1992

Optimizing query performance in relational database systems

Anthony Casamassa
Lehigh University

AUTHOR:

Casamassa, Anthony

TITLE:

Optimizing Query Performance in Relational Database Systems

DATE: May 31, 1992

OPTIMIZING QUERY PERFORMANCE IN RELATIONAL DATABASE SYSTEMS

by

Anthony Casamassa

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

March 1992

This thesis is accepted and approved in partial fulfillment of the requirements for the Degree of Master of Science.

March 25, 1992
Date

Professor Donald Hillman
Thesis Advisor

Chairman of Department

TABLE OF CONTENTS

ABSTRACT

The issue of query performance is important as relational database management systems become larger. Query optimization of relational database systems will be discussed. Three aspects of query optimization will be examined. The first will be query optimization through database design. Query performance should be considered when designing relational database systems. Normalized and denormalized design will be examined as they relate to query performance. The second aspect of query optimization that will be explored is utilizing indexes to enhance query performance. The proper use of indexes can greatly enhance the performance of a query. Under certain circumstances the use of indexes can reduce the performance of queries. General rules about indexing will be examined. The effect of indexing on the overall database performance will be explored. The third aspect of query performance that will be examined is controlling the optimizer of the database system to improve query performance. The optimizer determines how a query will be executed based on heuristic rules. It is important to understand how the optimizer functions in order to achieve optimal query performance.

QUERY OPTIMIZATION THROUGH DATABASE DESIGN

INTRODUCTION:

The relational data model is based on mathematical principles which include set theory, the theory of relations, and first order predicate logic. It was introduced by E.F. Codd in 1970. The main goals of the relational data model are data independence and data integrity.[1] Data independence is achieved by separating the physical format of the data from the view that the user has of that data. Simplification through design avoids data inconsistencies and anomalies which leads to data integrity. Data objects and the relationships that exist between them are defined conceptually by the relational model. A relation is the basic structure of a relational system and can be defined as a two dimensional matrix consisting of a time varying set of tuples where each tuple consists of a set of attributes.[1] A table can be defined as a instance of a relation in the database system.[1] A table consists of rows and columns of data elements.

Relations possess the following properties: there are no duplicate tuples, tuples are unordered, and attributes are unordered. A normalized relation has the additional property that all attribute values are atomic. Such a relation is said

to be in first normal form. A table is in first normal form if at every row and column position within the table, there exists exactly one value, never a list of values or repeating groups.[1] A relation is in second normal form if it is in first normal form and every nonkey attribute is fully dependent on the primary key.[1] A nonkey attribute is fully dependent on a primary key when the nonkey attribute can only be identified by the primary key. A relation is in third normal form if it is in second normal form and each tuple consists of a primary key together with a set of mutually independent attribute values; no non-primary key column can be functionally dependent on another non-primary-key column.[1] Normalization is the technique by which the correct location for each attribute and correct structure for the relation is identified.[1] Normalization eliminates insert, delete, and update anomalies and reduces redundancy. Normalization also reduces query performance under most circumstances by requiring join operations on tables in order to retrieve data. The join operations increase I/O time and cpu time which reduces the performance of the query. Performance can be enhanced through the placement of duplicated data in tables or the combination of many small tables into larger ones. This process is called denormalization.

DENORMALIZATION:

Tables which are joined frequently are prime candidates for denormalization. The join process increases I/O time, which is the most expensive computer resource because it operates at mechanical rather than electrical speed.[2] It should be noted that although denormalization can increase performance, a tradeoff exists between performance and the negative effects of denormalization. Redundant data can increase the time required for update, delete, or insert procedures. Redundant data can also lead to update, insert, and delete anomalies and an increase in memory and disk storage requirements. During the design phase all tradeoffs should be considered before denormalizing.

The following examples illustrate how query performance can be enhanced through denormalization. In the first example STUDENT_INFO AND TEST_INFO are normalized tables which will be joined in a select statement to retrieve student and test information about a particular student. The STUDENT_INFO table consists of a student id, first name, and last name field. The TEST_INFO table consists of a student id, score date, and score result field. There are two test scores for each student so that the TEST_INFO table contains two rows for each corresponding row in the STUDENT_INFO table. The STUDENT_INFO table contains 8791 rows. The TEST_INFO table contains 17582 rows. The tables are structured as follows:

STUDENT_INFO TABLE STRUCTURE

Name	Null?	Type
ID	NOT NULL	NUMBER(8)
FNAME		CHAR(15)
LNAME	NOT NULL	CHAR(60)

TEST_INFO TABLE STRUCTURE

Name	Null?	Type
ID	NOT NULL	NUMBER(8)
SCORE_DATE	NOT NULL	DATE
SCORE	NOT NULL	CHAR(5)

The query is structured as follows:

```
select student_info.id,
       student_info.fname,
       student_info.lname,
       test_info.score,
       test_info.score_date
from student_info,test_info
where student_info.lname = 'JOHNSON'
      and student_info.id = test_info.id
```

The results of the query yields two rows:

```
    ID      FNAME    LNAME          SCORE   SCORE_DAT
```

99999	JOHN	JOHNSON	920	12-JAN-91
99999	JOHN	JOHNSON	990	25-MAY-91

The performance information for the query is as follows:

```
================================================================

        count      cpu    elap     phys       cr      cur     rows
Parse:      1        0       0        0        0        0        0
Execute:    1      341     378      168      306      372        0
Fetch:      1      372     389      123      185      213        2

Execution plan:
MERGE JOIN
  SORT (JOIN)
    TABLE ACCESS (FULL) OF 'TEST_INFO'
  SORT (JOIN)
    TABLE ACCESS (FULL) OF 'STUDENT_INFO'

================================================================
```

The above performance information indicates that the number of data blocks read from disk during the execute and fetch steps total 291. The elapsed time in hundredths of seconds for all steps totals 767. The performance of this query can be

enhanced by denormalizing the STUDENT_INFO and TEST_INFO tables by adding repeating groups of score results and score date information to the STUDENT_INFO table. This eliminates the need for the join to the TEST_INFO table. The new structure of the STUDENT_INFO table violates first normal form because of the repeating groups of score information.

STUDENT_INFO2 TABLE STRUCTURE

Name	Null?	Type
ID	NOT NULL	NUMBER(8)
FNAME		CHAR(15)
LNAME	NOT NULL	CHAR(60)
SCORE1_DATE	NOT NULL	DATE
SCORE1	NOT NULL	CHAR(5)
SCORE2_DATE	NOT NULL	DATE
SCORE2	NOT NULL	CHAR(5)

The query is structured as follows:

```
 select id,
        fname,
        lname,
        score1,
        score1_date,
        score2,
        score2_date
```

```
from student_info2
where lname= 'JOHNSON';
```

The result of the query yields one row:

```
ID    FNAME  LNAME    SCORE1  SCORE1_DATE  SCORE2 SCORE2_DATE
----- ------ -------- ------  -----------  ------ ----------
99999 JOHN   JOHNSON  920     12-JAN-91    990    25-MAY-91
```

The performance information for the query is as follows:

```
==========================================================

        count      cpu    elap     phys       cr      cur      rows
Parse:      1        1       1        0        0        0
Execute:    1        0       0        1        0        2         0
Fetch:      1       53      54      142      142        0         1

Execution plan:
TABLE ACCESS (FULL) OF 'STUDENT_INFO2'

==========================================================
```

The above performance information indicates that the number of data blocks read from disk during the execute and fetch steps totals 143. The elapsed time in hundredths of seconds for all steps totals 55. Performance was enhanced by eliminating the join in the first query. This required adding test

information in repeating groups to the STUDENT_INFO table. This violated first normal form but increased performance. The negative side effect of denormalization of this type is that the addition of other test information would require modification to the structure the table.

The next example begins with two normalized tables which will be joined in order to retrieve query information. The student_info table is the same table used in the previous example, the ID column is the key attribute. The class_info table contains a key ID column and a class_name column. The class_info table contains the current classes that students are attending. The tables are structured as follows:

STUDENT_INFO TABLE

Name	Null?	Type
ID	NOT NULL	NUMBER(8)
FNAME		CHAR(15)
LNAME	NOT NULL	CHAR(60)

CLASS_INFO TABLE

Name	Null?	Type
ID	NOT NULL	NUMBER(8)
CLASS_NAME	NOT NULL	CHAR(30)

The query will retrieve the student id, first name, last name, and classes a student is attending. The structure of the query is:

```
select student_info.id,
       student_info.fname,
       student_info.lname,
       class_info.class_name
  from student_info, class_info
  where student_info.lname = 'DOE' and
        student_info.id = class_info.id;
```

The results of the query are:

```
ID        FNAME     LNAME     CLASS_NAME
--------  -------   ------    --------------------
88888     JOHN      DOE       INTRO TO ACCOUNTING
88888     JOHN      DOE       CALCULUS I
88888     JOHN      DOE       ECONOMICS II
```

The performance information for the query is as follows:

```
================================================================

      count      cpu     elap     phys       cr      cur     rows
Parse:    1        1        1        0        0        0
```

Execute:	1	656	**687**	**219**	388	496	0
Fetch:	1	697	**717**	**162**	184	277	3

Execution plan:

```
MERGE JOIN
  SORT (JOIN)
    TABLE ACCESS (FULL) OF 'CLASS_INFO'
  SORT (JOIN)
    TABLE ACCESS (FULL) OF 'STUDENT_INFO'
```

==========

The above performance information indicates that the number of data blocks read from disk during the execute and fetch steps totals 381. The elapsed time in hundredths of seconds for all steps totals 1405. The performance of this query can be enhanced by denormalizing the STUDENT_INFO table. Adding class_name to the STUDENT_INFO table will enhance the query by eliminating the join required to retrieve the data. This denormalization violates second normal form because class_name will contain multiple values for each student id. Class_name is not functionaly dependent on the primary key student id attribute. The STUDENT_INFO table before denormalization contained 8792 rows but after denormalization it contains 35167 rows. The structure of the new STUDENT_INFO table is:

```
STUDENT_INFO3 STRUCTURE
Name                                Null?    Type
----------------------------------- -------- ----
ID                                  NOT NULL NUMBER(8)
FNAME                                        CHAR(15)
LNAME                               NOT NULL CHAR(60)
CLASS_NAME                          NOT NULL CHAR(30)
```

The structure of the query is:

```
select ID,
       FNAME,
       LNAME,
       CLASS_NAME
from student_info3
where lname='DOE';
```

The query yields the same results but the performance is enhanced.

```
ID        FNAME     LNAME     CLASS_NAME
--------  -------   ------    --------------------
88888     JOHN      DOE       INTRO TO ACCOUNTING
88888     JOHN      DOE       CALCULUS I
88888     JOHN      DOE       ECONOMICS II
```

==========

	count	cpu	elap	phys	cr	cur	rows
Parse:	1	0	0	0	0	0	
Execute:	1	0	0	0	0	2	0
Fetch:	1	154	156	101	256	0	3

Execution plan:

TABLE ACCESS (FULL) OF 'STUDENT_INFO3'

==========

Note the physical reads have been reduced to 101 and the elapsed time is reduced to 156. The query was enhanced by adding the class_name to the STUDENT_INFO table but this caused redundant data to occur for the fname and lname columns. This redundancy could lead to insert, update, and delete anomalies. One example of an update anomaly could occur when the last name of a student changes. All rows would have to be updated for each class attended. If just one row was missed in the update process, inconsistencies in the database would occur.

The last example, once again, first demonstrates the joining of two tables in a query. The STU_INFO table contains attributes id, fname, lname, and school_code. The school_code attribute represents a six character code which corresponds to the school_desc field in the SCHOOL_INFO table. The

SCHOOL_INFO table contains a school code and description for most high schools and colleges in the country. The SCHOOL_INFO table contains 31512 rows. The tables are structured as follows:

STU_INFO STRUCTURE

Name	Null?	Type
ID	NOT NULL	NUMBER(8)
FNAME		CHAR(15)
LNAME	NOT NULL	CHAR(60)
SCHOOL_CODE	NOT NULL	CHAR(6)

SCHOOL_INFO STRUCTURE

Name	Null?	Type
SCHOOL_CODE	NOT NULL	CHAR(6)
SCHOOL_DESC	NOT NULL	CHAR(30)

The query will retrieve school description from the SCHOOL_INFO table based on the school code in the STU_INFO table. This requires the joining of the two tables. The query is structured as follows:

```
select stu_info.id,
       stu_info.fname,
```

```
        stu_info.lname,
        school_info.school_desc
from stu_info, school_info
where stu_info.id = '77777'
      and stu_info.school_code = school_info.school_code
```

The result of the query is:

```
ID          FNAME           LNAME              SCHOOL_DESC
--------    ------------    ---------------    ----------------
77777       JAMES           JONES              HOLT HIGH SCHOOL
```

The performance information for the query is as follows:

```
================================================================

        count      cpu    elap     phys       cr      cur     rows
Parse:      1        1       1        0        0        0
Execute:    1      563    1009      760      887     1049        0
Fetch:      1      448     534      540      202      645        1

Execution plan:
MERGE JOIN
  SORT (JOIN)
    TABLE ACCESS (FULL) OF 'SCHOOL_INFO'
  SORT (JOIN)
```

TABLE ACCESS (FULL) OF 'STU_INFO'

==========

The performance information indicates that the number of blocks read from disk total 1300. The total elapsed time in hundredths of seconds is 1544. The query will be enhanced by adding school description to the STU_INFO table. This violates third normal form because school_desc depends on school_code which is a non prime attribute. The new structure of the table is:

STU_INFO2 TABLE STRUCTURE:

Name	Null?	Type
ID	NOT NULL	NUMBER(8)
FNAME		CHAR(15)
LNAME	NOT NULL	CHAR(60)
SCHOOL_CODE	NOT NULL	CHAR(6)
SCHOOL_DESC	NOT NULL	CHAR(30)

The query is structured as follows:

```
select id,
       fname,
       lname,
```

```
        school_desc
from    stu_info2
where   id = '77777'
```

The performance information for the query is as follows:

```
================================================================

      count       cpu      elap      phys        cr       cur      rows
Parse:    1         0         0         0         0         0
Execute:  1         0         0         1         0         2         0
Fetch:    1        50        50       132       132         0         1

Execution plan:
TABLE ACCESS (FULL) OF 'STU_INFO2'

================================================================
```

Note that the number of physical blocks read from disk and elapsed time for the query has been reduced. Performance was enhanced by eliminating the join to the SCHOOL_INFO table which was large. The disadvantage of this type of denormalization is evident when a school description changes. If a school description changes all rows in the STU_INFO2 table which reference the old school description must change.

When the tables were normalized, only one row in the SCHOOL_INFO table would have to be changed.

CONCLUSION:

Query performance should be considered when designing relational database systems.[9] The types of queries which will be used should be evaluated during the design phase to insure optimal query performance. Query performance is reduced when tables must be joined to satisfy a query request. The denormalization of tables will enhance query performance by reducing the number of joins needed for query processing. Denormalization can cause update, delete, or insert anomalies to occur. Normalized database design helps insure database integrity. The tradeoff between denormalized and normalized design must be evaluated during the design phase.

UTILIZING INDEXES TO ENHANCE QUERY PERFORMANCE

INTRODUCTION:

The following section will examine how indexes can be used to enhance query performance. The relational database management system which will be used for this examination is Oracle version 6.0. Although Oracle is being used in this instance, the general concepts discussed in the following section can be applied to most of the other popular relational database management systems.

ADVANTAGES:

Indexes are the simplest way to enhance query performance.[5] The performance of a SQL query can be enhanced by simply creating an index on a column or columns which are referenced in the WHERE clause of the SQL statement. If an index is created with more than one column the index is called a concatenated index.[6] Concatenated indexes can enhance the query performance of SQL statements in which the "where" clause references all or the leading portion of the columns in the concatenated index.

The introduction of indexes does not require any changes to the wording of the SQL statements. The index is then

maintained by the relational database management system. Any change to the indexed column is automatically maintained during additions, deletions and modifications to the column. Indexes are fully independent of the table data. An index can be dropped without effecting the table which the index was created for or any other indexes on that table.[6]

In addition to enhanced performance, another advantage of indexes is their ability to enforce uniqueness. Indexes can be used to enforce uniqueness by creating the index with the UNIQUE option.[7] This will guarantee that all rows in a NOT NULL column or columns are unique and subsequent entries into this column will be unique. If uniqueness is needed across more than one column a concatenated index can be created with the UNIQUE option.

DISADVANTAGES:

Additional overhead results from the use of indexes. Many indexes can be created for a given table but overhead increases for each additional index that is added. The over use of indexes will cause slow system performance during insert, delete, and modify operations.[5] All indexes on a table must be updated when a row is inserted or deleted. A Index will also be updated when its associated indexed column is modified. Thus a tradeoff exists between query performance and the performance of insertions, deletions, and

modifications.

Disk drive contention is also increased through the use of indexes when table and index data reside on the same disk.[5] Index and table information must be read from disk when a indexed column or columns are selected in a query. Insertions, deletions, and modifications cause index and table information to be written to disk. The reading and writing of index and table data on the same disk causes contention. If multiple disk drives exist, contention can be reduced by placing indexes and tables on separate devices.

INDEX STRUCTURE:

Oracle uses B-tree indexes.[6] B-tree indexes equalize access to any row in a indexed table by balancing tree information.[4] Every leaf node is the same distance from the root node of the tree. All searches require traversing an equal number of nodes from the root down to the leaves.[4] This results in consistent and fast access time for both exact and range searches.

The upper nonleaf nodes of B-tree indexes contain navigational information which points to the lower level nodes.[4] The nonleaf nodes of the tree do not contain key value data. The navigational information consists of separator pair values which are sorted in ascending order and pointers to lower level nodes. Leaf nodes at the bottom of

the tree contain all indexed data values and their associated ROWID in a double linked list.[6] The ROWID is the logical address of a row in a table and it is the fastest method of accessing a particular row in a table.[6] There is one ROWID per data value for unique indexes.[6] The ROWID is appended to the end of the index key for non-unique indexes and sorted by key and ROWID.[6] Searching for a key value in a B-tree index starts at the root node and continues until a separator that is less than or equal to the key is found. The pointer to the right of the separator is followed to the leaf node where a compare is done to see if the values match. The main disadvantage of B-tree indexes is storage overhead.[5] B-tree indexes consume more storage than conventual indexes because of the space required by the interior nodes. Another disadvantage is the system overhead incurred in managing the balanced data structure.[5] Figure 1 and figure 2 illustrate the B-tree index structure.[5]

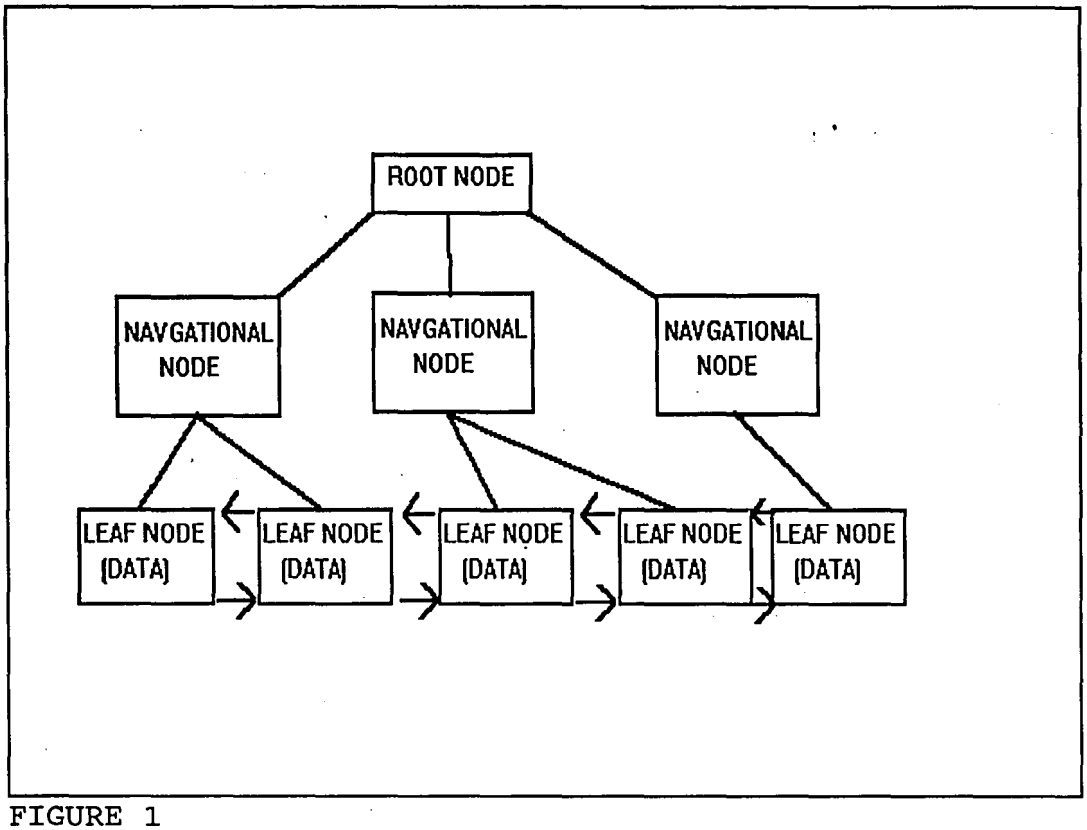

FIGURE 1

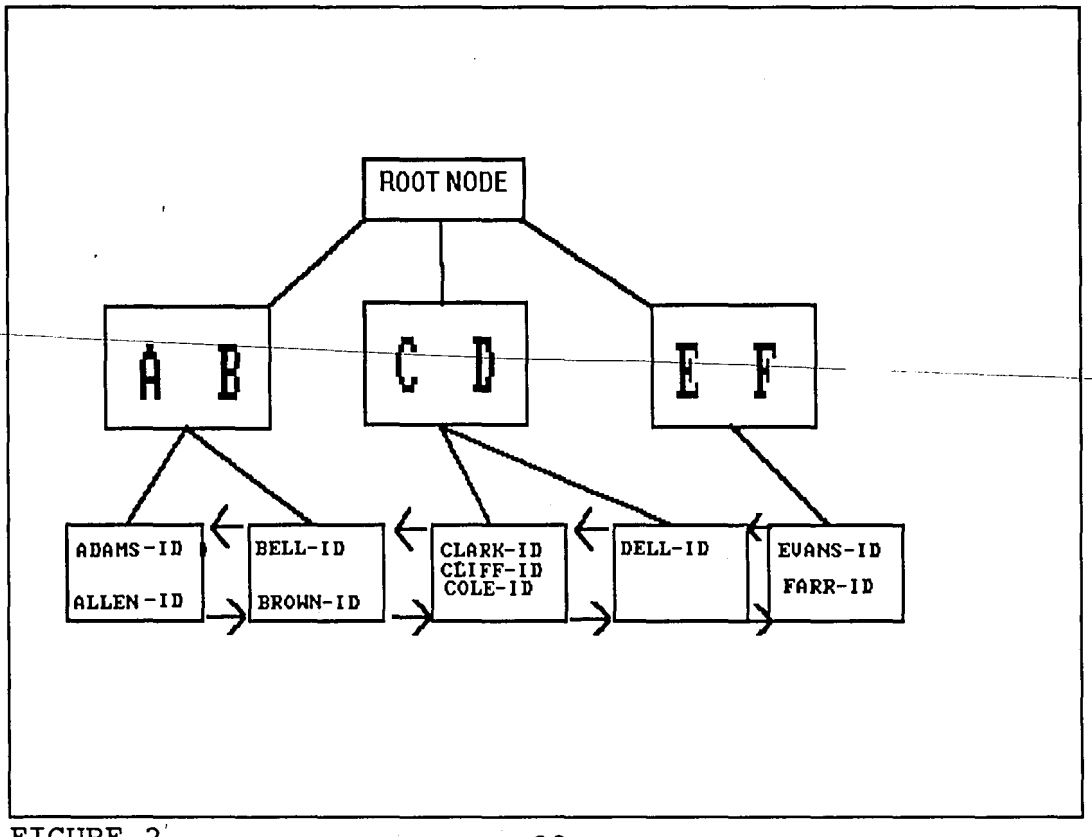

FIGURE 2

INDEX PERFORMANCE

The use of indexes will enhance SQL query performance under most circumstances. There are exceptions to this rule which will be discussed later. The following examples illustrate the effectiveness of indexes in increasing query performance.

The first example will demonstrate a simple query without a index and then with a index. The performance of the queries will then be compared. The structure of the table to be queried is presented below.

NAME_INFO TABLE STRUCTURE

Name	Null?	Type
NAME_PIDM	NOT NULL	NUMBER(8)
NAME_SSN	NOT NULL	CHAR(9)
NAME_LAST	NOT NULL	CHAR(60)
NAME_FIRST		CHAR(15)
NAME_MI		CHAR(15)

In this example the NAME_LAST column will be queried for a particular value and one row will be returned. The NAME_INFO table contains 25,000 rows. The structure of the query and its performance information is presented below.

```
select name_ssn,
       name_last,
       name_first,
       name_mi
from name_info
where name_last = 'Selnick';
```

```
================================================================

        count      cpu    elap     phys        cr      cur      rows
Parse:      1        0       0        0         0        0
Execute:    1        0       0        0         0        2         0
Fetch:      1      156     156      466       513        1         1

Execution plan:
TABLE ACCESS (FULL) OF 'NAME_INFO'

================================================================
```

The performance information execution plan indicates that a full table scan was performed. No index was used by the query. The query elapsed time was 156 hundredths of seconds and the number of data blocks read from disk was 466. The query's performance can be enhanced by creating a non-unique index named I_NAME_INFO$LAST on the name_last column. The index will be used in the execution plan because the name_last

column is referenced in the WHERE clause of the SQL statement. The performance information for the query after the index was created is presented below.

```
============================================================

select name_ssn,
       name_last,
       name_first,
       name_mi
from name_info
where name_last = 'Selnick'
```

	count	cpu	elap	phys	cr	cur	rows
Parse:	1	0	0	0	0	0	
Execute:	1	0	0	0	0	0	0
Fetch:	1	0	4	6	5	0	1

```
Execution plan:
TABLE ACCESS (BY ROWID) OF 'NAME_INFO'
  INDEX (RANGE SCAN) OF 'I_NAME_INFO$LAST' (NON-UNIQUE)

============================================================
```

The performance information indicates that the I_NAME_INFO$LAST index was utilized for this query. The

elapsed time and physical reads where reduced through the use of this index. The execution plan shows that the table data was accessed by ROWID after the index information was found so that a full table scan was avoided.

The next example will compare the performance of a join with and without the use of an index. The name information table (name_info) will be joined with an address information table (addr_info). Both tables contain approximately 25,000 rows. A unique identification number (PIDM) will be used to join the two tables. The structure of the name_info table will remain the same and the structure of the addr_info table is presented below.

ADDR_INFO TABLE STRUCTURE

Name	Null?	Type
ADDR_PIDM	NOT NULL	NUMBER(8)
ADDR_STREET1		CHAR(30)
ADDR_STREET2		CHAR(30)
ADDR_CITY		CHAR(30)
ADDR_ST		CHAR(2)
ADDR_ZIP		CHAR(5)

The query will join the addr_info table in order to retrieve the city for a selected name. The structure of the query and performance information is presented below.

```
================================================================

select name_ssn,
       name_last,
       addr_city
from name_info, addr_info
where name_last = 'Selnick' and name_pidm = addr_pidm

      count      cpu    elap     phys       cr      cur     rows
Parse:    1        1       1        0        0        0
Execute:  1        1       1        0        0        2        0
Fetch:    1     4987    5215      738   170615        1        1

Execution plan:
NESTED LOOPS
  TABLE ACCESS (FULL) OF 'ADDR_INFO'
  TABLE ACCESS (BY ROWID) OF 'NAME_INFO'
    INDEX (RANGE SCAN) OF 'I_NAME_INFO$LAST' (NON-UNIQUE)

================================================================
```

The performance information indicates that the addr_info table was accessed by a full table scan for the join. The I_NAME_INFO$LAST was used for access to the name_info table as in the previous example. The full table scan of the addr_info table caused the poor performance of this query. The cr

statistic pertains to the number of buffers retrieved in consistent mode.[5] A buffer is retrieved in consistent mode when a read-consistent version of the buffer is needed.[5] The number of consistent mode buffers is high since much of the table information is buffered in memory for this query. In order to enhance the performance of this join, a unique index named I_ADDR_INFO$PIDM will be created on the addr_pidm in the addr_info table. The performance information for the query after the index was created is presented below.

==========

```
select name_ssn,
       name_last,
       addr_city
from name_info, addr_info
where name_last = 'Selnick' and name_pidm = addr_pidm
```

	count	cpu	elap	phys	cr	cur	rows
Parse:	1	1	1	0	0	0	
Execute:	1	0	0	0	0	0	0
Fetch:	1	1	1	7	10	0	1

```
Execution plan:
NESTED LOOPS
  TABLE ACCESS (BY ROWID) OF 'NAME_INFO'
```

```
   INDEX (RANGE SCAN) OF 'I_NAME_INFO$LAST' (NON-UNIQUE)
TABLE ACCESS (BY ROWID) OF 'ADDR_INFO'
   INDEX (RANGE SCAN) OF 'I_ADDR_INFO$PIDM' (UNIQUE)
```

==========

The execution plan indicates that the addr_info table was accessed by ROWID through the I_ADDR_INFO$PIDM index. This caused the performance of the join to increase. The total elapsed time went from 5217 to 2. The total physical reads went from 738 to 7 and the total number of consistent mode buffers went from 170615 to 10. This dramatic increase in performance proves the importance of indexes in enhancing query performance especially, when joins are involved.

INDEX RULE OF THUMB:

The use of indexes under certain circumstances can reduce the performance of queries. These circumstances pertain to the reading of index and table information. Table information is not read when only the index key or a portion of the index key is requested in the query. Table data, in addition to index data, must be read when information is requested other than or in addition to the index key.[5] The performance of the query will decrease when the total number of index and table block reads exceeds the amount of table blocks read

during a full table scan.

Determining the number of index and table blocks read for a query is difficult; therefore the number of rows returned by the query can be used to determine if the use of a index is beneficial. The rule of thumb suggested in the Oracle Database Administrators's Guide is to create an index if you will frequently want to retrieve less than 10-15 percent of the rows in a large table.[6] If more than 10-15 percent of the table's rows are returned than a full table scan is suggested.[6] Small tables of eight data blocks or less should always be accessed by a full table scan.

The index rule of thumb is a general guide for determining whether to retrieve rows by index or full table scan. Under certain circumstances, indexes can yield better results when more than 10-15 percent of the rows are returned. The size of the indexed column or columns effects the optimal percentage. The percentage increases when the length of the indexed column or columns is smaller and decreases when the length of the column or columns is larger. This is because the number of index blocks that must be read to retrieve a given number of rows increases when the length of the index column increases. Therefore less rows can be retrieved efficiently using the index because more index information must be read. Conversely, more rows can be retrieved efficiently when the index column or columns are smaller because less index blocks need to be read for each row

retrieved. Therefore the length of the index column or columns directly effects the index rule of thumb.

CONCLUSION:

The use of indexes can greatly enhance query performance in most circumstances. The columns that are used to join tables should be indexed to enhance the performance of the join operation. Indexes should not be used when a query returns a large amount of rows. In general a index should not be used when a query returns more than 10-15% of the rows in a table. The use of indexes causes additional overhead on the RDBMS. Indexes must be updated during insert, delete, and modify operations on indexed columns therefore the use of many indexes can cause slow performance during these operations.

CONTROLLING THE OPTIMIZER TO ENHANCE QUERY PERFORMANCE

INTRODUCTION

The role of the optimizer is to enhance the performance of the database system by determining how SQL statements can be executed efficiently.[5] The optimizer tries to determine the best access path to the data. Optimizers are used by all major vendors of relational database management systems. Their method of implementation distinguishes one product from another. The performance of the database relates directly to the implementation of the optimizer. Most optimizers are based on heuristics, therefore they are not always effective in finding the best access path to the data. It is important to understand how the optimizer is implemented in order to structure SQL statements for optimal performance. The Oracle optimizer will be examined in this section to show how knowledge of the optimizer can be useful in enhancing the performance of SQL query statements.

ORACLE OPTIMIZER:

The Oracle optimizer is part of the RDBMS kernel.[6] It tries to choose the best access path by evaluating all possible strategies. The optimizer will try to choose the

best strategy based on heuristic rules about what is best in which situation and a ranking scheme for WHERE clause predicates. The optimizer examines the syntax of the SQL statement, the predicates of the where clause, the database objects used, and any indexes that exist on the database objects to choose the execution plan.[6] The optimizer does not use any information about the relative size of the tables, key distribution within indexes, or other statistical methods in order to pick a particular access path. The optimizer can be influenced by creating indexes on columns referenced in the WHERE clause, by creating clusters, and by formulating SQL statements differently.[5]

RANKING RULES:

The following list represents query paths ranked in order of speed for Oracle version 6.0.[10] The lower rankings represent the faster paths with ROWID equals constant being the fastest path.

1.) ROWID = constant

2.) entire unique concatenated index = constant

3.) unique indexed column = constant

4.) entire cluster key = corresponding cluster key in another table in same cluster

5.) entire cluster key = constant

6.) entire non-unique concatenated index = constant

7.) non-unique single column index merge

8.) most leading concatenated index = constant

9.) indexed column BETWEEN low value AND high value, or indexed column LIKE 'C%' (bounded range)

10.) sort/merge (joins only)

11.) MAX or MIN of single indexed column

12.) ORDER BY entire index

13.) full table scans

14.) unindexed column = constant, or column is NULL, or column LIKE '%C%' (full table scan)

Selecting a row by ROWID is the fastest access method available; no table scans or indexes are used in the selection process.[5] ROWID is the hexadecimal representation of the address of the row. It contains the logical block number, row sequence number and file identification number of the row.[6] ROWID is a pseudo-column name that can be used as a column name, except it cannot be updated or inserted.[7] Therefore the following select statement can be used to access information by ROWID.

```
select * from dept where ROWID = '0000034B.0002.0001'
```

The second ranking is: entire unique concatenated index = constant. A concatenated index is created on two or more

columns in a table. Concatenated indexes are single indexes that reference more than one column in a table.[6] If all columns in the concatenated index are referenced in the WHERE clause as equal to constants, then the conditions for ranking two will be meet.

The third ranking is: unique indexed column = constant. A index created with the unique option ensures that no duplicate data exist in the column that the index was created on.[7] When a unique indexed column and non-unique indexed column are named in the WHERE clause, the ORACLE optimizer will use the unique index and disregard the non-unique index to avoid the merging of the two indexes.[10]

The fourth and fifth rankings refer to cluster keys which are used to cluster tables. The purpose of clustering is to read one block with data from all tables being joined, instead of reading one block per table.[5] Clustered tables pre-join rows from different tables so they are all in the same physical block. The information is accessed in the various tables by the cluster key.

The sixth ranking is: entire non-unique concatenated index = constant. The only difference between this and the second ranking is that the concatenated index is non-unique, which means duplicate data is allowed in the indexed columns.

The seventh ranking is: non-unique single column index merge. Multiple indexes will be used by the optimizer when two or more predicates on the same table are referenced in the

WHERE clause of the SQL statement. The indexes must be non-unique and the predicates must be equalities. Index information from each index search is merged together to obtain the query results.[10]

The eighth ranking is: most leading concatenated index = constant. This refers to the referencing of the leading column or columns of a concatenated index in a WHERE clause. A concatenated index can be used if all or the leading column or columns are referenced.[5] If only the trailing portion of the concatenated index is referenced the index will be ignored.

The ninth ranking is: indexed column BETWEEN low value AND high value, or indexed column LIKE 'C%'. Any indexed columns which are referenced in WHERE clauses using the BETWEEN or LIKE statements pertain to this rule. For example "WHERE deptno BETWEEN 10 AND 20" or "WHERE lastname LIKE 'JON%'. The percentage sign in the LIKE statement is used as a wildcard character and must trail a constant in order for the conditions of the rule to be meet.

The tenth ranking is: sort/merge for join operations. This rule applies to join operations which use the sort/merge routine to order data. The sort/merge routine increases the performance of certain operations such as joins.[5] Incoming data which must be ordered is first internally sorted into several sets of ordered runs, and these runs are merged into a final sorted result.

The eleventh ranking is: MAX or MIN of single indexed column. This rule refers to the use of the MIN or MAX function on a single indexed column. For example the SQL statement "select MAX(SAL) from emp" will make use of a index created on the SAL column. This is the one exception to the indexed column cannot be modified rule.[6] If any other function was used on the SAL column besides MAX or MIN, the index would not be used.

The twelfth ranking is: ORDER BY entire index. This ranking refers to using the ORDER BY statement on a indexed column or columns. For example "select * from emp ORDER BY ename" will use a indexed created on the ename column.

The thirteenth and fourteenth rankings refer to full table scans. A full table scan is performed if a select statement contains no WHERE clause or the WHERE clause references a un-indexed column.[5] Full table scans are also performed if the WHERE condition tests for the NULL condition or the LIKE statement tests for %constant%.[6] For example the following WHERE conditions will force full table scans. The clauses "WHERE deptno is NULL" or "WHERE lastname LIKE %ONES%" will force a full table scan.

OPTIMIZER INDEX RULES:

A index on a column can be used if the column is

referenced in the WHERE clause of the SQL statement although other conditions could exist to cause the index to be disregarded. A index can be used for equality, bounded range, and unbounded range searches.[10] The following SQL statements would use a index created on the DEPTNO column for the DEPT table. The index could be unique or non-unique.

```
select * from DEPT where DEPTNO = 9000;
select * from DEPT where DEPTNO between 2000 and 3000;
select * from DEPT where DEPTNO > 5000;
```

A index will not be used if the column is modified by a function or an expression, unless the function used is the MAX or MIN functions.[5] The index on the DEPTNO column will be disregarded on the following SQL statements due to the modification to DEPTNO.

```
select * from DEPT where DEPTNO + 1 > 1000;
select * from DEPT where DEPTNO * 2 = 2000;
```

The index on the DNAME column will be disregarded on the following SQL statements because of function modification to the DNAME column.

```
select * from DEPT where UPPER(DNAME) ='ACCOUNTING';
```

```
select * from DEPT where SUBSTR(DNAME,1,1) ='A';
```

The index on the DEPTNO column will be used for the following SQL statements. In the first SQL statement the DEPTNO column is not modified therefore the index can be used. The second and third SQL statements use the MIN and MAX functions which are the only functions which can modify a column and still use the index on that column.

```
select * from DEPT where DEPTNO > 5000/2;
select MAX(DEPTNO) from DEPT;
select MIN(DEPTNO) from DEPT;
```

Modification to a column can be used intentionally to disable index use.[6] Under certain circumstances, disabling index use can speed performance of the SQL statement.[5] The index rules of thumb which are mentioned in section two must be considered when determining whether to intentionally disable an index. A index on a character column can be disabled by concatenating the column with a null value.[6] A index on a number or date column can be disabled by adding a zero to the column value.[6] The following SQL statements disable the use of indexes without effecting the result of the queries.

```
select * from DEPT where DNAME||'' = 'ACCOUNTING';
```

```
select * from DEPT where DEPTNO + 0 = 1000;
select * from DEPT where DEPT_DATE + 0 = TO_DATE('01-DEC-91');
```

A concatenated index can be used if all columns or the leading columns of the concatenated index are referenced in the WHERE clause of the SQL statement.[6] The order of the columns in the WHERE clause is un-important but at least the first column of the concatenated index must be referenced in order for the index to be used. The following SQL statements will use the concatenated index created on the DEPTNO, DNAME, and DEPT_DATE columns for the DEPT table. DEPTNO is the first column of the concatenated index.

```
create index dept$cat on DEPT (DEPTNO,DNAME,DEPT_DATE);
select * from DEPT where DEPTNO > 1000 and DNAME
='ACCOUNTING';
select * from DEPT where DEPTNO = 2000 and DEPT_DATE >
TO_DATE('01-DEC-91');
select * from DEPT where DEPTNO > 9000;
```

The following SQL statement will not use the concatenated index because the first column of the index is not referenced.

```
select * from DEPT where DNAME='ACCOUNTING' and
DEPT_DATE=TO_DATE('01-DEC-91');
```

A index can be used for the LIKE clause if the column is a character column and the comparison string starts with a character.[10] If the column is a number or date column than Oracle internally has to modify the column with a function to convert it to character and therefore an index on the column cannot be used. The following SQL statement will make use of the index created on the DNAME column.

```
select * from DEPT where DNAME LIKE 'ACC%';
```

The following SQL statements will not use any indexes because the comparison string does not start with a character or the column type in not character.

```
select * from DEPT where DNAME LIKE '%ACC%';
select * from DEPT where DEPTNO LIKE '20%';
select * from DEPT where DEPT_DATE LIKE '01%';
```

Indexes are not used if the IS NULL or IS NOT NULL conditions are used in the WHERE clause of the SQL statement.[6] There is no index entry if the column values in a index have a null value therefore a index cannot be used for this test condition. The following SQL statements will not use indexes because of the use of IS NULL or IS NOT NULL conditions.

```
select * from DEPT where DNAME IS NULL;
select * from DEPT where DEPTNO IS NOT NULL;
```

A heuristic rule of the Oracle optimizer is that most records will be retrieved for a NOT EQUAL to condition therefore the optimizer will not use indexes for this test condition.[10] The optimizer will transform other NOT conditions so indexes can be used. The optimizer will evaluate a NOT > expression to a <= expression, a NOT < expression to a >= expression, a NOT >= expression to a < expression, and a NOT <= expression to a > expression.[5] The following SQL statement will not use a index due to the presence of the NOT equal condition.

```
select * from DEPT where DEPTNO != 0;
```

The following SQL statements can use indexes because of the transformation of the condition.

```
select * from DEPT where NOT DEPTNO > 2000; is transformed to
select * from DEPT where DEPTNO <= 2000;
select * from DEPT where NOT DEPTNO < 2000; is transformed to
select * from DEPT where DEPTNO >= 2000;
select * from DEPT where NOT DEPTNO >= 2000; is transformed to
where DEPTNO < 2000;
select * from DEPT where NOT DEPTNO <= 2000; is transformed to
```

```
select * from DEPT where DEPTNO > 2000;
```

The DISTINCT, UNION, MINUS, INTERSECT, and GROUP BY operators will not use indexes when referenced in a SQL statement.[5] The DISTINCT and GROUP BY operators require a sort operation when referenced.[10] The UNION, MINUS, and INTERSECT operators require two sort operations when referenced.[10] The following SQL statements will not use indexes.

```
select DISTINCT DEPTNO from EMP;
select DEPTNO from DEPT MINUS select DEPTNO from EMP;
select DEPTNO from DEPT UNION select DEPTNO from EMP;
select DEPTNO from DEPT INTERSECT select DEPTNO from EMP;
select DEPTNO from DEPT GROUP BY DEPTNO;
```

INDEX SELECTION RULES:

The optimizer will select which indexes to use based on heuristic rules which are incorporated in the design of the optimizer.[10] The optimizer will select a unique index before a non-unique index.[6] The following SQL statements create a unique index dept$deptno on the DEPTNO column and a non-unique index dept$dname on the DNAME column. Both columns are referenced in the SELECT statement but only the

dept$deptno index will be used by the optimizer. The unique index is viewed to be more selective and the optimizer disregards the non-unique index to avoid the merging of the two indexes.

```
create unique index dept$deptno on DEPT (DEPTNO);
create index dept$dname on DEPT (DNAME);
select * from DEPT where DEPTNO = 2000 and DNAME =
'ACCOUNTING';
```

If multiple unique indexes are referenced in the SQL statement then the optimizer will use the first column specified in the WHERE clause.[10] In the following SELECT statement the dept$dept_date index will be used because the DEPT_DATE column is specified first in the WHERE clause.

```
create unique index dept$dept_date on DEPT (DEPT_DATE);
create unique index dept$deptno on DEPT(DEPTNO);
select * from DEPT where DEPT_DATE = TO_DATE('01-JAN-90') and
DEPTNO = 1000;
```

When multiple non-unique indexes are referenced as equalities in the WHERE clause of the SQL statement, the optimizer will merge up to five of the indexes.[10] If more than five non-unique indexes are referenced as equalities, the optimizer will use the first five mentioned in the WHERE

clause. If five indexes are being merged for a given SQL statement then performance may improve by disabling some of the indexes. The dept$deptno and dept$dname indexes will be merged for the following SELECT statement.

```
create index dept$deptno on DEPT (DEPTNO);
create index dept$dname on DEPT (DNAME);
select * from DEPT where DEPTNO = 2000 and DNAME =
'ACCOUNTING';
```

An index merge is not performed on non-unique indexes when a condition includes a unbounded range test.[10] If an unbounded range condition and an equality condition exist, then the index for the equality condition will be used. If two or more unbounded range conditions exist, then the index on the first unbounded range condition in the WHERE clause will be used. In the following SQL statements the dept$deptno index will be used and a merge will not take place.

```
create index dept$deptno on DEPT (DEPTNO);
create index dept$dept_date on DEPT (DEPT_DATE);
select * from DEPT where DEPTNO=1000 and DEPT_DATE >
TO_DATE('01-JAN-90');
select * from DEPT where DEPTNO > 1000 and DEPT_DATE >
TO_DATE('01-JAN-90');
```

A concatenated index can be used for unbounded range conditions.[10] The following SQL statements will use the concatenated index dept$cat for the following SELECT statements.

```
create index dept$cat on DEPT (DEPTNO, DEPT_DATE);

select * from DEPT where DEPTNO=1000 and DEPT_DATE >
TO_DATE('01-JAN-90');
select * from DEPT where DEPTNO > 1000 and DEPT_DATE >
TO_DATE('01-JAN-90');
```

OPTIMIZER JOIN RULES:

When tables are joined, the optimizer will decide how to join the tables, which table will be the driving table, the join chain, and the best access path to the tables. The joining of tables can occur through a full table scan join, a sort/merge join, or a index join. The table to start the join with is the driving table. The order in which the tables are joined is the join chain.

A full table scan join occurs when a non-indexed and a non-equal join predicate are used to join the tables.[10] When the columns of the join predicate are not indexed and a non-equal condition is used in the join predicate, a full

table scan join is used. The driving table of the full table scan join will be the last table referenced in the FROM clause.[10] For every row of the driving table, a full table scan of the non-driving table is needed therefore this type of join is the most time consuming. The following SQL statement will join the DEPT and EMP tables by a full table scan join. The driving table will be the EMP table. The DEPT.DEPTNO and EMP.DEPTNO columns are not indexed.

```
select DEPTNO, ENAME from DEPT, EMP where DEPT.DEPTNO >
EMP.DEPTNO
```

The sort/merge join is used when the join predicate condition is an equality and the columns referenced in the join predicate are not indexed.[10] The sort/merge executes the join by performing two separate queries and then merging the results of the queries.[6] The following SQL statement will use the sort/merge join. The DEPT.DEPTNO and EMP.DEPTNO columns are not indexed. Since two separate queries are executed, there is no driving table.

```
select DNAME,ENAME from DEPT,EMP where DEPT.DEPTNO =
EMP.DEPTNO;
```

The optimizer will execute this query as two separate queries and then merge the results. The optimizer breaks the query

into the following queries and then merges the results.

```
select DEPTNO,DNAME from DEPT order by DEPTNO;
select DEPTNO,ENAME from EMP order by DEPTNO;
```

An indexed join is used for a join or non-join predicate which has indexed columns.[10] When both columns of the join predicate are indexed, the optimizer will choose the access path based on the query path rules introduced earlier in this section. If the ranks are equal the optimizer will use the last table in the FROM clause as the driving table. Thus it is advantageous to list the table with the smallest number of qualified rows last in the FROM list. The following SQL statement will use a indexed join to join the DEPT and EMP tables. Both columns in the join predicates are indexed with non-unique indexes so the last table in the FROM list, which is the EMP table, will be the driving table.

```
select * from DEPT, EMP where DEPT.DEPTNO = EMP.DEPTNO;
```

When only one of the columns of the join predicate is indexed, then the driving table is the one without the indexed column.[10] In the following SQL statement the DEPT.DEPTNO column is indexed and the EMP.DEPTNO is not. The EMP table will be the driving table. A full table scan will be used to access the EMP table and the index on DEPT.DEPTNO will be used

to access the DEPT table.

```
select * from EMP, DEPT where DEPT.DEPTNO = EMP.DEPTNO;
```

When a index exists only in a non-join predicate then that index is used for the index join.[10] In the following SQL statement a index exists on the DNAME column which is referenced in a non-join predicate. The DNAME column is the only column which is indexed therefore the index on DNAME will be used for the indexed join. The EMP table will be the driving table since it has no usable indexes.

```
select * from DEPT, EMP where EMP.DEPTNO = DEPT.DEPTNO and
DNAME = 'ACCOUNTING';
```

The outer join is used to join two or more tables and return rows from one table which have no direct match in the other table.[7] The outer join is applied by placing a (+) after the outer join table in the join predicate.[7] When a outer join occurs, the non-outer join table is the driving table. In the following SQL statement the driving table is the DEPT table.

```
select * from DEPT, EMP where DEPT.DEPTNO = EMP.DEPTNO (+)
```

A index will be used for non-join predicates on the non-outer

join table because they are applied before the join.[10] The following SQL statement will use a index created on the DNAME column which is referenced in the non-outer join table DEPT.

```
select * from DEPT, EMP where DEPT.DEPTNO = EMP.DEPTNO(+) and
DNAME = 'ACCOUNTING';
```

Indexes will also be used for non-join predicates on the outer join table with a (+) appended to the column name.[10] If a (+) is not appended to the column name the index will not be used. In the following SQL statements, the first SELECT statement will use the index on the SAL column and the second SELECT statement will not use the index on the SAL column.

```
select * from DEPT, EMP where DEPT.DEPTNO = EMP.DEPTNO(+) and
EMP.SAL(+) = 1000;
select * from DEPT, EMP where DEPT.DEPTNO = EMP.DEPTNO(+) and
EMP.SAL = 1000;
```

If both columns in the join predicate are indexed, the index on outer join column will be used to access the table and a full table scan will be used for the non-outer join table.[10] In the following SQL SELECT statement, the index on DEPT.DEPTNO will be ignored and the DEPT table will be accessed by a full table scan. The EMP table will use the index on EMP.DEPT.

```
select * from DEPT, EMP where DEPT.DEPTNO = EMP.DEPTNO(+);
```

CONCLUSION:

It is important to understand how the optimizer works in order to write efficient SQL statements or to tune existing SQL statements. This section examined the rules of the optimizer and how it uses those rules to determine the access paths for the SQL statement. The optimizer uses heuristic rules to determine how a SQL statement should be executed. These rules will not always produce the best access path. It is therefore very important to understand the workings of the optimizer and manipulate it when it does not choose the best strategy for executing the SQL statement.

REFERENCES

1. Date, C. J., An Introduction to Database Systems. Reading,Mass:Addison-Wesley,1990.

2. Date, C. J., Relational database writings, 1985-1989. Reading,Mass:Addison-Wesley,1990.

3. Hursch, Carolyn J., SQL, structured query language. Blue Ridge Summit,Pa:Windcrest,1991.

4. Johnson, Ted and Shasha, Dennis, "Reexamining B-Trees", Dr. Dobb's Journal, Vol.184, 01/92, pp.44-48.

5. Oracle RDBMS Performance Tuning Guide Version 6.0. Redwood,Ca:Oracle Corporation,1990.

6. Oracle RDBMS Database Administrator's Guide Version 6.0. Redwood,Ca:Oracle Corporation,1990.

7. Oracle SQL Language Reference Manual Version 6.0. Redwood,Ca:Oracle Corporation,1990.

8. Oracle Unix Technical Reference Guide Version 6.0. Redwood,Ca:Oracle Corporation,1990.

9. Reiner, David S., Kim, Won and Batory, Don S., Query Processing in database systems. New York:Springer-Verlag,1985.

10. Supplied by Oracle Corporation, Redwood Ca.

VITA

The author was born on September 5, 1958. The author received a Bachelor of Science Degree in Business Management from the Pennsylvania State University in 1980 and an Associates Degree in Computer Science from Northampton County Area Community College in 1987. The author is currently employed by Lehigh University as a Senior Technical Services Consultant for the Administrative Systems department.

END

OF

TITLE