNCTION Rand RETURN NUMBER;
-- PRAGMA RESTRICT_REFERENCES(Rand, WNDS, WNPS);

 -- Same as Rand, but with a procedural interface.
PROCEDURE GetRand(p_RandomNumber OUT NUMBER);

 --Returns a random integer between 1 and p_MaxVal.
FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER;
-- PRAGMA RESTRICT_REFERENCES(RandMax, WNDS);

 -- Same as RandMax, but with a procedural interface.
PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
              p_MaxVal IN NUMBER);
END Random;
/

create or replace package body Random IS

/* Used for calculating the next number. */
v_Multiplier   CONSTANT NUMBER := 22695477;
v_increment    CONSTANT NUMBER := 1;

/* Seed used to generate random sequence. */
v_Seed  number := 1;
v_Count number := 0;

PROCEDURE ChangeSeed(p_NewSeed IN NUMBER) IS
BEGIN
 v_Seed := p_NewSeed;
END ChangeSeed;

FUNCTION Rand RETURN NUMBER IS
```

```
BEGIN
  v_Seed := MOD(v_Multiplier * v_Seed + v_Increment, (2 ** 32));
  RETURN BITAND(v_Seed/(2 ** 16), 32767);
END Rand;

PROCEDURE GetRand(p_RandomNumber OUT NUMBER) IS
BEGIN
  -- Simply call Rand and return the value.
  p_RandomNumber := Rand;
END GetRand;

FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER IS
BEGIN
  RETURN MOD(Rand, p_MaxVal) + 1;
END RandMax;

PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                  p_MaxVal IN NUMBER) IS
BEGIN
  -- Simply call RandMax and return the value
  p_RandomNumber := RandMax(p_MaxVal);
END GetRandMax;

BEGIN
  /* Package initialization.  Initialize the seed to the current
     time in seconds. */
  v_count := v_count + 1;
  IF mod(v_count, 6) = 0 THEN
     ChangeSeed(TO_NUMBER(TO_CHAR(SYSDATE,'SSSSS'))*147);
  ELSIF mod(v_count, 6) = 3 THEN
    ChangeSeed(TO_NUMBER(TO_CHAR(SYSDATE,'SSSSS'))*587);
  ELSE
    ChangeSeed(TO_NUMBER(TO_CHAR(SYSDATE,'SSSSS')));
  END IF;
END Random;
/
```

Package Array

```
REM
REM PACKAGE
REM      array

PROMPT
PROMPT Creating Package Specification array
CREATE OR REPLACE PACKAGE array IS
-- **************************************************
-- Author        :-  Ada Mallet
```

```
-- Date Created  :- 5/7/97
--
-- ********************************
--This package contains functions and procedures to initialise, add,
-- update and delete records from a PL/SQL table (OR array).
-- The package is also used to generate a table with unique number
-- that does not follow a sequential order. An array A1 is first initialised with
-- unique sequential number. Every time a random number is
-- generated, it is placed in another array A2 and that number is
-- removed from A1. If a generated number already
-- exists in A2, then a number form A1 is picked. This process ensures
-- a unique number in array A2.

 PROCEDURE add_row(p_row IN NUMBER);
 FUNCTION get_last_row RETURN NUMBER;
 FUNCTION get_row(p_index IN BINARY_INTEGER) RETURN NUMBER;
 PROCEDURE set_row(p_value IN NUMBER);
 PROCEDURE clear_rows;
 FUNCTION retrieve_row(p_index IN BINARY_INTEGER)
      RETURN NUMBER;
 PROCEDURE populate_array
     (p_max_array INTEGER);

 PRAGMA RESTRICT_REFERENCES(get_row, WNDS, WNPS, RNDS);

END array;
/

REM
PROMPT
PROMPT Creating Package Body array
CREATE OR REPLACE PACKAGE BODY array IS
    TYPE row_array_type IS TABLE OF NUMBER(6) INDEX BY
    BINARY_INTEGER;
    TYPE row_array_type1 IS TABLE OF NUMBER(6) INDEX BY
    BINARY_INTEGER;
    vrow_array ROW_ARRAY_TYPE
    vrow_array1 ROW_ARRAY_TYPE1;
    vrow_index BINARY_INTEGER DEFAULT 0;
    vrow_index1 BINARY_INTEGER DEFAULT 0;

PROCEDURE add_row(p_row IN NUMBER)
IS
     -- This procedure adds details of a
     -- row to an array and assigns the record
     -- a unique number in the array
BEGIN
     vrow_index := vrow_index + 1;
```

```
     vrow_array(vrow_index) := p_row;
END add_row;

PROCEDURE set_row(p_value IN NUMBER)
IS
     -- This procedure assigns a number to the next
     -- row in the array.

BEGIN
      vrow_index1 := vrow_index1 + 1;
      vrow_array1(vrow_index1) := p_value;
END set_row;

FUNCTION get_last_row RETURN NUMBER
IS
     -- This procedure returns the value that is stored
     -- in the last row of the array.

     v_return NUMBER(6);
     v_index BINARY_INTEGER;
BEGIN
     v_index := vrow_array.LAST;
     v_return := vrow_array(v_index);
     vrow_array.DELETE(v_index);
     RETURN(v_return);
END get_last_row;

FUNCTION get_row(p_index IN BINARY_INTEGER) RETURN NUMBER
IS
     -- This procedure retrieves the value of a
     -- particular row in the array.
     v_return NUMBER(6);
BEGIN
       v_return := vrow_array1(p_index);
     RETURN(v_return);
END get_row;

PROCEDURE clear_rows
IS
     -- This procedure clears all rows
     -- currently in the two arrays
BEGIN
     WHILE vrow_index > 0 LOOP
       vrow_array(vrow_index) := NULL;
       vrow_index := vrow_index - 1;
     END LOOP;

     WHILE vrow_index1 > 0 LOOP
```

```
      vrow_array1(vrow_index1) := NULL;
       vrow_index1 := vrow_index1 - 1;
    END LOOP;

    vrow_array.DELETE;
    vrow_array1.DELETE;
END clear_rows;

FUNCTION retrieve_row(p_index IN BINARY_INTEGER)
   RETURN NUMBER
IS
    -- This procedure retrieves details of a
    -- particular row from an array  and deletes
    -- the row from the array.
    v_number NUMBER(6);
BEGIN
   IF vrow_array.EXISTS(p_index) THEN
      v_number := vrow_array(p_index);
      vrow_array.DELETE(p_index);
     RETURN (v_number);
   ELSE
     RETURN(0);
   END IF;
END retrieve_row;

PROCEDURE populate_array
     (p_max_array INTEGER) AS

    -- This procedure populates the first array with a unique
    -- sequential number.

  v_number BINARY_INTEGER := 1;
BEGIN
  array.clear_rows;
  LOOP
   array.add_row(v_number);
   v_number := v_number + 1;
   EXIT WHEN v_number > p_max_array;
  END LOOP;
END populate_array;

END array;
/
```

Package ReadFile

```
REM
REM PACKAGE
```

```
REM     readfile

PROMPT
PROMPT Creating Package Specification readfile

CREATE OR REPLACE PACKAGE readfile IS
-- ****************************************************
-- Author          :-  Ada Mallet
-- Date Created   :-  21/10/97
--
-- ****************************************************
-- This package is used read a file and store the values in an array.

  TYPE array_type IS TABLE OF VARCHAR2(100) INDEX BY
BINARY_INTEGER;
  array_out ARRAY_TYPE;
  v_index INTEGER;

 PROCEDURE file_to_array (loc_in IN VARCHAR2, file_in IN VARCHAR2);
 FUNCTION get_row(p_index IN INTEGER) RETURN NUMBER;

 PRAGMA RESTRICT_REFERENCES(get_row, WNDS, RNDS);

END readfile;
/

REM
PROMPT
PROMPT Creating Package Body readfile
CREATE OR REPLACE PACKAGE BODY readfile IS

 PROCEDURE clear_array
 IS
     -- This procedure clears all records
     -- currently in the array
   BEGIN
     WHILE v_index > 0 LOOP
       array_out(v_index) := NULL;
       v_index := v_index - 1;
     END LOOP;

   END clear_array;


PROCEDURE get_nextline
        (file_in IN UTL_FILE.FILE_TYPE,
         line_out OUT VARCHAR2,
         eof_out OUT BOOLEAN)
```

```
IS
      -- This procedure gets the next line from
      -- the file to be read
BEGIN
        UTL_FILE.GET_LINE (file_in, line_out);
        eof_out := FALSE;
EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
                line_out := NULL;
                eof_out  := TRUE;
END;

PROCEDURE file_to_array
  (loc_in IN VARCHAR2, file_in IN VARCHAR2)
IS
  /* Open file and get handle right in declaration */
  names_file UTL_FILE.FILE_TYPE;

  /* counter used to create the Nth name. */
  line_counter INTEGER := 1;

  end_of_file BOOLEAN := FALSE;
BEGIN
  clear_array;
  names_file := UTL_FILE.FOPEN (loc_in, file_in, 'R');
  WHILE NOT end_of_file
  LOOP
   v_index := line_counter;
   get_nextline (names_file, array_out(line_counter), end_of_file);
   line_counter := line_counter + 1;

  END LOOP;
  UTL_FILE.FCLOSE (names_file);
END;

FUNCTION get_row(p_index IN INTEGER) RETURN NUMBER
 IS
     -- This procedure retrieves details of a
     -- row from an array and then removes the
     -- record from the array
     v_return VARCHAR2(100);
   BEGIN
       v_return := array_out(p_index);
     RETURN(TO_NUMBER(v_return));
 END get_row;

END readfile;
```

```
/
```

Package Sequence

```
CREATE OR REPLACE PACKAGE sequence IS
  PROCEDURE get_next_sequence(p_random IN INTEGER);
END sequence;
/

CREATE OR REPLACE PACKAGE BODY sequence IS
 PROCEDURE get_next_sequence (p_random IN INTEGER)
 IS
     -- This procedure populates the array with
     -- unique random values. If a generated random
     -- number already exists in the array, then a number
     -- from another array is used.

  v_random NUMBER;  /* store random number */
  v_return NUMBER;  /* store the first row of array */
  v_count INTEGER := 0;  /* counter */
  v_number NUMBER;      /* store number retrieved from array */
  v_max_random INTEGER := p_random + 1;
BEGIN
  LOOP
    v_count := v_count + 1;
    v_random := random.RandMax(p_random);
    v_number := array.retrieve_row(v_random);
    BEGIN
      IF v_number <> 0 THEN
        array.set_row(v_number);
      ELSE
       v_return := array.get_last_row;
       array.set_row(v_return);
      END IF;
    EXCEPTION WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE( 'no more data to be read');
    END;
EXIT WHEN v_count = p_random;
  END LOOP;
END get_next_sequence;

END sequence;
/
```

Package Table_Sizing

Amount of Space occupied by tables

```
CREATE OR REPLACE PACKAGE TABLE_SIZING AS
 -- FUNCTION Get_block_Size RETURN NUMBER;
  PROCEDURE table_size (tablename_in IN VARCHAR2,
                  tablesize_out IN OUT NUMBER );
END TABLE_SIZING;
/
CREATE OR REPLACE PACKAGE BODY TABLE_SIZING AS

  Block_size NUMBER;
  Block_Header_PartA NUMBER;
  Block_Header_PartB NUMBER;

/*  FUNCTION Get_block_Size RETURN NUMBER
  IS
    db_blocksize NUMBER;
  BEGIN
    BEGIN
      select value
        into db_blocksize
        from v$parameter
       where name = 'db_block_size';
      exception
      when others then
        db_blocksize := 2048;
    END;
    RETURN (db_blocksize);
  END Get_Block_Size; */

 FUNCTION TOTAL_BLOCK_HEADER_SIZE( INITTRANS_IN IN
NUMBER DEFAULT 1 )
   RETURN NUMBER
   IS
    Fixed_Header CONSTANT NUMBER := 57;
    Table_Directory CONSTANT NUMBER := 4;
    --
  BEGIN
    -- Block header, part A = fixed header + variable transaction header
    Block_Header_PartA := Fixed_Header + ( 23 * INITTRANS_IN );

    -- Block header, part B = table directory + row directory
    Block_Header_PartB := Table_Directory; -- +  ( 2 * Rows_In_Block_IN );
    RETURN ( Block_Header_PartA + Block_Header_PartB );
  END Total_Block_Header_Size;

  FUNCTION Space_Per_Block( Header_Size_In IN NUMBER,
                  PctFree_In IN NUMBER )
   RETURN NUMBER
   IS
```

```
    return_value NUMBER;
  BEGIN
    return_value := ( block_size - Header_Size_In ) -
                 ( ( block_size - Block_Header_PartA ) * (
PctFree_In/100 ) );
    RETURN (return_value);
  END Space_Per_Block;

  FUNCTION avg_column_size(table_name_in in VARCHAR2,
                       column_name_in in VARCHAR2 )
   RETURN NUMBER
   IS
     avg_col_size NUMBER;
     cursor_handle INTEGER;
     execute_feedback INTEGER;
  BEGIN
    cursor_handle := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE( cursor_handle,
        'SELECT AVG(NVL(VSIZE('||column_name_in||'),0)) ' ||
        'FROM ' || table_name_in, 2 );
    DBMS_SQL.DEFINE_COLUMN(cursor_handle, 1, avg_col_size );
    execute_feedback := DBMS_SQL.EXECUTE_AND_FETCH
        (cursor_handle,true);
    DBMS_SQL.COLUMN_VALUE( cursor_handle, 1, avg_col_size );
    DBMS_SQL.CLOSE_CURSOR( cursor_handle );
    avg_col_size := NVL( avg_col_size, 0 );
    RETURN ( avg_col_size );
  END Avg_Column_Size;

  FUNCTION Calculate_Combined_Data_Space( tablename_in IN VARCHAR2
)
   RETURN NUMBER
   IS
    DataUsage NUMBER := 0;
  BEGIN
    for column_rec in ( select table_name, column_name
                       from user_tab_columns
                      where table_name = tablename_in )
    loop
       DataUsage := DataUsage + Avg_Column_size( tablename_in,
                                   column_rec.column_name );
    end loop;
    RETURN ( DataUsage );
  END Calculate_Combined_Data_Space;

  FUNCTION Total_Average_Row_Size( table_name_in IN VARCHAR2 ,
                         Step3_Combined_Dataspace IN NUMBER )
   RETURN NUMBER IS
```

```
    RowHeader CONSTANT NUMBER := 3;
    F_plus_v NUMBER;
    nReturn_Value NUMBER;
  BEGIN
    SELECT SUM( DECODE(GREATEST(DATA_LENGTH,250),250,1,3 ) )
      INTO F_PLUS_V
      FROM USER_TAB_COLUMNS
     WHERE TABLE_NAME = table_name_in;
    nReturn_Value := RowHeader + F_Plus_V + Step3_Combined_Dataspace;
    -- The absolute minimum rowsize of a non-clustered row is 9 bytes.
    RETURN ( GREATEST( nReturn_Value, 9 ) );
  END Total_Average_Row_Size;

  FUNCTION get_num_rows( tablename_in in varchar2 ) RETURN NUMBER
IS
    results number;
    cursor_handle integer;
    execute_feedback integer;
  BEGIN
    cursor_handle := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE( cursor_handle,
          'SELECT COUNT(*) ' ||
          'FROM ' || tablename_in, 2 );
    DBMS_SQL.DEFINE_COLUMN(cursor_handle, 1, results );
    execute_feedback := DBMS_SQL.EXECUTE_AND_FETCH( cursor_handle,
true);
    DBMS_SQL.COLUMN_VALUE( cursor_handle, 1, results );
    RETURN (results);
  END get_num_rows;

  PROCEDURE Table_Size( tablename_in IN VARCHAR2,
                  tablesize_out IN OUT NUMBER ) is
    db_IniTrans NUMBER;
    db_PctFree NUMBER;
    Header_size NUMBER;
    Available_Data_Space NUMBER;
    Combined_Data_Space NUMBER;
    Avg_Row_Size NUMBER;
    Rows_Per_Block NUMBER;
    Number_Of_Rows NUMBER;
    DEFAULT_INITIAL_EXTENT CONSTANT NUMBER := 10240;
  BEGIN
    select ini_trans, pct_Free
      into db_IniTrans,
          db_PctFree
      from User_Tables
     where table_name = tablename_in;
```

```
    -- step 1: Calculate the total block header size ( excludes row
    -- directory - 2*R )
    Header_size := Total_Block_Header_Size( db_IniTrans );
    Available_Data_Space := Space_Per_Block( Header_Size, db_PctFree );
    Combined_Data_Space := Calculate_Combined_Data_Space( tablename_in );
    Avg_Row_Size := Total_Average_Row_Size( tablename_in,
                                  Combined_Data_Space );

    -- R (avg. # of rows/block) = available space / average row size;
    Rows_Per_Block := TRUNC( Available_Data_Space / ( 2 + Avg_Row_Size )
);
    Number_Of_Rows := Get_Num_rows( tablename_in );

    tablesize_out := ( ceil( Number_Of_Rows / Rows_Per_Block ) * block_size );
  END Table_Size;

BEGIN
  -- Package Initialization
  block_size := 2048;
END TABLE_SIZING;
/

-- get the size of tables CUSTOMERS, QUOTES, QUOTE
DECLARE
  v_int INTEGER;
BEGIN
  table_sizing.table_size('CUSTOMERS', v_int);
  DBMS_OUTPUT.PUT_LINE('the size of customers is '||v_int);
  table_sizing.table_size('QUOTES', v_int);
  DBMS_OUTPUT.PUT_LINE('the size of quotes is '||v_int);
  table_sizing.table_size('QUOTE', v_int);
  DBMS_OUTPUT.PUT_LINE('the size of quote is '||v_int);
END;
```

Procedure Get_Amount

```
CREATE OR REPLACE FUNCTION get_amount
  (p_amount NUMBER, p_seq INTEGER)
   RETURN NUMBER
AS
    -- This function is used to update the
     -- amount value with a unique value that
     -- does not follow a sequential order.
     -- Amount value of $50000 are not
     -- considered.
BEGIN
  IF p_amount <> 50000 THEN
    RETURN(array.get_row(p_seq));
```

```
  ELSE
    RETURN p_amount;
  END IF;
END get_amount;
/
```

Procedure Set_Quote

```
CREATE OR REPLACE PROCEDURE set_quote
AS
      -- This procedure is used to update the
      -- join attribute value (customer id) with a
      -- random value. A quote number is chosen
      -- at random and its customer value is then
      -- updated with a random value ranging
      -- from 1 to 1000.
v_count INTEGER(5) := 0;
BEGIN
  array.populate_array(10000);
  sequence.get_next_sequence(10000);
  readfile.file_to_array('d:\script', 'ranq5.lis');
  LOOP
      v_count := v_count+ 1;
      IF v_count > 1001  THEN
           EXIT;
      ELSE
         UPDATE QUOTE
         SET cust_id = readfile.get_row(v_count)
         WHERE quote_no = array.get_row(v_count)+1000;
     END IF;
--   EXIT WHEN v_count > 1001;
  END LOOP;
  COMMIT;
END setquote;
/
```

Creation of Tables

Customers Table

```
-- Author : Ada Mallet
-- Date   : 25/08/1997
-- Purpose: Create customer tables
--
***************************************************************
*
```

```
DROP TABLE customers CASCADE CONSTRAINTS;
DROP TABLE customers_small CASCADE CONSTRAINTS;
DROP TABLE customers_temp CASCADE CONSTRAINTS;
DROP SEQUENCE customer_seq;
DROP SEQUENCE postcode_seq;
DROP SEQUENCE customersm_seq;

-- create the customer table and the intermediate tables
-- in the small tablespace and the indexes in the index
-- tablespace
--
************************************************************************
**

CREATE TABLE    customers
 (cust_id       NUMBER(5) CONSTRAINT cust_pk PRIMARY KEY,
  name          VARCHAR2(10) NOT NULL,
  state         VARCHAR2(3) NOT NULL,
  postcode      NUMBER(4) NOT NULL)
  TABLESPACE SMALL_TABLES
  ENABLE PRIMARY KEY USING INDEX TABLESPACE USER_INDEXES;

CREATE TABLE    customers_small
  (cust_id      NUMBER(5) CONSTRAINT custsm_pk PRIMARY KEY,
  name          VARCHAR2(10) NOT NULL,
  state VARCHAR2(3) NOT NULL,
  postcode      NUMBER(4) NOT NULL)
  TABLESPACE SMALL_TABLES
  ENABLE PRIMARY KEY USING INDEX TABLESPACE USER_INDEXES;

CREATE TABLE    customers_temp
  (cust_id      NUMBER(5) CONSTRAINT custtp_pk PRIMARY KEY,
  name          VARCHAR2(10) NOT NULL,
  state VARCHAR2(3) NOT NULL,
  postcode      NUMBER(4) NOT NULL)
  TABLESPACE SMALL_TABLES
  ENABLE PRIMARY KEY USING INDEX TABLESPACE USER_INDEXES;

-- populate tables
--
************************************************************************
**

-- populate tables with 12 rows
INSERT INTO customers_small VALUES
(1,'ECU','WA', 6050);
INSERT INTO customers_small VALUES
```

```
(2,'Ada Mallet','NSW', 6004);
INSERT INTO customers_small VALUES
(3,'Sage Com','WA', 6025);
INSERT INTO customers_small VALUES
(4,'Edgar Mic','WA', 6005);
INSERT INTO customers_small VALUES
(5,'Australian','VIC',5006) ;
INSERT INTO customers_small VALUES
(6,'Conservati','NSW',2003);
INSERT INTO customers_small VALUES
(7,'Peter','NSW',2003);
INSERT INTO customers_small VALUES
(8,'Mark Ric','NSW',2003);
INSERT INTO customers_small VALUES
(9,'Newface','WA',6639);
INSERT INTO customers_small VALUES
(10,'Business','NSW',2056);
INSERT INTO customers_small VALUES
(11,'Peterson','VIC',1887);
INSERT INTO customers_small VALUES
(12,'Oracle','WA',1025);

-- use the sequencing to generate unique number
-- for customer id
CREATE SEQUENCE customer_seq START WITH 13;

-- populate table with 24 rows
INSERT INTO customers_small
  SELECT customer_seq.nextval,
       name, state, postcode
  FROM customers_small;

COMMIT;

-- populate table with 48 rows
INSERT INTO customers_small
  SELECT customer_seq.nextval,
       name, state, postcode
  FROM customers_small;

COMMIT;

-- populate table with 96 rows
INSERT INTO customers_small
  SELECT customer_seq.nextval,
       name, state, postcode
  FROM customers_small;
```

```
COMMIT;

-- add 4 more rows
INSERT INTO customers_small VALUES
(97,'Jean Hall','WA',6050);
INSERT INTO customers_small VALUES
(98,'Pierre Ric','WA',6141);
INSERT INTO customers_small VALUES
(99,'Sylvie Van', 'VIC', 1224);
INSERT INTO customers_small VALUES
(100,'A Appadu','NSW',5141);

COMMIT;

-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'rand1.lis');

CREATE SEQUENCE customersm_seq START WITH 1;

-- populate intermediate table with 100 rows
INSERT INTO customers_temp
  SELECT readfile.get_row(customersm_seq.nextval),
       name, state, postcode
   FROM customers_small;

COMMIT;

-- populate intermediate table with 200 rows
INSERT INTO customers_temp
  SELECT readfile.get_row(customersm_seq.nextval),
       name, state, postcode
   FROM customers_temp;

COMMIT;

-- populate intermediate table with 400 rows
INSERT INTO customers_temp
  SELECT readfile.get_row(customersm_seq.nextval),
       name, state, postcode
   FROM customers_temp;

COMMIT;

-- populate intermediate table with 500 rows
INSERT INTO customers_temp
  SELECT readfile.get_row(customersm_seq.nextval),
       name, state, postcode
   FROM customers_small;
```

```
COMMIT;

-- populate intermediate table with 1000 rows
INSERT INTO customers_temp
  SELECT readfile.get_row(customersm_seq.nextval),
       name, state, postcode
   FROM customers_temp;

COMMIT;

-- Creating customer table with random number for post code

-- EXEC array.populate_array(1000);

EXEC readfile.file_to_array('d:\script', 'rand2.lis');

--EXEC sequence.get_next_sequence(1000);
CREATE SEQUENCE postcode_seq START WITH 1;
INSERT INTO customers
  SELECT cust_id,name, state,
       readfile.get_row(postcode_seq.nextval) + 6000
   FROM customers_temp;
COMMIT;

DROP TABLE customers_temp CASCADE CONSTRAINTS;
DROP TABLE customers_small CASCADE CONSTRAINTS;
DROP SEQUENCE customer_seq;
DROP SEQUENCE postcode_seq;
DROP SEQUENCE customersm_seq;
```

Quotes Table

```
REM create tables and data
REM create the table for quotes

REM drop tables
REM
****************************************************************
*

DROP SEQUENCE customer_seq;
DROP SEQUENCE amount_seq;
DROP SEQUENCE quotelg_seq;
DROP SEQUENCE quotesm_seq;
DROP TABLE quotes CASCADE CONSTRAINTS;
DROP TABLE quotes_temp CASCADE CONSTRAINTS;
```

```
DROP TABLE quotes_small CASCADE CONSTRAINTS;
DROP TABLE quotes_large CASCADE CONSTRAINTS;
DROP INDEX quote_ix;

REM create tables
REM
*****************************************************************
**

CREATE TABLE quotes_small (
quote_no      NUMBER(6) CONSTRAINT quotesm_pk primary key,
description   VARCHAR2(35),
amount        NUMBER(7),
cust_id       NUMBER(5) NOT NULL CONSTRAINT custsm_fk
              REFERENCES customers(cust_id))
TABLESPACE LARGE_TABLES
ENABLE PRIMARY KEY USING INDEX TABLESPACE USER_INDEXES;

CREATE TABLE quotes(
quote_no      NUMBER(6) CONSTRAINT quote_pk PRIMARY KEY,
description   VARCHAR2(35) NOT NULL,
amount        NUMBER(7) NOT NULL,
cust_id       NUMBER(5) NOT NULL CONSTRAINT cust_fk REFERENCES
              customers(cust_id))
TABLESPACE LARGE_TABLES
ENABLE PRIMARY KEY USING INDEX TABLESPACE USER_INDEXES;

CREATE TABLE quotes_temp(
quote_no      NUMBER(6) CONSTRAINT quotetm_pk PRIMARY KEY,
description   VARCHAR2(35),
amount        NUMBER(7),
cust_id       NUMBER(5)  NOT NULL CONSTRAINT custtm_fk
              REFERENCES customers(cust_id))
TABLESPACE LARGE_TABLES
ENABLE PRIMARY KEY USING INDEX TABLESPACE USER_INDEXES;

CREATE TABLE quotes_temp (
quote_no      NUMBER(6) CONSTRAINT quotelg_pk PRIMARY KEY,
description   VARCHAR2(35),
amount        NUMBER(7),
cust_id      NUMBER(5)  NOT NULL CONSTRAINT custlg_fk
             REFERENCES customers(cust_id))
TABLESPACE LARGE_TABLES
ENABLE PRIMARY KEY USING INDEX TABLESPACE USER_INDEXES;

REM populate tables
REM
*****************************************************************
```

```
**

--
-- Create the quotes_small table
--

CREATE SEQUENCE quotesm_seq START WITH 1;

INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Mowing the lawn and gardening',5000,6);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Vacuum Clean and dry four bedrooms',1000, 4);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Removing roof tiles with BBB tiles',5000, 9);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Adding a new 2GB hard disk',5000, 8);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Replacing motherboard with IntelP',200,3);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Evaluating the land value',600,10);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Installing air conditioning',5000,1);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Quality Review and Acceptance',700,5);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Servicing the car',1200,7);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Repairing the door lock',5000, 2);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Painting the House',900, 12);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Placing tiles and painting',5000,11);

COMMIT;

CREATE SEQUENCE customer_seq START WITH 13;

INSERT INTO quotes_small
  SELECT quotesm_seq.nextval,
       description,  amount, customer_seq.nextval
  FROM quotes_small;

COMMIT;

INSERT INTO quotes_small
  SELECT quotesm_seq.nextval,
       description,  amount,customer_seq.nextval
  FROM quotes_small;
```

```
COMMIT;

INSERT INTO quotes_small
  SELECT quotesm_seq.nextval,
       description, amount, customer_seq.nextval
  FROM quotes_small;

COMMIT;


INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Cleaning the back yard',5000,customer_seq.nextval);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Painting 3 bedrooms',2000, customer_seq.nextval);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Placing the fence and security',5000,
customer_seq.nextval);
INSERT INTO quotes_small VALUES
(quotesm_seq.nextval,'Repairing the garage lock',3000, customer_seq.nextval);

COMMIT;

INSERT INTO quotes_temp
  SELECT quote_no,
       description,  amount, cust_id
  FROM quotes_small;

COMMIT;

INSERT INTO quotes_small
  SELECT quotesm_seq.nextval,
       description, amount, customer_seq.nextval
  FROM quotes_small;

COMMIT;

INSERT INTO quotes_small
  SELECT quotesm_seq.nextval,
       description,  amount, customer_seq.nextval
  FROM quotes_small;

COMMIT;

INSERT INTO quotes_small
  SELECT quotesm_seq.nextval,
       description,  amount, customer_seq.nextval
  FROM quotes_small;
```

```
COMMIT;

INSERT INTO quotes_small
  SELECT quotesm_seq.nextval,
       description, amount, customer_seq.nextval
  FROM quotes_temp;

COMMIT;

INSERT INTO quotes_small
  SELECT quotesm_seq.nextval,
       description, amount, customer_seq.nextval
  FROM quotes_temp;

COMMIT;

DELTE quotes_temp WHERE quote_no IS NOT NULL;
COMMIT;

DROP SEQUENCE quotesm_seq;
DROP SEQUENCE customer_seq;
--
-- Create the quotes_large table
--

-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq1.lis');

CREATE SEQUENCE customer_seq START WITH with 1;

-- create random value of customer id for 1000 cusstomers
INSERT INTO quotes_temp
  SELECT quote_no,
       description, amount,
       readfile.get_row(customer_seq.nextval)
   FROM quotes_small;

COMMIT;

EXEC array.populate_array(10000);
EXEC sequence.get_next_sequence(10000);
CREATE SEQUENCE quotelg_seq START WITH 1;
DROP SEQUENCE customer_seq;
CREATE SEQUENCE customer_seq START WITH 1;

-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq1.lis');
```

```
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq2.lis');
CREATE SEQUENCE customer_seq START WITH 1;

-- 2000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq3.lis');
CREATE SEQUENCE customer_seq START WITH 1;

-- 3000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq4.lis');
CREATE SEQUENCE customer_seq STRAT WITH 1;

-- 4000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;
```

```
COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq5.lis');
CREATE SEQUENCE customer_seq START WITH 1;

-- 5000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq6.lis');
CREATE SEQUENCE customer_seq start with 1;

-- 6000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq7.lis');
CREATE SEQUENCE customer_seq start with 1;

-- 7000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq8.lis');
```

```
CREATE SEQUENCE customer_seq start with 1;

-- 8000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq9.lis');
CREATE SEQUENCE customer_seq start with 1;

-- 9000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;

DROP SEQUENCE customer_seq;
-- generate random numbers in an array
EXEC readfile.file_to_array('d:\script', 'ranq10.lis');
CREATE SEQUENCE customer_seq start with 1;

-- 10000 rows created
INSERT INTO quotes_large
  SELECT array.get_row(quotelg_seq.nextval) +1000,
       description, amount,
       readfile.get_row(customer_seq.nextval)
  FROM quotes_temp;

COMMIT;
-- Create the quote table with random amount number

EXEC array.populate_array(10000);
EXEC sequence.get_next_sequence(10000);
CREATE SEQUENCE amount_seq START WITH 1;
INSERT INTO quotes
  SELECT quote_no, description,
        get_amount(amount, amount_seq.nextval), cust_id
  FROM quotes_large;
```

```
CREATE INDEX QUOTE_IX ON quotes(cust_id) TABLESPACE
USER_INDEXES;

DROP TABLE quotes_small CASCADE CONSTRAINTS;
DROP TABLE quotes_temp CASCADE CONSTRAINTS;
DROP SEQUENCE customer_seq;
DROP SEQUENCE quotelg_seq;
DROP SEQUENCE amount_seq;
```

Quote Table

```
REM create tables and data
REM create the table for quotes

REM create tables
REM
***************************************************************
**

--
-- Create the quote table
--
DROP TABLE quote CASCADE CONSTRAINTS;
DROP SEQUENCE customer_seq;
DROP SEQUENCE quote_seq;

CREATE TABLE quote (
quote_no       NUMBER(6) CONSTRAINT quote1_pk PRIMARY KEY,
description    VARCHAR2(35) NOT NULL,
amount         NUMBER(7) NOT NULL,
cust_id        NUMBER(5) CONSTRAINT customer_fk REFERENCES
               customers(cust_id))
TABLESPACE LARGE_TABLES
ENABLE PRIMARY KEY USING INDEX TABLESPACE USER_INDEXES;

REM populate tables
REM
***************************************************************
*

CREATE SEQUENCE customer_seq START WITH 1;
CREATE SEQUENCE quote_seq START WITH 1;

INSERT INTO quote
  SELECT quote_no, description,  amount,
      null
  FROM quotes;
```

```
COMMIT;
EXEC set_quote;

CREATE INDEX QUO_IX ON quote(cust_id) TABLESPACE
USER_INDEXES;

DROP SEQUENCE customer_seq;
DROP SEQUENCE quote_seq;
```

APPENDIX C- Query Statements

Experiment 1 - One-to-many relationship

Nested Loop Join

```
ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
/
ALTER SESSION SET SQL_TRACE = TRUE;
ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
/
ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6051
/
ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6101
/
ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6201
/
 ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
```

```
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6301
/
 ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6401
/
ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6501
/
 ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6601
/
 ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6701
/
ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6801
/
 ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6901
/
ALTER SYSTEM FLUSH_SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
```

```
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 7001
/
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

Sort Merge Join

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6001
/
 ALTER SESSION SET SQL_TRACE = TRUE;
 ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6001
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6051
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6101
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6201
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
```

```
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6301
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6401
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6501
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6601
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6701
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6801
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6901
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
```

```
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 7001
/
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

Hash Join

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
/
 ALTER SESSION SET SQL_TRACE = TRUE;
 ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6051
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6101
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6201
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
```

```
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6301
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6401
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6501
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6601
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6701
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6801
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6901
/
ALTER SYSTEM FLUSH SHARED_POOL;
```

```
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 7001
/
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

Experiment 2 - One-to-one relationship

Nested Loop Join

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
/
ALTER SESSION SET SQL_TRACE = TRUE;
 ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
/
 ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6051
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6101
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6201
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6301
/
ALTER SESSION FLUSH SHARED_POOL;
```

```
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6401
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6501
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6601
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6701
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6801
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6901
/
ALTER SESSION FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 7001
/
```

```
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

Sort Merge Join

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM  quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6001
/
 ALTER SESSION SET SQL_TRACE = TRUE;
 ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6001
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6051
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6101
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6201
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6301
/
```

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6401
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6501
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6601
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6701
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6801
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 6901
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id+0 = quo.cust_id+0
 AND postcode < 7001
```

```
/
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

Hash Join

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
/
ALTER SESSION SET SQL_TRACE = TRUE;
 ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6051
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6101
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6201
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6301
```

```
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6401
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6501
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6601
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6701
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6801
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6901
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quote quo, customers cus
 WHERE cus.cust_id = quo.cust_id
```

```
 AND postcode < 7001
/
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

Experiment 3 - Predicate on Inner Table

Nested Loop Join

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SESSION SET SQL_TRACE = TRUE;
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6051
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6101
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6201
 AND amount < 50000  -- predicate aplied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
```

```
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6301
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6401
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6501
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6601
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6701
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6801
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
```

```
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6901
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 7001
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

Sort Merge Join

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
 AND amount < 50000  -- predicate applied to inner table
/
 ALTER SESSION SET SQL_TRACE = TRUE;
 ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6501
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
```

```
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6101
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6201
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6301
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6401
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6501
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6601
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
```

```
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6701
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6801
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6901
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 7001
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

Hash Join

```
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6001
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SESSION SET SQL_TRACE = TRUE;
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
```

```
 AND postcode < 6001
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6051
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6101
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6201
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6301
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6401
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
```

```
 AND postcode < 6501
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6601
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6701
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6801
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 6901
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SYSTEM FLUSH SHARED_POOL;
 SELECT /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 FROM quotes quo, customers cus
 WHERE cus.cust_id = quo.cust_id
 AND postcode < 7001
 AND amount < 50000  -- predicate applied to inner table
/
ALTER SESSION SET SQL_TRACE = FALSE;
QUIT
```

APPENDIX D –Unix Scripts

To extract the performance data from the text files

```
for file in `ls *.lis`
do
ex $file << EOF
g/*+/.,.13w! >> $file.out
EOF
done
for file in `ls *.out`
do
grep -E 'total' $file | $file.dat
done
```

To extract the required fields from the text files

```
for file in `ls *.new`
do
  awk '{print $3"\t"$4"\t"$6 }' $file > $file.dat
done
```

To display each random data on a single line.

```
for file in `ls *.lis`
do
  awk '{print $1"\n"$2"\n"$3"\n"$4"\n"$5"\n"$6"\n"$7"\n"$8"\n"$9"\n"$10 }'
$file > $file.dat
done
```

APPENDIX E – Trace Files Generated For Each Run

Outer Selectivity for a one to many relationship

Runs	Scripts	Generated Trace files
1	outsmain1.sql	ORA00132.trc
	outsmain3sql	ORA00063.trc
	outsmain2.sql	ORA00070.trc
2	outsmain2.sql	ORA00130.trc
	outsmain3sql	ORA00143.trc
	outsmain1ql	ORA00071.trc
3	outsmain3.sql	ORA00099.trc
	outsmain1.sql	ORA00104.trc
	outsmain2.sql	ORA00117.trc
4	outsmain1.sql	ORA00101.trc
	outsmain2.sql	ORA00107.trc
	outsmain3.sql	ORA00093.trc
5	outsmain3.sql	ORA00102.trc
	outsmain1.sql	ORA00075.trc
	outsmain2.sql	ORA00079.trc
6	outsmain2.sql	ORA00044.trc
	outsmain3.sql	ORA00042.trc
	outsmain1.sql	ORA00116.trc
7	outsmain3.sql	ORA00099a.trc
	outsmain2.sql	ORA00042a.trc
	outsmain1.sql	ORA00094.trc
8	outsmain1.sql	ORA00072.trc
	outsmain2.sql	ORA00101a.trc
	outsmain3.sql	ORA00072a.trc
9	outsmain2.sql	ORA00102a.trc
	outsmain3.sql	ORA00134.trc
	outsmain1.sql	ORA00074.trc
10	outsmain3.sql	ORA00134a.trc
	outsmain1.sql	ORA00097.trc
	outsmain2.sql	ORA00065.trc
11	outsmain1.sql	ORA00065a.trc
	outsmain2.sql	ORA00095.trc
	outsmain3.sql	ORA00068.trc
12	outsmain2.sql	ORA00103.trc
	outsmain3.sql	ORA00137.trc
	outsmain1.sql	ORA00127.trc
13	outsmain3.sql	ORA00103a.trc
	outsmain2.sql	ORA00079a.trc
	outsmain1.sql	ORA00044a.trc
14	outsmain1.sql	ORA00141.trc

	outsmain3.sql	ORA00121.trc
	outsmain2.sql	ORA00117.trc
15	outsmain2.sql	ORA00103b.trc
	outsmain3.sql	ORA00061.trc
	outsmain1.sql	ORA00139.trc
16	outsmain2.sql	
	outsmain3.sql	
	outsmain1.sql	
17	outsmain3.sql	
	outsmain2.sql	
	outsmain1.sql	

Outer Selectivity for a one to one relationship

Runs	Scripts	Generated Trace files
Trial	Script run	Trace file generated
1	outmain1.sql	ORA00119.trc
	outmain3.sql	ORA00126.trc
	outmain2.sql	ORA00044.trc
2	outmain2.sql	ORA00044a.trc
	outmain3sql	ORA00042.trc
	outmain1.sql	ORA00119a.trc
3	outmain3.sql	ORA00127.trc
	outmain1.sql	ORA00080.trc
	outmain2.sql	ORA00083.trc
4	outmain1.sql	ORA00057.trc
	outmain2.sql	ORA00063.trc
	outmain3.sql	ORA00107.trc
5	outmain3.sql	ORA00044b.trc
	outmain1.sql	ORA00100.trc
	outmain2.sql	ORA00112.trc
6	outmain2.sql	ORA00037.trc
	outmain3.sql	ORA00070.trc
	outmain1.sql	ORA00065.trc
7	outmain3.sql	ORA00081.trc
	outmain2.sql	ORA00081a.trc
	outmain1.sql	ORA00107a.trc
8	outmain1.sql	ORA00089.trc
	outmain2.sql	ORA00042a.trc
	outmain3.sql	ORA00098.trc
9	outmain2.sql	ORA00136.trc
	outmain3.sql	ORA00100a.trc
	outmain1.sql	ORA00095.trc
10	outmain3.sql	ORA00130.trc
	outmain1.sql	ORA00138.trc

	outmain2.sql	ORA00074a.trc
11	outmain1.sql	ORA00129.trc
	outmain2.sql	ORA00081b.trc
	outmain3.sql	ORA00071.trc
12	outmain2.sql	ORA00118.trc
	outmain3.sql	ORA00139a.trc
	outmain1.sql	ORA00095.trc
13	outmain3.sql	ORA00061.trc
	outmain2.sql	ORA00044.trc
	outmain1.sql	ORA00135.trc
14	outmain1.sql	ORA00074.trc
	outmain3.sql	ORA00128.trc
	outmain2.sql	ORA00093.trc
15	outmain2.sql	ORA00093a.trc
	outmain3.sql	ORA00126a.trc
	outmain1.sql	ORA00126b.trc

Outer Selectivity with a filter criteria on inner table for a one to one relationship

Runs	Scripts	Generated Trace files
1	iomain1.sql	ORA00107.trc
	iomain3.sql	ORA00120.trc
	iomain2.sql	ORA00138.trc
2	iomain2.sql	ORA00073.trc
	iomain3sql	ORA00092.trc
	iomain1.sql	ORA00093.trc
3	iomain3.sql	ORA00134.trc
	iomain1.sql	ORA00135.trc
	iomain2.sql	ORA00191.trc
4	iomain1.sql	ORA00057.trc
	iomain2.sql	ORA00065.trc
	iomain3.sql	ORA00044.trc
5	iomain3.sql	ORA00065a.trc
	iomain1.sql	ORA00065b.trc
	iomain2.sql	ORA00123.trc
6	iomain2.sql	ORA00123a.trc
	iomain3.sql	ORA00123b.trc
	iomain1.sql	ORA00068.trc
7	iomain3.sql	ORA00108.trc
	iomain2.sql	ORA00126.trc
	iomain1.sql	ORA00112.trc
8	iomain1.sql	ORA00044.trc
	iomain2.sql	ORA00099.trc
	iomain3.sql	ORA00099a.trc
9	iomain2.sql	ORA00097.trc

	iomain3.sql	ORA00105.trc
	iomain1.sql	ORA00105a.trc
10	iomain3.sql	ORA00116.trc
	iomain1.sql	ORA00136.trc
	iomain2.sql	ORA00109.trc
11	iomain1.sql	ORA00121.trc
	iomain2.sql	ORA00123c.trc
	iomain3.sql	ORA00124.trc
12	iomain2.sql	ORA00115.trc
	iomain3.sql	ORA00037.trc
	iomain1.sql	ORA00074.trc
13	iomain3.sql	ORA00118.trc
	iomain2.sql	ORA00075.trc
	iomain1.sql	ORA00102.trc
14	iomain1.sql	ORA00057a.trc
	iomain3.sql	ORA00057b.trc
	iomain2.sql	ORA00142.trc
15	iomain2.sql	ORA00066.trc
	iomain3.sql	ORA00095.trc
	iomain1.sql	ORA00064.trc

APPENDIX F - Example of Random Numbers Generated

437	920	173	750	615	665	651	178	465	937
560	893	753	384	165	848	184	985	88	943
987	258	824	556	302	804	392	706	573	544
815	415	202	549	455	318	157	483	420	591
366	957	505	662	980	332	448	393	889	825
452	628	135	811	206	229	13	852	918	191
973	763	48	279	776	710	794	297	418	215
481	65	690	512	400	540	526	693	837	786
479	514	170	671	941	867	494	484	908	196
934	922	522	511	704	27	630	622	703	130
8	802	910	878	147	99	111	718	692	854
679	712	809	193	334	360	249	642	212	818
843	19	616	672	542	234	336	320	493	849
329	473	548	259	168	245	243	51	640	79
305	793	251	319	1	339	674	226	971	461
274	678	435	291	304	958	871	948	407	300
381	261	492	180	698	697	132	353	221	519
122	219	709	68	688	216	430	744	545	959
991	716	993	463	84	850	447	944	829	223
247	546	561	81	602	311	92	269	52	324
110	475	869	439	733	816	21	458	758	530
949	445	218	839	42	550	947	317	658	543
284	262	90	62	553	862	751	112	433	240
571	232	707	741	266	735	755	739	174	557
891	142	539	834	619	390	536	199	929	177
676	350	624	799	56	365	136	436	645	55
401	953	689	123	532	945	903	790	647	113
996	442	6	395	609	406	107	3	936	740
725	761	382	667	727	150	158	371	423	34
649	820	708	795	868	1000	928	562	472	28
833	842	144	576	568	372	77	713	675	351
108	327	778	140	827	506	995	246	308	487
724	161	770	238	54	260	896	779	978	880
845	555	194	330	263	789	272	528	524	438
290	231	333	156	306	326	129	587	358	569
289	559	620	691	629	417	200	925	986	357
518	15	976	164	187	626	819	419	97	627
148	551	956	499	227	368	298	69	659	286
248	128	800	863	984	784	517	632	912	421
743	598	171	935	12	408	766	316	175	235
321	759	835	117	664	66	873	2	503	17
823	764	625	646	385	138	950	961	362	343
926	478	116	287	345	411	211	474	355	599
652	477	409	338	14	558	467	975	653	89
432	593	197	807	963	280	547	654	895	38
723	225	43	900	554	185	124	41	10	281
951	198	373	812	145	114	870	310	397	288
422	656	374	47	581	938	217	872	612	998
490	205	782	195	67	575	749	22	500	282
921	470	36	501	388	370	611	201	121	151
513	169	377	592	347	537	762	414	924	754
855	314	660	965	404	344	830	413	886	434
64	380	682	349	650	363	661	746	45	981
244	39	877	901	95	771	154	914	781	100
74	883	386	141	582	93	50	915	102	496
527	9	340	443	44	955	267	309	983	960
76	186	271	98	605	777	655	49	459	997
798	722	222	813	239	572	700	803	881	464
814	897	586	378	488	337	57	20	952	416
120	552	538	888	864	694	757	182	346	252
495	352	87	535	402	643	356	931	756	529
637	633	773	853	394	990	482	861	797	988
257	277	695	398	497	489	403	315	515	657

972	75	410	361	105	601	131	32	531	256
737	53	720	325	866	613	686	383	60	607
565	982	264	181	153	930	322	696	831	801
946	699	734	254	70	292	451	648	594	295
450	905	882	590	146	717	994	507	578	917
772	902	638	860	574	913	954	24	702	521
765	29	250	933	125	454	127	203	748	115
241	242	715	687	729	510	683	7	71	964
516	35	840	58	666	162	207	859	364	969
480	424	471	204	631	143	155	230	5	16
909	209	331	342	808	884	26	680	608	806
25	525	468	59	328	101	275	149	874	970
577	564	856	23	851	714	190	94	134	192
846	446	942	736	412	563	104	265	858	167
875	774	788	728	670	236	30	906	296	747
644	787	584	887	273	431	462	31	176	635
780	821	600	979	968	228	596	405	857	923
533	444	103	817	323	237	792	810	508	214
641	614	210	429	389	731	376	916	449	992
769	721	188	588	911	80	270	966	589	106
255	617	606	285	579	83	604	805	927	4
96	580	486	126	719	585	847	166	159	293
623	567	276	399	932	892	359	904	224	898
534	428	301	387	940	760	109	885	610	268
520	967	894	335	348	822	502	726	768	832
440	457	876	785	711	838	118	46	509	426
701	133	618	907	732	639	313	299	745	307
453	391	738	791	668	583	396	730	72	595
541	962	742	63	354	685	974	213	603	40
783	491	705	523	498	989	681	82	767	233
369	826	312	796	375	341	634	752	469	152
673	18	61	160	977	139	504	253	208	879
621	86	379	441	78	636	841	119	844	939
91	899	684	73	137	303	11	163	183	220
294	456	466	427	172	663	33	278	425	865
85	775	485	836	919	189	566	476	367	37
999	179	828	597	283	677	570	890	460	669

APPENDIX G – Example of Generated Trace Files

```
TKPROF: Release 7.3.2.2.0 - Production on Wed Oct 29 21:30:14 1997

Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

Trace file: c:\amallet\trace\ora00065.trc
Sort options: default

********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************

alter session set sql_trace = true

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0      0.00       0.00          0          0          0           0
Execute      1      0.05       0.09          7         30          1           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        1      0.05       0.09          7         30          1           0

Misses in library cache during parse: 0
Misses in library cache during execute: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)
********************************************************************************

alter system flush shared_pool


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       12      0.21       0.28          0          0          0           0
Execute     12      0.50       0.52          0          0          0           0
```

```
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       24      0.71       0.80          0          0          0          0

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)
********************************************************************************

 select /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 from  quotes quo, customers cus
 where cus.cust_id+0 = quo.cust_id+0
 and postcode < 6001

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.61       0.78          0          0          2          0
Execute      2      0.39       0.52          0          0          1          0
Fetch        1      3.07       4.15        270        270        247          0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      4.07       5.45        270        270        250          0

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      0   MERGE JOIN
  10000    SORT (JOIN)
  10000     TABLE ACCESS (FULL) OF 'QUOTES'
      0    SORT (JOIN)
   1000     TABLE ACCESS (FULL) OF 'CUSTOMERS'

********************************************************************************

select /*+ USE_MERGE(QUO, CUS) */
 cus.name, quo.quote_no
 from  quotes quo, customers cus
 where cus.cust_id+0 = quo.cust_id+0
```

```
 and postcode < 6101

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.40       0.46          0          0          0           0
Execute      2      0.50       0.55          0          0          1           0
Fetch       67      5.18       7.78        289        270        326        1000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       70      6.08       8.79        289        270        327        1000

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)
```

```
TKPROF: Release 7.3.2.2.0 - Production on Wed Oct 29 14:50:05 1997

Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

Trace file: c:\amallet\trace\ora00094.trc
Sort options: default

********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************
alter session set sql_trace = true

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0      0.00       0.00          0          0          0           0
Execute      1      0.05       0.09          7         30          1           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        1      0.05       0.09          7         30          1           0

Misses in library cache during parse: 0
Misses in library cache during execute: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)
********************************************************************************

alter system flush shared_pool


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       12      0.21       0.28          0          0          0           0
Execute     12      0.50       0.52          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       24      0.71       0.80          0          0          0           0
```

```
Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)
********************************************************************************

 select /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 from  quotes quo, customers cus
 where cus.cust_id = quo.cust_id
 and postcode < 6001

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.70       0.84          0          0          2           0
Execute      1      0.02       0.01          0          0          0           0
Fetch        1      0.03       0.07          6         15          2           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.75       0.92          6         15          4           0

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      0   NESTED LOOPS
   1000    TABLE ACCESS (FULL) OF 'CUSTOMERS'
      0    TABLE ACCESS (BY ROWID) OF 'QUOTES'
      0     INDEX (RANGE SCAN) OF 'QUOTE_IX' (NON-UNIQUE)

********************************************************************************

select /*+ USE_INDEX(QUO, CUS) */
 cus.name, quo.quote_no
 from  quotes quo, customers cus
 where cus.cust_id = quo.cust_id
 and postcode < 6101

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
```

```
Parse        1      0.41       0.47         0          0          0           0
Execute      1      0.00       0.00         0          0          0           0
Fetch       67      1.61       8.24       578       2315          2        1000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       69      2.02       8.71       578       2315          2        1000

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
   1000   NESTED LOOPS
   1000    TABLE ACCESS (FULL) OF 'CUSTOMERS'
   1000    TABLE ACCESS (BY ROWID) OF 'QUOTES'
   1100     INDEX (RANGE SCAN) OF 'QUOTE_IX' (NON-UNIQUE)
```

```
TKPROF: Release 7.3.2.2.0 - Production on Thu Oct 30 10:07:56 1997

Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

Trace file: c:\amallet\trace\ora00137.trc
Sort options: default

********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************

alter session set sql_trace = true
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0      0.00       0.00          0          0          0           0
Execute      1      0.07       0.10          0         30          1           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        1      0.07       0.10          0         30          1           0

Misses in library cache during parse: 0
Misses in library cache during execute: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)
********************************************************************************

alter system flush shared_pool

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       12      0.22       0.26          0          0          0           0
Execute     12      0.35       0.39          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       24      0.57       0.65          0          0          0           0
```

```
Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)
********************************************************************************

 select /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 from  quotes quo, customers cus
 where cus.cust_id = quo.cust_id
 and postcode < 6001

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.65       0.77          0          0          2           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.05       0.07          6         15          2           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.70       0.84          6         15          4           0

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      0   HASH JOIN
   1000    TABLE ACCESS (FULL) OF 'CUSTOMERS'
      0    TABLE ACCESS (FULL) OF 'QUOTES'

********************************************************************************


 select /*+ USE_HASH(QUO, CUS) */
 cus.name, quo.quote_no
 from  quotes quo, customers cus
 where cus.cust_id = quo.cust_id
 and postcode < 6101

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.40       0.46          0          0          0           0
```

```
Execute      1      0.01       0.01          0          0          0           0
Fetch       67      0.94       1.61        270        336          4        1000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       69      1.35       2.08        270        336          4        1000

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 14  (ADA)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
   2510   HASH JOIN
   1000    TABLE ACCESS (FULL) OF 'CUSTOMERS'
  10000    TABLE ACCESS (FULL) OF 'QUOTES'

********************************************************************************
```

APPENDIX H – Performance Data Collected

Table 14: Response Time v/s Join Selectivity Factor for a one-to-many relationship

	Join Method		
Join Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	0.81	5.42	0.81
1×10^{-4}	8.57	7.93	2.14
2×10^{-4}	14.59	7.75	2.38
3×10^{-4}	21.13	7.94	2.83
4×10^{-4}	27.63	8.54	3.34
5×10^{-4}	33.56	9.04	3.73
6×10^{-4}	40.07	9.42	4.07
7×10^{-4}	46.72	9.77	4.5
8×10^{-4}	54.47	10.13	4.86
9×10^{-4}	61.5	10.54	5.26
10×10^{-4}	65.85	11.18	5.7

Table 15: CPU Time v/s Join Selectivity Factor for a one-to-many relationship

	Join Method		
Join Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	0.69	4.25	0.67
1×10^{-4}	1.98	6.07	1.34
2×10^{-4}	3.47	6.47	1.7
3×10^{-4}	4.91	6.81	2.12
4×10^{-4}	6.51	7.25	2.57
5×10^{-4}	7.95	7.75	3
6×10^{-4}	9.47	8.16	3.39
7×10^{-4}	10.82	8.66	3.79
8×10^{-4}	12.47	9.04	4.16
9×10^{-4}	14.05	9.45	4.58
10×10^{-4}	15.4	9.91	5.02

Table 16: Number of I/O reads v/s Join Selectivity Factor for a one-to-many relationship

	Join Method		
Join Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	15	270	15
1×10^{-4}	2315	270	336
2×10^{-4}	4615	270	403
3×10^{-4}	6914	270	470
4×10^{-4}	9214	270	536
5×10^{-4}	11514	270	603
6×10^{-4}	13814	270	670
7×10^{-4}	16114	270	736
8×10^{-4}	18414	270	803
9×10^{-4}	20714	270	870
10×10^{-4}	23014	270	936

Table 17: Response Time v/s Join Selectivity Factor for a one-to-one relationship

	Join Method		
Join Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	0.78	4.59	0.82
1×10^{-5}	1.06	4.58	1.29
2×10^{-5}	1.37	5.09	1.41
3×10^{-5}	1.06	4.89	1.54
4×10^{-5}	3.31	4.58	1.68
5×10^{-5}	2.89	4.78	1.76
6×10^{-5}	3.26	4.94	1.74
7×10^{-5}	2.91	4.95	1.76
8×10^{-5}	4.52	4.84	1.9
9×10^{-5}	4.27	5.11	1.91
10×10^{-5}	4.58	5.14	1.89

Table 18: CPU Time v/s Join Selectivity Factor for a one-to-one relationship

	Join Method		
Join Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	0.66	3.49	0.68
1×10^{-5}	0.6	3.4	0.82
2×10^{-5}	0.77	3.51	0.87
3×10^{-5}	0.56	3.56	0.93
4×10^{-5}	1.26	3.61	0.97
5×10^{-5}	1.02	3.66	1.04
6×10^{-5}	1.53	3.72	1.07
7×10^{-5}	1.28	3.8	1.14
8×10^{-5}	1.87	3.86	1.17
9×10^{-5}	1.68	3.92	1.23
10×10^{-5}	2.14	3.95	1.25

Table 19: Number of I/O reads v/s Join Selectivity Factor for a one-to-one relationship

	Join Method		
Join Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	15	250	15
1×10^{-5}	515	250	256
2×10^{-5}	1015	250	263
3×10^{-5}	1514	250	270
4×10^{-5}	2014	250	276
5×10^{-5}	2514	250	283
6×10^{-5}	3014	250	290
7×10^{-5}	3514	250	296
8×10^{-5}	4014	250	303
9×10^{-5}	4514	250	310
10×10^{-5}	5014	250	316

Table 20: Response Time v/s Outer Selectivity Factor for a one-to-many relationship with a predicate on inner table

	Join Method		
Outer Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	4.62	3.96	1.86
1×10^{-1}	7.35	4.87	1.78
2×10^{-1}	14.37	4.96	2.26
3×10^{-1}	20.83	5.14	2.13
4×10^{-1}	27.05	5.33	2.43
5×10^{-1}	33.25	6	2.53
6×10^{-1}	39.32	5.9	2.69
7×10^{-1}	45.24	6.2	2.91
8×10^{-1}	53.02	6.18	3.05
9×10^{-1}	60.25	6.44	3.3
1	65.69	6.63	3.47

Table 21: CPU Time v/s Outer Selectivity Factor for a one-to-many relationship with a predicate on inner table

	Join Method		
Outer Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	0.63	3.04	1.34
1×10^{-1}	1.73	3.74	1.25
2×10^{-1}	3.09	3.94	1.48
3×10^{-1}	4.41	4.21	1.68
4×10^{-1}	5.78	4.44	1.87
5×10^{-1}	6.97	4.62	2.04
6×10^{-1}	8.36	4.86	2.16
7×10^{-1}	9.59	5.08	2.41
8×10^{-1}	10.89	5.31	2.54
9×10^{-1}	12.28	5.47	2.76
1	13.63	5.72	2.93

Table 22: Number of I/O reads v/s Outer Selectivity Factor for a one-to-many relationship with a predicate on inner table

	Join Method		
Outer Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	15	270	270
1×10^{-1}	2315	270	305
2×10^{-1}	4615	270	338
3×10^{-1}	6914	270	374
4×10^{-1}	9214	270	409
5×10^{-1}	11514	270	441
6×10^{-1}	13814	270	472
7×10^{-1}	16114	270	506
8×10^{-1}	18414	270	537
9×10^{-1}	20714	270	572
1	23014	270	603

Table 23: Response Time v/s Outer Selectivity Factor for a one-to-many relationship with no predicate on inner table

	Join Method		
Outer Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	0.81	5.42	0.81
1×10^{-1}	8.57	7.93	2.14
2×10^{-1}	14.59	7.75	2.38
3×10^{-1}	21.13	7.94	2.83
4×10^{-1}	27.63	8.54	3.34
5×10^{-1}	33.56	9.04	3.73
6×10^{-1}	40.07	9.42	4.07
7×10^{-1}	46.72	9.77	4.5
8×10^{-1}	54.47	10.13	4.86
9×10^{-1}	61.5	10.54	5.26
1	65.85	11.18	5.7

Table 24: CPU Time v/s Outer Selectivity Factor for a one-to-many relationship with no predicate on inner table

	Join Method		
Outer Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	0.69	4.25	0.67
1×10^{-1}	1.98	6.07	1.34
2×10^{-1}	3.47	6.47	1.7
3×10^{-1}	4.91	6.81	2.12
4×10^{-1}	6.51	7.25	2.57
5×10^{-1}	7.95	7.75	3
6×10^{-1}	9.47	8.16	3.39
7×10^{-1}	10.82	8.66	3.79
8×10^{-1}	12.47	9.04	4.16
9×10^{-1}	14.05	9.45	4.58
1	15.4	9.91	5.02

Table 25: Number of I/O reads v/s Outer Selectivity Factor for a one-to-many relationship with no predicate on inner table

	Join Method		
Outer Selectivity Factor	Nested Loop	Sort Merge	Hash Join
0	15	270	15
1×10^{-1}	2315	270	336
2×10^{-1}	4615	270	403
3×10^{-1}	6914	270	470
4×10^{-1}	9214	270	536
5×10^{-1}	11514	270	603
6×10^{-1}	13814	270	670
7×10^{-1}	16114	270	736
8×10^{-1}	18414	270	803
9×10^{-1}	20714	270	870
1	23014	270	936

References

Aronoff, E., Loney, K., & Sonawalla, N. (1997). *Advanced ORACLE tuning and administration - Making your database perform at peak.* New York: Osborne McGraw-Hill.

Atzeni, P., & De Antonellis, V. (1993). *Relational database theory.* Redwood City, CA: Benjamin/Cummings.

Bennett, K., Ferris, M.C., & Ioannidis, Y. E. (1991). *A genetic algorithm for database query optimization.* Madison, Wisconsin: University of Wisconsin, Computer Sciences Department.

Cheng, J., Haderle, D., Hedges, R., Iyer, B. R., Messinger, T., Mohan, C., & Wang, Y. (1991). An efficient hybrid join algorithm: A DB2 prototype, *Seventh international conference on data engineering* (pp. 171-180). Los Alamitos, USA: IBM.

Codd, E.F. (1970). A relational model for large shared data banks. *Communications of the ACM, 13* (6), 377-387.

Codd, E. F. (1990). *The relational model for database management - version 2.* Reading, MA: Addison-Wesley.

Corey, M. J., Abbey, M., & Dechichio, D. J., Jr. (1995). *Tuning oracle.* Berkeley, CA: Oracle Press/Osborne McGraw-Hill.

Corrigan, P., & Gurry, M. (1996). *ORACLE performance tuning* (2nd rev ed.). Sebastopol, CA: O'Reilly & Associates.

Database Market to top $10.1 billion (1997). [On-line]. Available: http://techweb1.web.cerf.net/wire/news/apr/0410database.html.body [1997, 30 April].

Date, C. J. (1986). *Relational database - selected writings.* Sydney, Australia: Addison-Wesley.

Date, C. J. (1989). *A guide to the SQL standard* (2nd ed.). Reading, MA: Addison-Wesley.

Deitel, H. M. (1990). *An introduction to operating systems* (2nd ed.). Reading, MA: Addison-Wesley.

Gaede, V., & Gunther, O. (1994). *Processing joins with user-defined functions.* Berkeley: International Computer Science Institute.

Gardarin, G., & Valduriez, P. (1989). *Relational databases and knowledge bases.* Sydney: Addison-Wesley.

Graefe, G. (1994). *Sort-Merge-Join: An idea whose time has(h) passed?* Available: http://www.cse.ogi.eduIDISC/projects/ereg/papers/graefe-papers.html [1997, October 17].

Graefe, G., Linville, A., & Shapiro, L. D. (1994). Sort vs. Hash revisited. *IEEE Transactions on knowledge and data engineering, 6* (6), 934-944.

Harris, E. P. (1995). *Towards optimal storage design for efficient query processing in relational database systems.* Unpublished doctoral dissertation, University of Melbourne, Victoria, Australia.

Harris, E. P., & Ramamohanarao, K. (1996). Join algorithm costs revisited. *The VLDB Journal, 5*, 64-84.

Jarke, M., & Koch, J. (1984). Query optimization in database systems. *Computing Surveys, 16* (2), 111-152.

Jarke, M., Koch, J., & Schmidt, J. W. (1985). Introduction to query processing. In W. Kim, D. S. Reiner, & D. S. Batory (Eds.), *Query processing in* *database systems* (pp. 3-28). Berlin, Germany: Springer-Verlag.

Kim, W. (1980). A new way to compute the product and join of relations, *Proceedings of the 1980 ACM SIGMOD International Conference on the Management of Data* (pp 179-187).

Kuznetsov, S.D. (1989). Logical query optimization in relational database management systems. *Programming and Computer Software, 15* (6), 271-281.

Li, Y., Kitagawa, H., & Ohbo, N. (1994). Optimization of join-type queries in nested relational databases, *7* (6), 648-659.

Lipton, R.J., & Naughton, J.F. (1990). Query size estimation by adaptive sampling. *SIGMOD Record, 19* (2), 40-46.

Lipton, R.J., Naughton, J.F., & Schneider, D.A. (1990). Practical selectivity estimation through adaptive sampling. *SIGMOD Record, 19* (2), 1-11.

Lu, H., & Carey, M.J. (1985). Some experimental results on distributed join algorithms in a local network, *Proceedings of Very Large Data Base 85* (pp. 292-304). Stockholm: Very Large Data Base Endowment.

March, S., & Carlis, J. (1985). Physical database design: Techniques for improved database performance. In W. Kim, D. S. Reiner, & D. S. Batory (Eds.), *Query processing in database systems* (pp. 279-296). Berlin, Germany: Springer-Verlag.

McFadden, F. R., & Hoffer, J. A. (1991). *Database management* (3rd ed.). Redwood City, CA: Benjamin/Cummings.

Meechan, D. J. (1988). *A heuristic approach to query optimization.* Unpublished masteral dissertation, University of Alberta, Alberta, Canada.

Mishra, P. & Eich, M. (1992). Join processing in relational databases. *ACM Computing Surveys, 24*(1), 63-113.

Pascal, F. (1993). *Understanding relational databases -with examples in SQL -92.* New York: John Wiley.

Piatetsky-Shapiro, G., & Connell, C. (1984). Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Record*, 256-216.

Roti, S. (1996). *Indexing and database mechanisms* [On-line]. Available: http://www.dbmsmag.com/9605d15.html [1997, 6 April].

Stanczyk, S. (1991). *Programming in SQL.* London: Pitman.

Topor, R. (n. d.). *Language and Information: Communicating with databases - An inaugural lecture.* (Available from School of Computing and Information Technology, Faculty of Science and Technology, Griffith University, Queensland 4111, Australia).

Urman, S. (1996). *Oracle PL/SQL programming*. Berkeley, CA: Osborne McGraw-Hill.

Weaver, W. & Raulin, M. (1994). Random Number Generator Program [on-line]. Available WWW: http://www.buffalo.edu/~raulin/random.html. [1997, 29 November].

Yu, P. S., & Cornell, D. W. (1991). Optimal buffer allocation in a multi-query environment, *Proceedings 7th International Conference on data engineering* (pp. 622-629).