6

The Basics of Efficient SQL

In the previous chapter we examined the basic syntax of SQL in Oracle Database. This chapter will attempt to detail the most simplistic aspects of SQL code tuning. In other words, we are going to discuss what in SQL statements is good for performance and what is not. The approach to performance in this chapter will be based on a purely SQL basis. We want to avoid the nitty-gritty and internal processing occurring in Oracle Database at this stage. It is essential to understand the basic facts about how to write well-performing SQL code first, without considering specific details of Oracle software.

The most important rule of thumb with SQL statements, and particularly SELECT statements, those most subject to tuning, is what is commonly known as the "KISS" rule "Keep It Simple Stupid!" The simpler your SQL statements are the faster they will be. There are two reasons for this. Firstly, simple SQL statements are much more easily tuned and secondly, the Optimizer will function a lot better when assessing less complex SQL code. The negative effect of this is granularity but this negative effect depends on how the application is coded. For instance, connecting to and disconnecting from the database for every SQL code statement is extremely inefficient.

Part of the approach in this chapter is to present SQL performance examples without bombarding the reader with the details of too much theory and reference material. Any reference items such as explanations of producing query plans will be covered later on in this book.

So what this chapter will cover is mostly a general type of SQL code tuning. Thus the title of this chapter: "The Basics of Efficient SQL." Let's start with a brief look at the SELECT statement.

6.1 The SELECT Statement

It is always faster to SELECT exact column names. Thus using the Employees schema

```
SELECT division_id, name, city, state, country FROM division;
```

is faster than

```
SELECT * FROM division;
```

Also since there is a primary key index on the Division table

```
SELECT division_id FROM division;
```

will only read the index file and should completely ignore the table itself. Since the index contains only a single column and the table contains five columns, reading the index is faster because there is less physical space to traverse.

In order to prove these points we need to use the EXPLAIN PLAN command. Oracle Database's EXPLAIN PLAN command allows a quick peek into how the Oracle Database Optimizer will execute an SQL statement, displaying a query plan devised by the Optimizer.

The EXPLAIN PLAN command creates entries in the PLAN_TABLE for a SELECT statement. The resulting query plan for the SELECT statement following is shown after it. Various versions of the query used to retrieve rows from the PLAN_TABLE, a hierarchical query, can be found in Appendix B. In order to use the EXPLAIN PLAN command statistics must be generated. Both the EXPLAIN PLAN command and statistics will be covered in detail in Chapter 9.

```
EXPLAIN PLAN SET statement_id='TEST' FOR SELECT * FROM
  division;
Query                                    Cost      Rows     Bytes
---------------------------------------- --------  -------- -----------
SELECT STATEMENT on                         1         10       460
 TABLE ACCESS FULL on DIVISION              1         10       460
```

One thing important to remember about the EXPLAIN PLAN command is it produces a listed sequence of events, a query plan. Examine the following query and its query plan. The "Pos" or positional column gives a rough guide to the sequence of events that the Optimizer will follow. In general, events will occur listed in the query plan from bottom to top, where additional indenting denotes containment.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT di.name, de.name, prj.name,
        SUM(prj.budget-prj.cost)
      FROM division di JOIN department de USING(division_id)
            JOIN project prj USING(department_id)
      GROUP BY di.name, de.name, prj.name
      HAVING SUM(prj.budget-prj.cost) > 0;
```

```
Query                                          Pos  Cost    Rows   Bytes
---------------------------------------------- ---  ------  -----  ------
SELECT STATEMENT  on                            97    97     250   17500
 FILTER  on                                      1
  SORT GROUP BY on                               1    97     250   17500
   HASH JOIN  on                                 1    24   10000  700000
    TABLE ACCESS FULL on DIVISION                1     1      10     170
    HASH JOIN  on                                2     3     100    3600
     TABLE ACCESS FULL on DEPARTMENT             1     1     100    1900
     TABLE ACCESS FULL on PROJECT                2    13   10000  340000
```

Now let's use the Accounts schema. The Accounts schema has some very large tables. Large tables show differences between the costs of data retrievals more easily. The GeneralLedger table contains over 700,000 rows at this point in time.

In the next example, we explicitly retrieve all columns from the table using column names, similar to using SELECT * FROM GeneralLedger. Using the asterisk probably involves a small overhead in re-interpretation into a list of all column names, but this is internal to Oracle Database and unless there are a huge number of these types of queries this is probably negligible.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT generalledger_id,coa#,dr,cr,dte FROM
        generalledger;
```

The cost of retrieving 752,740 rows is 493 and the GeneralLedger table is read in its entirety indicated by "TABLE ACCESS FULL".

```
Query                                    Cost     Rows      Bytes
---------------------------------------- -------  --------  --------
SELECT STATEMENT  on                      493     752740    19571240
 TABLE ACCESS FULL on GENERALLEDGER       493     752740    19571240
```

Now we will retrieve only the primary key column from the GeneralLedger table.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT generalledger_id FROM generalledger;
```

For the same number of rows the cost is reduced to 217 since the byte value is reduced by reading the index only, using a form of a full index scan. This means that only the primary key index is being read, not the table.

```
Query                                          Cost    Rows    Bytes
---------------------------------------------- ------- ------- -------
SELECT STATEMENT  on                            217  752740  4516440
 INDEX FAST FULL SCAN on XPKGENERALLEDGER       217  752740  4516440
```

Here is another example using an explicit column name but this one has a greater difference in cost from that of the full table scan. This is because the column retrieved uses an index, which is physically smaller than the index for the primary key. The index on the COA# column is consistently 5 bytes in length for all rows. For the primary key index only the first 9,999 rows have an index value of less than 5 bytes in length.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT coa# FROM generalledger;
```

```
Query                                      Cost     Rows     Bytes
------------------------------------------ -------- -------- --------
SELECT STATEMENT  on                          5   752740   4516440
 INDEX FAST FULL SCAN on XFK_GL_COA#          5   752740   4516440
```

Following are two interesting examples utilizing a composite index. The structure of the index is built as the SEQ# column contained within the CHEQUE_ID column (CHEQUE_ID + SEQ#) and not the other way around. In older versions of Oracle Database this probably would have been a problem. The Oracle9*i* Database Optimizer is now much improved when matching poorly ordered SQL statement columns to existing indexes. Both examples use the same index. The order of columns is not necessarily a problem in Oracle9*i* Database.

10g Oracle Database 10*g* has Optimizer improvements such as less of a need for SQL code statements to be case sensitive.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT cheque_id, seq# FROM cashbookline;
```

```
Query                                      Cost    Rows    Bytes
------------------------------------------ ------- ------- -------
SELECT STATEMENT  on                         65  188185  1505480
 INDEX FAST FULL SCAN on XPKCASHBOOKLINE     65  188185  1505480
```

It can be seen that even with the columns selected in the reverse order of the index, the index is still used.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT seq#,cheque_id FROM cashbookline;
```

Query	Cost	Rows	Bytes
SELECT STATEMENT on	65	188185	1505480
INDEX FAST FULL SCAN on XPKCASHBOOKLINE	65	188185	1505480

The GeneralLedger table has a large number of rows. Now let's examine the idiosyncrasies of very small tables. There are some differences between the behavior of SQL when dealing with large and small tables.

In the next example, the Stock table is small and thus the costs of reading the table or the index are the same. The first query, doing the full table scan, reads around 20 times more physical space but the cost is the same. When tables are small the processing speed may not be better when using indexes. Additionally when joining tables the Optimizer may very well choose to full scan a small static table rather than read both index and table. The Optimizer may select the full table scan as being quicker. This is often the case with generic static tables containing multiple types since they are typically read more often.

```
EXPLAIN PLAN SET statement_id='TEST' FOR SELECT * FROM
  stock;
```

Query	Cost	Rows	Bytes
SELECT STATEMENT on	**1**	118	**9322**
TABLE ACCESS FULL on STOCK	1	118	9322

```
EXPLAIN PLAN SET statement_id='TEST' FOR SELECT stock_id
  FROM stock;
```

Query	Cost	Rows	Bytes
SELECT STATEMENT on	**1**	118	**472**
INDEX FULL SCAN on XPKSTOCK	1	118	472

So that is a brief look into how to tune simple SELECT statements. *Try to use explicit columns and try to read columns in index orders if possible, even to the point of reading indexes and not tables.*

6.1.1 A Count of Rows in the Accounts Schema

I want to show a row count of all tables in the Accounts schema I have in my database. If you remember we have already stated that larger tables are more likely to require use of indexes and smaller tables are not. Since the Accounts schema has both large and small tables, SELECT statements and various clauses executed against different tables will very much affect how those different tables should be accessed in the interest of good performance. Current row counts for all tables in the Accounts schema are shown in Figure 6.1.

ⓘ Accounts schema row counts vary throughout this book since the database is continually actively adding rows and occasionally recovered to the initial state shown in Figure 6.1. Relative row counts between tables remain constant.

Figure 6.1 *Row Counts of Accounts Schema Tables*

Table	Rows
CASHBOOK	188,185
CASHBOOKLINE	188,185
CATEGORY	13
COA	55
CUSTOMER	2,694
GENERALLEDGER	752,740
ORDERS	172,304
ORDERSLINE	540,827
PERIOD	60
PERIODSUM	0
POSTING	8
STOCK	118
STOCKMOVEMENT	570,175
STOCKSOURCE	12,083
SUBTYPE	4
SUPPLIER	3,874
TRANSACTIONS	188,185
TRANSACTIONSLINE	570,175
TYPE	6

6.1.2 Filtering with the WHERE Clause

Filtering the results of a SELECT statement with a WHERE clause implies retrieving only a subset of rows from a larger set of rows. The WHERE clause can be used to either include wanted rows, exclude unwanted rows or both.

Once again using the Employees schema, in the following SQL statement we filter rows to include only those rows we want, retrieving only those rows with PROJECTTYPE values starting with the letter "R".

```
SELECT * FROM projecttype WHERE name LIKE 'R%';
```

Now we do the opposite and filter out rows we do not want. We get everything with values not starting with the letter "R".

```
SELECT * FROM projecttype WHERE name NOT LIKE 'R%';
```

How does the WHERE clause affect the performance of a SELECT statement? If the sequence of expression comparisons in a WHERE clause can match an index it should. The WHERE clause in the SELECT statement above does not match an index and thus the whole table will be read. Since the Employees schema ProjectType table is small, having only 11 rows, this is unimportant. However, in the case of the Accounts schema, where many of the tables have large numbers of rows, avoiding full table scans, and forcing index reading is important. Following is a single WHERE clause comparison condition example SELECT statement. We will once again show the cost of the query using the EXPLAIN PLAN command. This query does an exact match on a very large table by applying an exact value to find a single row. Note the unique index scan and the low cost of the query.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stockmovement WHERE stockmovement_id =
        5000;

Query                                          Cost    Rows  Bytes
---------------------------------------------- ------- ----- -----
SELECT STATEMENT  on                             3       1     24
 TABLE ACCESS BY INDEX ROWID on
   STOCKMOVEMENT                                 3       1     24
  INDEX UNIQUE SCAN on
    XPKSTOCKMOVEMENT                             2       1
```

Now let's compare the query above with an example which uses another single column index but searches for many more rows than a single row. This example consists of two queries. The first query gives us the WHERE clause literal value for the second query. The result of the first query is displayed here.

```
SQL> SELECT coa#, COUNT(coa#) "Rows" FROM generalledger
  GROUP BY coa#;

COA#                Rows
------------      ----------------------
30001              310086
40003               66284
41000              173511
50001              169717
60001               33142
```

Now let's look at a query plan for the second query with the WHERE clause filter applied. The second query shown next finds all the rows in one of the groups listed in the result shown above.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger WHERE coa# = 40003;

Query                                          Cost     Rows     Bytes
---------------------------------------------------------------- -------- ----------- -------------
SELECT STATEMENT  on                            493   150548   3914248
 TABLE ACCESS FULL on GENERALLEDGER              493   150548   3914248
```

The query above has an interesting result because the table is fully scanned. This is because the Optimizer considers it more efficient to read the entire table rather than use the index on the COA# column to find specific columns in the table. This is because the WHERE clause will retrieve over 60,000 rows, just shy of 10% of the entire GeneralLedger table. Over 10% is enough to trigger the Optimizer to execute a full table scan.

In comparison to the above query the following two queries read a very small table, the first with a unique index hit, and the second with a full table scan as a result of the range comparison condition (<). In the second query, if the table were much larger possibly the Optimizer would have executed an index range scan and read the index file. However, since the table is small the Optimizer considers reading the entire table as being faster than reading the index to find what could be more than a single row.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM category WHERE category_id = 1;

Query                                        Cost   Rows  Bytes
---------------------------------------------------------------- --------- -------- -----------
SELECT STATEMENT  on                          1      1     12
 TABLE ACCESS BY INDEX ROWID on CATEGORY      1      1     12
  INDEX UNIQUE SCAN on XPKCATEGORY                  1
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM category WHERE category_id < 2;
```

The costs of both index use and the full table scan are the same because the table is small.

```
Query                                          Cost      Rows      Bytes
---------------------------------------------- --------  --------  --------
SELECT STATEMENT  on                              1         1         12
 TABLE ACCESS FULL on CATEGORY                    1         1         12
```

So far we have looked at WHERE clauses containing single comparison conditions. In tables where multiple column indexes exist there are other factors to consider. The following two queries produce exactly the same result. Note the unique index scan on the primary key for both queries. As with the ordering of index columns in the SELECT statement, in previous versions of Oracle it is possible that the same result would not have occurred for the second query. This is because in the past the order of table column comparison conditions absolutely had to match the order of columns in an index. In the past the second query shown would probably have resulted in a full table scan. The Optimizer is now more intelligent in Oracle9*i* Database.

10*g* Oracle Database 10*g* has Optimizer improvements such as less of a need for SQL code statements to be case sensitive.

We had a similar result previously in this chapter using the CHEQUE_ID and SEQ# columns on the CashbookLine table. The same applies to the WHERE clause.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM ordersline WHERE order_id = 3137 AND
        seq# = 1;
```

```
Query                                       Cost     Rows    Bytes
------------------------------------------- -------- -------- --------
SELECT STATEMENT  on                           3        1       17
 TABLE ACCESS BY INDEX ROWID on
   ORDERSLINE                                  3        1       17
  INDEX UNIQUE SCAN on XPKORDERSLINE           2        1
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM ordersline WHERE seq# = 1 AND
        order_id= 3137;
```

```
Query                                          Cost     Rows     Bytes
---------------------------------------------- -------- -------- --------
SELECT STATEMENT  on                              3        1        17
 TABLE ACCESS BY INDEX ROWID on
   ORDERSLINE                                     3        1        17
  INDEX UNIQUE SCAN on XPKORDERSLINE              2        1
```

Let's now try a different variation. The next example query should only use the second column in the composite index on the STOCK_ID and SUPPLIER_ID columns on the StockSource table. What must be done first is to find a StockSource row uniquely identified by both the STOCK_ID and SUPPLIER_ID columns. Let's simply create a unique row. I have not used sequences in the INSERT statements shown because I want to preserve the values of the sequence objects.

ⓘ The names of the columns in the Stock table Stock.MIN and Stock.MAX refer to minimum and maximum Stock item values in the Stock table, not the MIN and MAX Oracle SQL functions.

```
INSERT INTO stock(stock_id, category_id, text, min, max)
      VALUES((SELECT MAX(stock_id)+1 FROM
        stock),1,'text',1,100);

INSERT INTO supplier(supplier_id, name, ticker)
      VALUES((SELECT MAX(supplier_id)+1 FROM supplier)
      ,'name','TICKER');

INSERT INTO stocksource
      VALUES((SELECT MAX(supplier_id) FROM supplier)
      ,(SELECT MAX(stock_id) FROM stock),100.00);
```

The INSERT statements created a single row in the StockSource table with the primary key composite index uniquely identifying the first column, the second column, and the combination of both. We can find those unique values by finding the maximum values for them.

```
SELECT COUNT(stock_id), MAX(stock_id)
FROM stocksource
WHERE stock_id = (SELECT MAX(stock_id) FROM stocksource)
GROUP BY stock_id;

COUNT(STOCK_ID)        MAX(STOCK_ID)
---------------------- ----------------------------------
              1                    119
```

```
SELECT COUNT(supplier_id), MAX(supplier_id)
FROM stocksource
WHERE supplier_id = (SELECT MAX(supplier_id) FROM supplier)
GROUP BY supplier_id;

COUNT(SUPPLIER_ID)  MAX(SUPPLIER_ID)
------------------  ----------------
                 1              3875
```

Now let's attempt that unique index hit on the second column of the composite index in the StockSource table, amongst other combinations.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stocksource WHERE supplier_id = 3875;
```

Something very interesting happens. The foreign key index on the SUPPLIER_ID column is range scanned because the WHERE clause is matched. The composite index is ignored.

```
Query                                          Cost   Rows   Bytes
---------------------------------------------- -----  -----  -----
1. SELECT STATEMENT  on                           2      3     30
2.  TABLE ACCESS BY INDEX ROWID on
      STOCKSOURCE                                 2      3     30
3.   INDEX RANGE SCAN on XFK_SS_SUPPLIER          1      3
```

The following query uses the STOCK_ID column, the first column in the composite index. Once again, even though the STOCK_ID column is the first column in the composite index the Optimizer matches the WHERE clause against the nonunique foreign key index on the STOCK_ID column. Again the result is a range scan.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stocksource WHERE stock_id = 119;
```

```
Query                                          Cost   Rows   Bytes
---------------------------------------------- -----  -----  -----
1. SELECT STATEMENT  on                           8    102    1020
2.  TABLE ACCESS BY INDEX ROWID on
      STOCKSOURCE                                 8    102    1020
3.   INDEX RANGE SCAN on XFK_SS_STOCK             1    102
```

The next query executes a unique index hit on the composite index because the WHERE clause exactly matches the index.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stocksource
      WHERE stock_id = 119 AND supplier_id = 3875;
```

```
Query                                                  Cost   Rows  Bytes
------------------------------------------------------ ------- ------ -----
1. SELECT STATEMENT  on                                   2       1     10
2.  TABLE ACCESS BY INDEX ROWID on
      STOCKSOURCE                                         2       1     10
3.    INDEX UNIQUE SCAN on XPK_STOCKSOURCE                1   12084
```

Let's clean up and delete the unique rows we created with the INSERT statements above. My script executing queries (see Appendix B) on the PLAN_TABLE contains a COMMIT command and thus ROLLBACK will not work.

```
DELETE FROM StockSource WHERE supplier_id = 3875 and
  stock_id = 119;
DELETE FROM supplier WHERE supplier_id = 3875;
DELETE FROM stock WHERE stock_id = 119;
COMMIT;
```

Now let's do something slightly different. The purpose of creating unique stock and supplier items in the StockSource table was to get the best possibility of producing a unique index hit. If we were to select from the StockSource table where more than a single row existed we would once again not get unique index hits. Depending on the number of rows found we could get index range scans or even full table scans.

Firstly, find maximum and minimum counts for stocks duplicated on the StockSource table.

```
SELECT * FROM(
      SELECT supplier_id, COUNT(supplier_id) AS suppliers
      FROM stocksource GROUP BY supplier_id ORDER BY
        suppliers DESC)
WHERE ROWNUM = 1
UNION
SELECT * FROM(
      SELECT supplier_id, COUNT(supplier_id) AS suppliers
      FROM stocksource GROUP BY supplier_id ORDER BY
        suppliers)
WHERE ROWNUM = 1;
```

There are nine suppliers with a SUPPLIER_ID column value of 2711 and one with SUPPLIER_ID column value 2.

```
SUPPLIER_ID                SUPPLIERS
-------------------------- ----------
                         2          1
                      2711          9
```

Both the next two queries perform index range scans. If one of the queries retrieved enough rows, as in the COA# = '40003' previously shown in this chapter, the Optimizer would force a read of the entire table.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stocksource WHERE supplier_id = 2;

Query                                          Cost   Rows   Bytes
---------------------------------------------- -------- -------- ----------
1. SELECT STATEMENT  on                           2      3      30
2.  TABLE ACCESS BY INDEX ROWID on
      STOCKSOURCE                                 2      3      30
3.   INDEX RANGE SCAN on XFK_SS_SUPPLIER          1      3

EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stocksource WHERE supplier_id = 2711;

Query                                          Cost   Rows   Bytes
---------------------------------------------- -------- -------- ----------
1. SELECT STATEMENT  on                           2      3      30
2.  TABLE ACCESS BY INDEX ROWID on
      STOCKSOURCE                                 2      3      30
3.   INDEX RANGE SCAN on XFK_SS_SUPPLIER          1      3
```

So try to always do two things with WHERE clauses. Firstly, *try to match comparison condition column sequences with existing index column sequences, although it is not strictly necessary*. Secondly, always *try to use unique, single-column indexes wherever possible*. A single-column unique index is much more likely to produce exact hits. An exact hit is the fastest access method.

6.1.3 Sorting with the ORDER BY Clause

The ORDER BY clause sorts the results of a query. The ORDER BY clause is always applied after all other clauses are applied, such as the WHERE and GROUP BY clauses. Without an ORDER BY clause in an SQL statement, rows will often be retrieved in the physical order in which they were added to the table. Rows are not always appended to the end of a table as space can be reused. Therefore, physical row order is often useless. Additionally the sequence and content of columns in the SELECT statement, WHERE and GROUP BY clauses can also somewhat determine returned sort order to a certain extent.

In the following example we are sorting on the basis of the content of the primary key index. Since the entire table is being read there is no use of the index. Note the sorting applied to rows retrieved from the table as a result of re-sorting applied by the ORDER BY clause.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id, name FROM customer ORDER BY
        customer_id;
```

```
Query                                  Cost   Rows   Bytes    Sort
-------------------------------------- -----  -----  -------  ------
SELECT STATEMENT  on                     25   2694   67350
 SORT ORDER BY on                        25   2694   67350  205000
  TABLE ACCESS FULL on CUSTOMER           9   2694   67350
```

In the next example the name column is removed from the SELECT statement and thus the primary key index is used. Specifying the CUSTOMER_ID column only in the SELECT statement forces use of the index, not the ORDER BY clause. Additionally there is no sorting because the index is already sorted in the required order. In this case the ORDER BY clause is unnecessary since an identical result would be obtained without it.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id FROM customer ORDER BY
        customer_id;
```

```
Query                                  Cost   Rows   Bytes    Sort
-------------------------------------- -----  -----  -------  ------
SELECT STATEMENT  on                      6   2694   10776
 INDEX FULL SCAN on XPKCUSTOMER           6   2694   10776
```

The next example re-sorts the result by name. Again the whole table is read so no index is used. The results are the same as for the query before the previous one. Again there is physical sorting of the rows retrieved from the table.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id, name FROM customer ORDER BY name;
```

```
Query                                  Cost   Rows   Bytes    Sort
-------------------------------------- -----  -----  -------  ------
SELECT STATEMENT  on                     25   2694   67350
 SORT ORDER BY on                        25   2694   67350  205000
  TABLE ACCESS FULL on CUSTOMER           9   2694   67350
```

The ORDER BY clause will re-sort results.

10g Queries and sorts are now less case sensitive than in previous versions of Oracle Database.

Overriding WHERE with ORDER BY

The following example is interesting because the primary key composite index is used. Note that there is no sorting in the query plan. It is unnecessary to sort since the index scanned is being read in the order required by the ORDER BY clause. In this case the Optimizer ignores the ORDER BY clause. The WHERE clause specifies ORDER_ID only, for which there is a nonunique foreign key index. A nonunique index is appropriate to a range scan using the < range operator as shown in the example. However, the primary key composite index is used to search with, as specified in the ORDER BY clause. Thus the ORDER BY clause effectively overrides the specification of the WHERE clause. The ORDER BY clause is often used to override any existing sorting parameters.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM ordersline WHERE order_id < 10
      ORDER BY order_id, seq#;
```

```
Query                                          Cost     Rows     Bytes
---------------------------------------------------------------------
SELECT STATEMENT  on                              4        3        51
 TABLE ACCESS BY INDEX ROWID on
   ORDERSLINE                                     4        3        51
  INDEX RANGE SCAN on XPKORDERSLINE               3        3
```

The second example excludes the overriding ORDER BY clause. Note how the index specified in the WHERE clause is utilized for an index range scan. Thus in the absence of the ORDER BY clause in the previous example the Optimizer resorts to the index specified in the WHERE clause.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM ordersline WHERE order_id < 10;
```

```
Query                                                  Cost   Rows   Bytes
-----------------------------------------------------  -----  -----  -----
SELECT STATEMENT  on                                      3      3      51
 TABLE ACCESS BY INDEX ROWID on
   ORDERSLINE                                             3      3      51
  INDEX RANGE SCAN on XFK_ORDERLINE_ORDER                 2      3
```

The next example retains the WHERE clause, containing the first column in the primary key index. It also uses an ORDER BY clause containing only the second column in the composite primary key. This query has a higher cost than both the first and second queries shown before. Why? The Optimizer is retrieving based on the WHERE clause and then being overridden by the ORDER BY clause. What is happening is that the ORDER BY is re-sorting the results of the WHERE clause.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM ordersline WHERE order_id < 10
      ORDER BY seq#;

Query                                                  Cost  Rows  Bytes
-----------------------------------------------------  ----  ----  -----
SELECT STATEMENT  on                                      5     3     51
 SORT ORDER BY on                                         5     3     51
  TABLE ACCESS BY INDEX ROWID on ORDERSLINE               3     3     51
   INDEX RANGE SCAN on XFK_ORDERLINE_ORDER                2     3
```

In general, it is difficult to demonstrate the performance tuning aspects of the ORDER BY clause. This is because re-sorting is executed after everything else has completed. The ORDER BY clause should not be allowed to conflict with the best Optimizer performance choices of previous clauses. An ORDER BY clause can be used as a refinement of previous clauses rather than replacing those previous clauses. The WHERE clause will filter rows and the ORDER BY re-sorts those filtered rows. The ORDER BY clause can sometimes persuade the Optimizer to use a less efficient key.

In some older relational databases it was always inadvisable to apply any sorting in the ORDER BY which was already sorted by the WHERE clause. In Oracle Database this is not the case. The Oracle Database Optimizer is now intelligent enough to often be able to utilize the best index for searching. Leaving columns out of the ORDER BY clause because they are already covered in the WHERE clause is not necessarily a sound approach. Additionally various other SELECT statement clauses execute sorting automatically. The GROUP BY and DISTINCT clauses are two examples that

do inherent sorting. Use inherent sorting if possible rather than doubling up with an ORDER BY clause.

So *the ORDER BY clause is always executed after the WHERE clause*. This does not mean that the Optimizer will choose either the WHERE clause or the ORDER BY clause as the best performing factor. *Try not to override the WHERE clause with the ORDER BY clause because the Optimizer may choose a less efficient method of execution based on the ORDER BY clause*.

6.1.4 Grouping Result Sets

The GROUP BY clause can perform some inherent sorting. As with the SELECT statement, WHERE clause and ORDER BY clause, matching of GROUP BY clause column sequences with index column sequences is relevant to SQL code performance.

The first example aggregates based on the non-unique foreign key on the ORDER_ID column. The aggregate is executed on the ORDER_ID column into unique values for that ORDER_ID. The foreign key index is the best performing option.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT order_id, COUNT(order_id) FROM ordersline
      GROUP BY order_id;
```

The foreign key index is already sorted in the required order. The NOSORT content in the SORT GROUP BY NOSORT on clause implies no sorting is required using the GROUP BY clause.

```
Query                                    Cost       Rows       Bytes
---------------------------------------- -------- ---------- ----------
SELECT STATEMENT  on                       26     172304      861520
 SORT GROUP BY NOSORT on                   26     172304      861520
  INDEX FULL SCAN on
    XFK_ORDERLINE_ORDER                    26     540827     2704135
```

The next example uses both columns in the primary key index and thus the composite index is a better option. However, since the composite index is much larger in both size and rows the cost is much higher.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT order_id, seq#, COUNT(order_id) FROM ordersline
      GROUP BY order_id, seq#;
```

```
Query                                          Cost      Rows     Bytes
---------------------------------------------- --------  -------  -------
SELECT STATEMENT  on                           1217      540827   4326616
 SORT GROUP BY NOSORT on                       1217      540827   4326616
  INDEX FULL SCAN on XPKORDERSLINE             1217      540827   4326616
```

In the next case we reverse the order of the columns in the GROUP BY sequence. As you can see there is no effect on cost since the Optimizer manages to match against the primary key composite index.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT order_id, seq#, COUNT(order_id) FROM ordersline
      GROUP BY seq#, order_id;
```

```
Query                                          Cost      Rows     Bytes
---------------------------------------------- --------  -------  -------
SELECT STATEMENT  on                           1217      540827   4326616
 SORT GROUP BY NOSORT on                       1217      540827   4326616
  INDEX FULL SCAN on XPKORDERSLINE             1217      540827   4326616
```

Sorting with the GROUP BY Clause

This example uses a non-indexed column to aggregate. Thus the whole table is accessed. Note that NOSORT is no longer included in the SORT GROUP BY clause in the query plan. The GROUP BY clause is now performing sorting on the AMOUNT column.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT amount, COUNT(amount) FROM ordersline
      GROUP BY amount;
```

```
Query                                    Cost      Rows     Bytes     Sort
---------------------------------------- --------  -------  --------  -------
SELECT STATEMENT  on                     4832      62371    374226
 SORT GROUP BY on                        4832      62371    374226    7283000
  TABLE ACCESS FULL on
    ORDERSLINE                           261       540827   3244962
```

Let's examine GROUP BY clause sorting a little further. Sometimes it is possible to avoid sorting forced by the ORDER BY clause by ordering column names in the GROUP BY clause. Rows will be sorted based on the contents of the GROUP BY clause.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT amount, COUNT(amount) FROM ordersline
      GROUP BY amount
      ORDER BY amount;
```

In this case the ORDER BY clause is ignored.

```
Query                                          Cost    Rows    Bytes
---------------------------------------------- ------- ------- -------
1. SELECT STATEMENT  on                        6722    62371   374226
2.  SORT GROUP BY on                           6722    62371   374226
3.   TABLE ACCESS FULL on ORDERSLINE           1023    540827  3244962
```

Inherent sorting in the GROUP BY clause can sometimes be used to avoid extra sorting using an ORDER BY clause.

Using DISTINCT

DISTINCT retrieves the first value from a repeating group. When there are multiple repeating groups DISTINCT will retrieve the first row from each group. Therefore, DISTINCT will always require a sort. DISTINCT can operate on a single or multiple columns. The first example executes the sort in order to find the first value in each group. The second example has the DISTINCT clause removed and does not execute a sort. As a result the second example has a much lower cost. DISTINCT will sort regardless.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT DISTINCT(stock_id) FROM stockmovement;
```

```
Query                                    Cost    Rows    Bytes
---------------------------------------- ------- ------- -------
SELECT STATEMENT  on                      704     118     472
 SORT UNIQUE on                          704     118     472
  INDEX FAST FULL SCAN on XFK_SM_STOCK     4     570175  2280700
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT stock_id FROM stockmovement;
```

```
Query                                  Cost    Rows    Bytes
-------------------------------------- ------- ------- -------
SELECT STATEMENT  on                    4      570175  2280700
 INDEX FAST FULL SCAN on
   XFK_SM_STOCK                         4      570175  2280700
```

As far as performance tuning is concerned *DISTINCT will always require a sort*. Sorting slows performance.

The HAVING Clause

Using the COUNT function as shown in the first two examples there is little difference in performance. The slight difference is due to the

application of the filter on the HAVING clause, allowing return of fewer rows. The mere act of using the HAVING clause to return fewer rows helps performance.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id, COUNT(order_id) FROM orders
      GROUP BY customer_id;
```

```
Query                                  Cost       Rows       Bytes
-------------------------------------------------------------------
SELECT STATEMENT  on                    298       2693        5386
 SORT GROUP BY on                       298       2693        5386
  TABLE ACCESS FULL on ORDERS           112     172304      344608
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id, COUNT(order_id) FROM orders
      GROUP BY customer_id
      HAVING customer_id < 10;
```

```
Query                                  Cost       Rows       Bytes
-------------------------------------------------------------------
SELECT STATEMENT  on                    296         10          20
 FILTER  on
  SORT GROUP BY on                      296         10          20
   TABLE ACCESS FULL on ORDERS          112     172304      344608
```

However, the next two examples using the SUM function as opposed to COUNT have a much bigger difference in cost. This is because the COUNT function is faster, especially when counting on indexes or using the COUNT(*) function with the asterisk option. The COUNT function will be demonstrated in detail later in this chapter. There is a lot of processing that the SUM function does which the COUNT function does not.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id, SUM(order_id) FROM orders
      GROUP BY customer_id;
```

```
Query                          Cost     Rows     Bytes      Sort
-----------------------------------------------------------------
SELECT STATEMENT  on           1383     2693     18851
 SORT GROUP BY on              1383     2693     18851   2827000
  TABLE ACCESS FULL on
   ORDERS                        112   172304   1206128
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id, SUM(order_id) FROM orders
      GROUP BY customer_id
      HAVING customer_id < 10;
```

```
Query                                    Cost     Rows     Bytes     Sort
---------------------------------------- -------- -------- --------- ---------
SELECT STATEMENT  on                      366        10        70
 FILTER  on
  SORT GROUP BY on                        366        10        70
   TABLE ACCESS FULL on ORDERS            112    172304   1206128
```

(10g) *The Spreadsheet Clause*

The spreadsheet clause extends the HAVING clause and allows display of data into multiple dimensions allowing calculations between rows much like a spreadsheet program can provide. The spreadsheet clause provides additional OLAP type functionality and is more applicable to data warehousing as opposed to Internet OLTP databases. However, using the spreadsheet clause can in some cases possibly reduce the number of tables in mutable joins and remove the need for set operators such as UNION, INTERSECT, and MINUS to merge multiple queries together.

The HAVING clause filter can help performance because it filters, allowing the return and processing of fewer rows. The *HAVING clause filtering* shown in the query plans above shows that HAVING clause filtering *is always executed after the GROUP BY sorting process*.

ROLLUP, CUBE, and GROUPING SETS

The ROLLUP, CUBE, and GROUPING SETS clauses can be used to create breaks and subtotals for groups. The GROUPING SETS clause can be used to restrict the results of ROLLUP and CUBE clauses. Before the advent of ROLLUP and CUBE, producing the same types of results would involve extremely complex SQL statements, probably with the use of temporary tables or perhaps use of PL/SQL as well. ROLLUP, CUBE, and GROUPING SETS are more applicable to reporting and data warehouse functionality.

(10g) The spreadsheet clause extension to the HAVING clause is similar in function.

The following examples simply show the use of the ROLLUP, CUBE, and GROUPING SETS clauses.

```
SELECT type, subtype, SUM(balance+ytd)FROM coa
      GROUP BY type, subtype;

SELECT type, subtype, SUM(balance+ytd)FROM coa
      GROUP BY ROLLUP (type, subtype);

SELECT type, subtype, SUM(balance+ytd)FROM coa
      GROUP BY CUBE (type, subtype);

SELECT type, subtype, SUM(balance+ytd)FROM coa
      GROUP BY GROUPING SETS ((type, subtype), (type),
        (subtype));
```

In general, *the GROUP BY clause can perform some sorting* if it matches indexing. *Filtering aggregate results with the HAVING clause can help to increase performance* by filtering aggregated results of the GROUP BY clause.

6.1.5 The FOR UPDATE Clause

The FOR UPDATE clause is a nice feature of SQL since it allows locking of selected rows during a transaction. There are rare circumstances where rows selected should be locked since there are dependent following changes in a single transaction, requiring selected data to remain the same during the course of that transaction.

```
SELECT …
   FOR UPDATE OF [ [schema.]table.]column [, … ] ]
     [ NOWAIT | WAIT n ]
```

Note the two WAIT and NOWAIT options in the preceding syntax. When a lock is encountered NOWAIT forces an abort. The WAIT option will force a wait for a number of seconds. The default simply waits until a row is available.

It should be obvious that with respect to tuning and concurrent multiuser capability of applications the FOR UPDATE clause should be avoided if possible. Perhaps the data model could be too granular thus necessitating the need to lock rows in various tables during the course of a transaction across multiple tables. Using the FOR UPDATE clause is not good for the efficiency of SQL code in general due to potential locks and possible resulting waits for and by other concurrently executing transactions.

6.2 Using Functions

The most relevant thing to say about functions is that they should not be used where you expect an SQL statement to use an index. There are function-based indexes of course. A function-based index contains the resulting value of an expression. An index search against that function-based index will search the index for the value of the expression.

Let's take a quick look at a few specific functions.

6.2.1 The COUNT Function

For older versions of Oracle Database the COUNT function has been recommended as performing better when used in different ways. Prior to Oracle9*i* Database the COUNT(*) function using the asterisk was the fastest form because the asterisk option was specifically tuned to avoid any sorting. Let's take a look at each of four different methods and show that they are all the same using both the EXPLAIN PLAN command and time testing. We will use the GeneralLedger table in the Accounts schema since it has the largest number of rows.

Notice how all the query plans for all the four following COUNT function options are identical. Additionally there is no sorting on anything but the resulting single row produced by the COUNT function, the sort on the aggregate.

Using the asterisk:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT COUNT(*) FROM generalledger;

Query                                                    Cost        Rows
-------------------------------------------------------- --------  -------------
1. SELECT STATEMENT  on                                   382            1
2.  SORT AGGREGATE on                                                    1
3.    INDEX FAST FULL SCAN on
       XPK_GENERALLEDGER                                  382       752825
```

Forcing the use of a unique index:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT COUNT(generalledger_id) FROM
        generalledger;
```

```
Query                                                    Cost     Rows
-------------------------------------------------------- -------  ------------
1. SELECT STATEMENT  on                                   382         1
2.  SORT AGGREGATE on                                                 1
3.   INDEX FAST FULL SCAN on XPK_GENERALLEDGER            382    752825
```

Using a constant value:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT COUNT(1) FROM generalledger;
```

```
Query                                                    Cost     Rows
-------------------------------------------------------- -------  ------------
1. SELECT STATEMENT  on                                   382         1
2.  SORT AGGREGATE on                                                 1
3.   INDEX FAST FULL SCAN on XPK_GENERALLEDGER            382    752825
```

Using a nonindexed column:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT COUNT(dr) FROM generalledger;
```

```
Query                                                    Cost     Rows
-------------------------------------------------------- -------  ------------
1. SELECT STATEMENT  on                                   382         1
2.  SORT AGGREGATE on                                                 1
3.   INDEX FAST FULL SCAN on                             382    752825
       XPK_GENERALLEDGER
```

Now with time testing, below I have simply executed the four COUNT function options with SET TIMING set to ON in SQL*Plus. Executing these four SQL statements twice will assure that all data is loaded into memory and that consistent results are obtained.

```
SQL> SELECT COUNT(*) FROM generalledger;

  COUNT(*)
----------
    752741

Elapsed: 00:00:01.01

SQL> SELECT COUNT(generalledger_id) FROM generalledger;

COUNT(GENERALLEDGER_ID)
-----------------------
                 752741

Elapsed: 00:00:01.01

SQL> SELECT COUNT(1) FROM generalledger;
```

```
   COUNT(1)
-----------
     752741

Elapsed: 00:00:01.01
SQL> SELECT COUNT(dr) FROM generalledger;

  COUNT(DR)
-----------
     752741

Elapsed: 00:00:01.01
```

As you can see from the time tests above, the COUNT function will perform the same no matter which method is used. In the latest version of Oracle Database different forms of the COUNT function will perform identically. No form of the COUNT function is better tuned than any other. *All forms of the COUNT function perform the same; using an asterisk, a constant or a column, regardless of column indexing, the primary key index is always used.*

6.2.2 The DECODE Function

DECODE can be used to replace composite SQL statements using a set operator such as UNION. The Accounts Stock table has a QTYONHAND column. This column denotes how many items of a particular stock item are currently in stock. Negative QTYONHAND values indicate that items have been ordered by customers but not yet received from suppliers.

The first example below uses four full reads of the Stock table and concatenates the results together using UNION set operators.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
SELECT stock_id||' Out of Stock' FROM stock WHERE
  qtyonhand <=0
UNION
SELECT stock_id||' Under Stocked' FROM stock
      WHERE qtyonhand BETWEEN 1 AND min-1
UNION
SELECT stock_id||' Stocked' FROM stock
      WHERE qtyonhand BETWEEN min AND max
UNION
SELECT stock_id||' Over Stocked' FROM stock
      WHERE qtyonhand > max;
```

```
Query                                          Pos  Cost    Rows    Bytes
---------------------------------------------- ---- ------- ------- -----------
SELECT STATEMENT  on                            12    12     123     1543
 SORT UNIQUE on                                  1    12     123     1543
  UNION-ALL  on                                  1
   TABLE ACCESS FULL on STOCK                    1     1       4       32
   TABLE ACCESS FULL on STOCK                    2     1       1       11
   TABLE ACCESS FULL on STOCK                    3     1      28      420
   TABLE ACCESS FULL on STOCK                    4     1      90     1080
```

This second example replaces the UNION set operators and the four full table scan reads with a single full table scan using nested DECODE functions. DECODE can be used to improve performance.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT stock_id||' '||
      DECODE(SIGN(qtyonhand)
            ,-1,'Out of Stock',0,'Out of Stock'
            ,1,DECODE(SIGN(qtyonhand-min)
                     ,-1,'Under Stocked',0,'Stocked'
                     ,1,DECODE(sign(qtyonhand-max)
                               ,-1,'Stocked',0,'Stocked'
                                 ,1,'Over Stocked'
                     )
            )
       ) FROM stock;
```

```
Query                                          Pos  Cost    Rows    Bytes
---------------------------------------------- ---- ------- ------- -----------
SELECT STATEMENT  on                            1     1     118     1770
 TABLE ACCESS FULL on STOCK                     1     1     118     1770
```

Using the DECODE function as a replacement for multiple query set operators is good for performance but should only be used in extreme cases such as the UNION clause joined SQL statements shown previously.

6.2.3 Datatype Conversions

Datatype conversions are a problem and will conflict with existing indexes unless function-based indexes are available and can be created. Generally, if a function is executed in a WHERE clause, or anywhere else that can utilize an index, a full table scan is likely. This leads to inefficiency. There is some capability in Oracle SQL for implicit datatype conversion but often use of functions in SQL

statements will cause the Optimizer to miss the use of indexes and perform poorly.

The most obvious datatype conversion concerns dates. Date fields in all the databases I have used are stored internally as a Julian number. A Julian number or date is an integer value from a database-specific date measured in seconds. When retrieving a date value in a tool such as SQL*Plus there is usually a default date format. The internal date value is converted to that default format. The conversion is implicit, automatic, and transparent.

```
SELECT SYSDATE, TO_CHAR(SYSDATE,'J') "Julian" FROM DUAL;

  SYSDATE         Julian
---------------------  ---------------
03-MAR-03         2452702
```

Now for the sake of demonstration I will create an index on the GeneralLedger DTE column.

```
CREATE INDEX ak_gl_dte ON GENERALLEDGER(DTE);
```

Now obviously it is difficult to demonstrate an index hit with a key such as this because the date is a datestamp as well as a simple date. A simple date format such as MM/DD/YYYY excludes a timestamp. Simple dates and datestamps (timestamps) are almost impossible to match. Thus I will use SYSDATE in order to avoid a check against a simple formatted date. Both the GeneralLedger DTE column and SYSDATE are timestamps since the date column in the table was created using SYSDATE-generated values. We are only trying to show Optimizer query plans without finding rows.

The first example hits the new index I created and has a very low cost.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger WHERE dte = SYSDATE;

Query                                          Cost    Rows     Bytes
---------------------------------------------------------------------
SELECT STATEMENT  on                              2     593     15418
 TABLE ACCESS BY INDEX ROWID on
   GENERALLEDGER                                  2     593     15418
  INDEX RANGE SCAN on AK_GL_DTE                   1     593

EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger
      WHERE TO_CHAR(dte, 'YYYY/MM/DD') = '2002/08/21';
```

This second example does not hit the index because the TO_CHAR datatype conversion is completely inconsistent with the datatype of the index. As a result the cost is much higher.

```
Query                                          Cost    Rows     Bytes
---------------------------------------------- ------- -------- -------------
SELECT STATEMENT  on                            493    7527     195702
 TABLE ACCESS FULL on GENERALLEDGER              493    7527     195702
```

Another factor to consider with datatype conversions is making sure that datatype conversions are not placed onto columns. Convert literal values not part of the database if possible. In order to demonstrate this I am going to add a zip code column to my Supplier table, create an index on that zip code column and regenerate statistics for the Supplier table. I do not need to add values to the zip code column to prove my point.

```
ALTER TABLE supplier ADD(zip NUMBER(5));
CREATE INDEX ak_sp_zip ON supplier(zip);
ANALYZE TABLE supplier COMPUTE STATISTICS;
```

Now we can show two examples. The first uses an index because there is no datatype conversion on the column in the table and the second reads the entire table because the conversion is on the column.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM supplier WHERE zip = TO_NUMBER('94002');

Query                                          Cost    Rows     Bytes
---------------------------------------------- ------- -------- -------------
SELECT STATEMENT  on                              1       1        142
 TABLE ACCESS BY INDEX ROWID on SUPPLIER          1       1        142
  INDEX RANGE SCAN on AK_SP_ZIP                   1       1

EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM supplier WHERE TO_CHAR(zip) = '94002';

Query                                          Cost    Rows     Bytes
---------------------------------------------- ------- -------- -------------
SELECT STATEMENT  on                             13       1        142
 TABLE ACCESS FULL on SUPPLIER                   13       1        142
```

Oracle SQL does not generally allow implicit type conversions but there is some capacity for automatic conversion of strings to integers, if a string contains an integer value. Using implicit type conversions is a very bad programming practice and is not recommended. A programmer should never rely on another tool to do their job for them. Explicit coding is less likely to meet with potential errors in the future.

It is better to be precise since the computer will always be precise and do exactly as you tell it to do. Implicit type conversion is included in Oracle SQL for ease of programming. Ease of program coding is a top-down application to database design approach, totally contradictory to database tuning. Using a database from the point of view of how the application can most easily be coded is not favorable to eventual production performance. Do not use implicit type conversions. As can be seen in the following examples implicit type conversions do not appear to make any difference to Optimizer costs.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM supplier WHERE supplier_id = 3801;

Query                                          Cost  Rows  Bytes
---------------------------------------------- ----- ----- -----
1. SELECT STATEMENT  on                           2     1   142
2.  TABLE ACCESS BY INDEX ROWID on SUPPLIER       2     1   142
3.   INDEX UNIQUE SCAN on XPK_SUPPLIER            1  3874

EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM supplier WHERE supplier_id = '3801';

Query                                          Cost  Rows  Bytes
---------------------------------------------- ----- ----- -----
1. SELECT STATEMENT  on                           2     1   142
2.  TABLE ACCESS BY INDEX ROWID on SUPPLIER       2     1   142
3.   INDEX UNIQUE SCAN on XPK_SUPPLIER            1  3874
```

In short, *try to avoid using any type of data conversion function in any part of an SQL statement which could potentially match an index*, especially if you are trying to assist performance by matching appropriate indexes.

6.2.4 Using Functions in Queries

Now let's expand on the use of functions by examining their use in all of the clauses of a SELECT statement.

Functions in the SELECT Statement

Firstly, let's put a datatype conversion into a SELECT statement, which uses an index. As we can see in the two examples below, use of the index is not affected by the datatype conversion placed into the SELECT statement.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id FROM customer;

Query                                          Cost    Rows     Bytes
---------------------------------------------- ------- -------- -------------
SELECT STATEMENT  on                              1    2694     10776
 INDEX FAST FULL SCAN on XPKCUSTOMER              1    2694     10776

EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT TO_CHAR(customer_id) FROM customer;

Query                                          Cost    Rows     Bytes
---------------------------------------------- ------- -------- -------------
SELECT STATEMENT  on                              1    2694     10776
 INDEX FAST FULL SCAN on XPKCUSTOMER              1    2694     10776
```

Functions in the WHERE Clause

Now let's examine the WHERE clause. In the two examples below the only difference is in the type of index scan utilized. Traditionally the unique index hit produces an exact match and it should be faster. A later chapter will examine the difference between these two types of index reads.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id FROM customer WHERE
        customer_id = 100;

Query                                          Cost    Rows     Bytes
---------------------------------------------- ------- -------- -------------
SELECT STATEMENT  on                              1       1         4
 INDEX UNIQUE SCAN on XPKCUSTOMER                 1       1         4

EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT customer_id FROM customer
      WHERE TO_CHAR(customer_id) = '100';

Query                                          Cost    Rows     Bytes
---------------------------------------------- ------- -------- -------------
SELECT STATEMENT  on                              1       1         4
 INDEX FAST FULL SCAN on XPKCUSTOMER              1       1         4
```

Functions in the ORDER BY Clause

The ORDER BY clause can utilize indexing well, as already seen in this chapter, as long as WHERE clause index matching is not compromised. Let's keep it simple. Looking at the following two examples it should

suffice to say that it might be a bad idea to include functions in ORDER BY clauses. An index is not used in the second query and consequently the cost is much higher.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger ORDER BY coa#;
```

```
Query                                    Cost    Rows    Bytes  Sort
---------------------------------------- ------ ------- -------- ----------
SELECT STATEMENT  on                      826 752740 19571240
 TABLE ACCESS BY INDEX ROWID on GL        826 752740 19571240
  INDEX FULL SCAN on XFK_GL_COA#           26 752740
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger ORDER BY TO_CHAR(coa#);
```

```
Query                          Cost     Rows      Bytes       Sort
------------------------------ --------- ------- -------- ----------
SELECT STATEMENT  on          19070   752740   19571240
 SORT ORDER BY on             19070   752740   19571240   60474000
  TABLE ACCESS FULL on
    GENERALLEDGER               493   752740   19571240
```

Here is an interesting twist to using the same datatype conversion in the above two examples but with the conversion in the SELECT statement and setting the ORDER BY clause to sort by position rather than using the TO_CHAR(COA#) datatype conversion. The reason why this example is lower in cost than the second example is because the conversion is done on selection and ORDER BY re-sorting is executed after data retrieval. In other words, in this example the ORDER BY clause does not affect the data access method.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT TO_CHAR(coa#), dte, dr cr FROM generalledger
        ORDER BY 1;
```

```
Query                          Cost     Rows      Bytes       Sort
------------------------------ --------- ------- -------- ----------
SELECT STATEMENT  on          12937   752740   13549320
 SORT ORDER BY on             12937   752740   13549320   42394000
  TABLE ACCESS FULL on
    GENERALLEDGER               493   752740   13549320
```

Functions in the GROUP BY Clause

Using functions in GROUP BY clauses will slow performance as shown in the following two examples.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT order_id, COUNT(order_id) FROM ordersline
      GROUP BY order_id;
```

Query	Cost	Rows	Bytes
SELECT STATEMENT on	**26**	172304	861520
SORT GROUP BY **NOSORT** on	26	172304	861520
INDEX FULL SCAN on XFK_ORDERLINE_ORDER	26	540827	2704135

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT TO_CHAR(order_id), COUNT(order_id) FROM
        ordersline
      GROUP BY TO_CHAR(order_id);
```

Query	Cost	Rows	Bytes	Sort
SELECT STATEMENT on	**3708**	172304	861520	
SORT GROUP BY on	3708	172304	861520	8610000
INDEX FAST FULL SCAN on XFK_ORDERLINE_ORDER	4	540827	2704135	

When using functions in SQL statements it is best to keep the functions away from any columns involving index matching.

6.3 Pseudocolumns

There are some ways in which pseudocolumns can be used to increase performance.

6.3.1 Sequences

A sequence is often used to create unique integer identifiers as primary keys for tables. A sequence is a distinct database object and is accessed as sequence.NEXTVAL and sequence.CURRVAL. Using the Accounts schema Supplier table we can show how a sequence is an efficient method in this case.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      INSERT INTO supplier (supplier_id, name, ticker)
      VALUES(supplier_seq.NEXTVAL,'A new supplier', 'TICK');
```

Query	Cost	Rows	Bytes
INSERT STATEMENT on	**1**	11	176
SEQUENCE on SUPPLIER_SEQ			

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      INSERT INTO supplier (supplier_id, name, ticker)
      VALUES((SELECT MAX(supplier_id)+1
      FROM supplier), 'A new supplier', 'TICK');

Query                                          Cost     Rows    Bytes
---------------------------------------------- -------- -------- -----------
INSERT STATEMENT  on                              1       11      176
```

The query plan above is the same. There is a problem with it. Notice that a subquery is used to find the next SUPPLIER_ID value. This subquery is not evident in the query plan. Let's do a query plan for the subquery as well.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT MAX(supplier_id)+1 FROM supplier;

Query                                          Cost     Rows    Bytes
---------------------------------------------- -------- -------- -----------
1. SELECT STATEMENT  on                           2        1        3
2.  SORT AGGREGATE on                                      1        3
3.   INDEX FULL SCAN (MIN/MAX)
       on XPK_SUPPLIER                            2     3874    11622
```

We can see that the subquery will cause extra work. Since the query plan seems to have difficulty with subqueries it is difficult to tell the exact cost of using the subquery. *Use sequences for unique integer identifiers; they are centralized, more controllable, more easily maintained, and perform better than other methods of counting.*

6.3.2 ROWID Pointers

A ROWID is a logically unique database pointer to a row in a table. When a row is found using an index the index is searched. After the row is found in the index the ROWID is extracted from the index and used to find the exact logical location of the row in its respective table. Accessing rows using the ROWID pseudocolumn is probably the fastest row access method in Oracle Database since it is a direct pointer to a unique address. The downside about ROWID pointers is that they do not necessarily point at the same rows in perpetuity because they are relative to datafile, tablespace, block, and row. These values can change. Never store a ROWID in a table column as a pointer to other tables or rows if data or structure will be changing in the database. If ROWID pointers can be used for data access they can be blindingly fast but are not recommended by Oracle Corporation.

6.3.3 ROWNUM

A ROWNUM is a row number or a sequential counter representing the order in which a row is returned from a query. ROWNUM can be used to restrict the number of rows returned. There are numerous interesting ways in which ROWNUM can be used. For instance, the following example allows creation of a table from another, including all constraints but excluding any rows. This is a useful and fast method of making an empty copy of a very large table.

```
CREATE TABLE tmp AS SELECT * FROM generalledger WHERE
  ROWNUM < 1;
```

One point to note is as in the following example. A ROWNUM restriction is applied in the WHERE clause. Since the ORDER BY clause occurs after the WHERE clause the ROWNUM restriction is not applied to the sorted output. The solution to this problem is the second example.

```
SELECT * FROM customer WHERE ROWNUM < 25 ORDER BY
  name;

SELECT * FROM (SELECT * FROM customer ORDER BY name) WHERE
  ROWNUM < 25;
```

6.4 Comparison Conditions

Different comparison conditions can have sometimes vastly different effects on the performance of SQL statements. Let's examine each in turn with various options and recommendations for potential improvement. The comparison conditions are listed here.

- Equi, anti, and range
  - expr { [!]= | > | < | <= | >= } expr
  - expr [NOT] BETWEEN expr AND expr
- LIKE pattern matching
  - expr [NOT] LIKE expr
- Set membership
  - expr [NOT] IN expr
  - expr [NOT] EXISTS expr

10g IN is now called an IN rather than a set membership condition in order to limit confusion with object collection MEMBER conditions.

- Groups
- expr [= | != | > | < | >= | <=] [ANY | SOME | ALL] expr

6.4.1 Equi, Anti, and Range

Using an equals sign (equi) is the fastest comparison condition if a unique index exists. Any type of anti comparison such as != or NOT is looking for what is not in a table and thus must read the entire table; sometimes full index scans can be used. Range comparisons scan indexes for ranges of rows. Let's look at some examples.

This example does a unique index hit; using the equals sign an exact hit single row is found.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger WHERE generalledger_id =
        100;

Query                                          Cost  Rows  Bytes
---------------------------------------------- ----- ----- -----
SELECT STATEMENT  on                              3     1    26
 TABLE ACCESS BY INDEX ROWID on
   GENERALLEDGER                                  3     1    26
  INDEX UNIQUE SCAN on XPKGENERALLEDGER           2     1
```

The anti (!=) comparison finds everything but the single row specified and thus must read the entire table.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger WHERE generalledger_id !=
        100;

Query                                Pos   Cost     Rows      Bytes
------------------------------------ ----- ------ -------- ----------
SELECT STATEMENT  on                 493    493   752739   19571214
 TABLE ACCESS FULL on GENERAL          1    493   752739   19571214
```

In the next case using the range (<) comparison searches a range of index values rather than a single unique index value.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger WHERE generalledger_id < 10;

Query                                          Cost   Rows   Bytes
---------------------------------------------- ------ ------ -----
SELECT STATEMENT  on                              4      1     26
 TABLE ACCESS BY INDEX ROWID on
   GENERALLEDGE                                   4      1     26
  INDEX RANGE SCAN on XPKGENERALLEDGER            3      1
```

In the next example the whole table is read rather than using an index range scan because most of the table will be read and thus the Optimizer considers reading the table as being faster.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger WHERE generalledger_id >=
        100;

Query                                 Pos   Cost    Rows      Bytes
------------------------------------- ----- ------ -------- ----------
SELECT STATEMENT  on                  493    493   752740   19571240
 TABLE ACCESS FULL on GENERAL           1    493   752740   19571240
```

Here the BETWEEN comparison causes a range scan on an index because the range of rows is small enough to not warrant a full table scan.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger
      WHERE generalledger_id BETWEEN 100 AND 200;

Query                                          Cost   Rows   Bytes
---------------------------------------------- ------ ------ -----
SELECT STATEMENT  on                              4      1     26
 TABLE ACCESS BY INDEX ROWID on
   GENERALLEDGE                                   4      1     26
  INDEX RANGE SCAN on XPKGENERALLEDGER            3      1
```

6.4.2 LIKE Pattern Matching

The approach in the query plan used by the Optimizer will depend on how many rows are retrieved and how the pattern match is constructed.

This query finds one row.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM supplier WHERE name like '24/7 Real
        Media, Inc.';
```

```
Query                                          Cost   Rows   Bytes
---------------------------------------------- ------ ------ -----
SELECT STATEMENT  on                              2      1    142
 TABLE ACCESS BY INDEX ROWID on SUPPLIER          2      1    142
  INDEX UNIQUE SCAN on AK_SUPPLIER_NAME           1      1
```

This query also retrieves a single row but there is a wildcard pattern match and thus a full table scan is the result.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM supplier WHERE name LIKE '21st%';

Query                                      Cost   Rows   Bytes
------------------------------------------ ------ ------ -----
SELECT STATEMENT  on                         13    491   69722
 TABLE ACCESS FULL on SUPPLIER               13    491   69722
```

The next query finds almost 3,000 rows and thus a full scan of the table results regardless of the exactness of the pattern match.

ⓘ A pattern match using a % full wildcard pattern matching character anywhere in the pattern matching string will usually produce a full table scan.

```
SQL> SELECT COUNT(*) FROM supplier WHERE name LIKE '%a%';

COUNT(*)
--------
    2926

EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM supplier WHERE name LIKE '%a%';

Query                                      Cost   Rows   Bytes
------------------------------------------ ------ ------ -----
SELECT STATEMENT  on                         13    194   27548
 TABLE ACCESS FULL on SUPPLIER               13    194   27548
```

In general, since LIKE will match patterns which are in no way related to indexes, LIKE will usually read an entire table.

6.4.3 Set Membership

IN should be used to test against literal values and EXISTS is often used to create a correlation between a calling query and a subquery. IN is best used as a pre-constructed set of literal values. IN will cause a subquery to be executed in its entirety before passing the

result back to the calling query. EXISTS will stop once a result is found.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM coa WHERE type IN ('A','L','I','E');
```

```
Query                                          Cost     Rows      Bytes
---------------------------------------------- -------- --------- -----------
SELECT STATEMENT  on                              1       38        950
 TABLE ACCESS FULL on COA                         1       38        950
```

There are two advantages to using EXISTS over using IN. The first advantage is the ability to pass values from a calling query to a subquery, never the other way around, creating a correlated query. The correlation allows EXISTS the use of indexes between calling query and subquery, particularly in the subquery. The second advantage of EXISTS is, unlike IN, which completes a subquery regardless, EXISTS will halt searching when a value is found. Thus the subquery can be partially executed, reading fewer rows.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM coa WHERE EXISTS
            (SELECT type FROM type WHERE type = coa.type);
```

```
Query                                          Cost    Rows     Bytes
---------------------------------------------- -------- -------- -----------
SELECT STATEMENT  on                              1      55      1485
 NESTED LOOPS SEMI on                             1      55      1485
  TABLE ACCESS FULL on COA                        1      55      1375
  INDEX UNIQUE SCAN on XPKTYPE                           6        12
```

Now let's compare the use of IN versus the use of EXISTS. The next two examples both use indexes and have the same result. The reason why IN is the same cost as EXISTS is because the query contained within the IN subquery matches an index based on the single column it selects.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT stock_id FROM stock s WHERE EXISTS
      (SELECT stock_id FROM stockmovement WHERE stock_id =
        s.stock_id);
```

```
Query                                          Cost     Rows      Bytes
---------------------------------------------- -------- --------- -----------
SELECT STATEMENT  on                            119       118       944
 NESTED LOOPS SEMI on                           119       118       944
  INDEX FULL SCAN on XPKSTOCK                     1       118       472
  INDEX RANGE SCAN on XFK_SM_STOCK                1    570175   2280700
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT stock_id FROM stock WHERE stock_id IN
      (SELECT stock_id FROM stockmovement);
```

Query	Cost	Rows	Bytes
SELECT STATEMENT on	**119**	118	944
NESTED LOOPS SEMI on	119	118	944
INDEX FULL SCAN on XPKSTOCK	1	118	472
INDEX RANGE SCAN on XFK_SM_STOCK	1	570175	2280700

Now let's do some different queries to show a very distinct difference between IN and EXISTS. Note how the first example is much lower in cost than the second. This is because the second option cannot match indexes and executes two full table scans.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stockmovement sm WHERE EXISTS
      (SELECT * FROM stockmovement
            WHERE stockmovement_id = sm.stockmovement_id);
```

Query	Cost	Rows	Bytes	Sort
SELECT STATEMENT on	**8593**	570175	16535075	
MERGE JOIN SEMI on	8593	570175	16535075	
TABLE ACCESS BY INDEX ROWID on SM	3401	570175	13684200	
INDEX FULL SCAN on XPKSTOCKMOVEMENT	1071	570175		
SORT UNIQUE on	5192	570175	2850875	13755000
INDEX FAST FULL SCAN on XPKSTMOVE	163	570175	2850875	

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stockmovement sm WHERE qty IN
            (SELECT qty FROM stockmovement);
```

Query	Cost	Rows	Bytes	Sort
SELECT STATEMENT on	**16353**	570175	15964900	
MERGE JOIN SEMI on	16353	570175	15964900	
SORT JOIN on	11979	570175	13684200	45802000
TABLE ACCESS FULL on STOCKMOVEMENT	355	570175	13684200	
SORT UNIQUE on	4374	570175	2280700	13755000
TABLE ACCESS FULL on STOCKMOVEMENT	355	570175	2280700	

Now let's go yet another step further and restrict the calling query to a single row result. What this will do is ensure that EXISTS

has the best possible chance of passing a single row identifier into the subquery, thus ensuring a unique index hit in the subquery. The StockMovement table has been joined to itself to facilitate the demonstration of the difference between using EXISTS and IN. Note how the IN subquery executes a full table scan and the EXISTS subquery does not.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stockmovement sm
      WHERE EXISTS(
            SELECT qty FROM stockmovement
            WHERE stockmovement_id = sm.stockmovement_id)
      AND stockmovement_id = 10;
```

```
Query                                          Cost      Rows    Bytes
---------------------------------------------- --------  --------  -----------
1.  SELECT STATEMENT on                           2          1       29
2.   NESTED LOOPS SEMI on                        2          1       29
3.    TABLE ACCESS BY INDEX ROWID on
        STOCKMOVEMENT                            2          1       24
4.     INDEX UNIQUE SCAN on
         XPK_STOCKMOVEMENT                       1     570175
3.    INDEX UNIQUE SCAN on
        XPK_STOCKMOVEMENT                                   1        5
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM stockmovement sm
      WHERE qty IN (SELECT qty FROM
        stockmovement)
      AND stockmovement_id = 10;
```

```
Query                                          Cost     Rows     Bytes
---------------------------------------------- --------  --------  -----------
1.  SELECT STATEMENT on                         563        1       28
2.   NESTED LOOPS SEMI on                      563        1       28
3.    TABLE ACCESS BY INDEX ROWID on
        STOCKMOVEMENT                            2        1       24
4.     INDEX UNIQUE SCAN on
         XPK_STOCKMOVEMENT                       1   570175
3.    TABLE ACCESS FULL on
        STOCKMOVEMENT                          561   570175  2280700
```

The benefit of using EXISTS rather than IN for a subquery comparison is that EXISTS can potentially find much fewer rows than IN. *IN is best used with literal values and EXISTS is best used as applying a fast access correlation between a calling and a subquery.*

6.4.4 Groups

ANY, SOME, and ALL comparisons are generally not very conducive to SQL tuning. In some respects they are best not used.

6.5 Joins

A join is a combination of rows extracted from two or more tables. Joins can be very specific, for instance an intersection between two tables, or they can be less specific such as an outer join. An outer join is a join returning an intersection plus rows from either or both tables, not in the other table.

This discussion on tuning joins is divided into three sections: join syntax formats, efficient joins, and inefficient joins. Since this book is about tuning it seems sensible to divide joins between efficient joins and inefficient joins.

Firstly, let's take a look at the two different available join syntax formats in Oracle SQL.

6.5.1 Join Formats

There are two different syntax formats available for SQL join queries. The first is Oracle Corporation's proprietary format and the second is the ANSI standard format. Let's test the two formats to see if either format can be tuned to the best performance.

The Oracle SQL proprietary format places join specifications into the WHERE clause of an SQL query. The only syntactical addition to the standard SELECT statement syntax is the use of the (+) or outer join operator. We will deal with tuning outer joins later in this chapter. Following is an example of an Oracle SQL proprietary join formatted query with its query plan, using the Employees schema. All tables are fully scanned because there is joining but no filtering. The Optimizer forces full table reads on all tables because it is the fastest access method to read all the data.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT di.name, de.name, prj.name
      FROM division di, department de, project prj
```

```
      WHERE di.division_id = de.division_id
      AND de.department_id = prj.department_id;
```

```
Query                                          Cost    Rows    Bytes
---------------------------------------------- ------- ------- -------
SELECT STATEMENT on                              23   10000   640000
 HASH JOIN on                                    23   10000   640000
  HASH JOIN on                                    3     100     3600
   TABLE ACCESS FULL on DIVISION                  1      10      170
   TABLE ACCESS FULL on DEPARTMENT                1     100     1900
  TABLE ACCESS FULL on PROJECT                   13   10000   280000
```

The next example shows the same query except using the ANSI standard join format. Notice how the query plan is identical.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT di.name, de.name, prj.name
      FROM division di JOIN department de
        USING(division_id)
            JOIN project prj USING (department_id);
```

```
Query                                          Cost    Rows    Bytes
---------------------------------------------- ------- ------- -------
SELECT STATEMENT on                              23   10000   640000
 HASH JOIN on                                    23   10000   640000
  HASH JOIN on                                    3     100     3600
   TABLE ACCESS FULL on DIVISION                  1      10      170
   TABLE ACCESS FULL on DEPARTMENT                1     100     1900
  TABLE ACCESS FULL on PROJECT                   13   10000   280000
```

What is the objective of showing the two queries above, including their query plan details? The task of this book is performance tuning. Is either of the two of Oracle SQL proprietary or ANSI join formats inherently faster? Let's try to prove it either way. Once again the Oracle SQL proprietary format is shown below but with a filter added, finding only a single row in the join.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT di.name, de.name, prj.name
      FROM division di, department de, project prj
      WHERE di.division_id = 5
      AND di.division_id = de.division_id
      AND de.department_id = prj.department_id;
```

```
Query                                        Cost  Rows  Bytes
-------------------------------------------- ----- ----- -------
SELECT STATEMENT on                              4   143    9152
 TABLE ACCESS BY INDEX ROWID on PROJECT          2 10000  280000
  NESTED LOOPS on                                4   143    9152
```

```
 NESTED LOOPS on                              2     1     36
  TABLE ACCESS BY INDEX ROWID on DIVISION     1     1     17
   INDEX UNIQUE SCAN on XPKDIVISION                 1
  TABLE ACCESS FULL on DEPARTMENT              1    10    190
 INDEX RANGE SCAN on XFKPROJECT_DEPARTMENT 1 10000
```

Next is the ANSI standard equivalent of the previous join, including the filter. Two of the most important aspects of tuning SQL join queries are the ability to apply filtering prior to joining tables and specifying the table with the largest filter applied as being the first table in the FROM clause, especially for very large tables. The question is this: Does the ANSI format allow for tuning of joins down to these levels of detail? Is the ANSI format a faster and more tunable option?

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT di.name, de.name, prj.name
      FROM division di JOIN department de
            ON(di.division_id = de.division_id)
            JOIN project prj ON(de.department_id =
              prj.department_id)
      WHERE di.division_id = 5;
```

In the previous join query filtering is visibly applied after the specification of the join. Also note that with the addition of filtering the ON clause rather than the USING clause is required. In the following query plan note that the Optimizer has not changed its plan of execution between the Oracle SQL proprietary and ANSI join formats. There is no difference in performance between Oracle SQL proprietary and ANSI standard join formats.

```
Query                                        Cost  Rows  Bytes
-------------------------------------------- ----- ----- -----
SELECT STATEMENT on                             4   143  9152
 TABLE ACCESS BY INDEX ROWID on PROJECT         2 10000 280000
  NESTED LOOPS on                               4   143  9152
   NESTED LOOPS on                              2     1    36
    TABLE ACCESS BY INDEX ROWID on DIVISION     1     1    17
     INDEX UNIQUE SCAN on XPKDIVISION           1
    TABLE ACCESS FULL on DEPARTMENT             1    10   190
   INDEX RANGE SCAN on XFKPROJECT_DEPARTMENT 1 10000
```

A more visibly tunable join could be demonstrated by retrieving a single row from the largest rather than the smallest table. Here is the Oracle SQL proprietary format.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT di.name, de.name, prj.name
```

```
      FROM project prj, department de, division di
      WHERE prj.project_id = 50
      AND de.department_id = prj.department_id
      AND di.division_id = de.division_id;
```

Notice in the following query plan that the cost is the same but the number of rows and bytes read are substantially reduced; only a single row is retrieved. Since the Project table is being reduced in size more than any other table it appears first in the FROM clause. The same applies to the Department table being larger than the Division table.

```
Query                                          Cost   Rows  Bytes
---------------------------------------------- -------  ---------  ----------
SELECT STATEMENT on                              4       1     67
 NESTED LOOPS on                                 4       1     67
  NESTED LOOPS on                                3       1     50
   TABLE ACCESS BY INDEX ROWID on PROJECT        2       1     31
    INDEX UNIQUE SCAN on XPKPROJECT              1       1
   TABLE ACCESS BY INDEX ROWID on
     DEPARTMENT                                  1     100   1900
    INDEX UNIQUE SCAN on XPKDEPARTMENT                100
  TABLE ACCESS BY INDEX ROWID on DIVISION        1      10    170
   INDEX UNIQUE SCAN on XPKDIVISION                     10
```

Now let's do the same query but with the ANSI join format. From the following query plan we can once again see that use of either the Oracle SQL proprietary or ANSI join format does not appear to make any difference to performance and capacity for tuning.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT di.name, de.name, prj.name
      FROM project prj JOIN department de
            ON(prj.department_id = de.department_id)
            JOIN division di ON(de.division_id =
              di.division_id)
      WHERE prj.project_id = 50;
```

```
Query                                          Cost   Rows  Bytes
---------------------------------------------- -------  ---------  ----------
SELECT STATEMENT on                              4       1     67
 NESTED LOOPS on                                 4       1     67
  NESTED LOOPS on                                3       1     50
   TABLE ACCESS BY INDEX ROWID on PROJECT        2       1     31
    INDEX UNIQUE SCAN on XPKPROJECT              1       1
   TABLE ACCESS BY INDEX ROWID on
     DEPARTMENT                                  1     100   1900
```

```
    INDEX UNIQUE SCAN on XPKDEPARTMENT              100
  TABLE ACCESS BY INDEX ROWID on DIVISION    1     10    170
   INDEX UNIQUE SCAN on XPKDIVISION                 10
```

Let's take this further and do some time testing. We will use the Accounts schema since the Employees schema does not have much data. We want to retrieve more rows to give a better chance of getting a time difference, thus we will not filter on the largest table first. As can be seen from the following results the timing is identical. Perhaps changing the join orders could make subtle differences but there is no reason why the ANSI join format should be considered less tunable.

```
SQL>SELECT COUNT(*) FROM (
  2 SELECT t.text, st.text, coa.text, gl.dr, gl.cr
  3 FROM type t , subtype st, coa, generalledger gl
  4 WHERE t.type = 'A'
  5 AND coa.type = t. type
  6 AND coa.subtype = st.subtype
  7 AND gl.coa# = coa.coa#);

   COUNT(*)
----------
    239848

Elapsed: 00:00:04.06

SQL>SELECT COUNT(*) FROM (
  2 SELECT t.text, st.text, coa.text, gl.dr, gl.cr
  3 FROM type t JOIN coa ON(t.type = coa.type)
  4 JOIN subtype st ON(st.subtype = coa.subtype)
  5 JOIN generalledger gl ON(gl.coa# = coa.coa#)
  6 WHERE t.type = 'A');

   COUNT(*)
----------
    239848

Elapsed: 00:00:04.06
```

6.5.2 Efficient Joins

What is an efficient join? An efficient join is a join SQL query which can be tuned to an acceptable level of performance. Certain types of join queries are inherently easily tuned and thus can give good performance. In general, a join is efficient when it can use indexes on

large tables or is reading only very small tables. Moreover, any type of join will be inefficient if coded improperly.

Intersections

An inner or natural join is an intersection between two tables. In Set parlance an intersection contains all elements occurring in both of the sets, or common to both sets. An intersection is efficient when index columns are matched together in join clauses. Obviously intersection matching not using indexed columns will be inefficient. In that case you may want to create alternate indexes. On the other hand, when a table is very small the Optimizer may conclude that reading the whole table is faster than reading an associated index plus the table. How the Optimizer makes a decision such as this will be discussed in later chapters since this subject matter delves into indexing and physical file block structure in Oracle Database datafiles.

In the example below both of the Type and COA tables are so small that the Optimizer does not bother with the indexes and simply reads both of the tables fully.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT t.text, coa.text FROM type t JOIN coa
        USING(type);
```

```
Query                                          Cost   Rows   Bytes
---------------------------------------------- -------  -------  -----------
SELECT STATEMENT on                               3     55   1430
 HASH JOIN on                                     3     55   1430
  TABLE ACCESS FULL on TYPE                       1      6     54
   TABLE ACCESS FULL on COA                       1     55    935
```

With the next example the Optimizer has done something a little odd by using a unique index on the Subtype table. The Subtype table has only four rows and is extremely small.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT t.text, coa.text FROM type t JOIN coa
        USING(type)
            JOIN subtype st USING(subtype);
```

```
Query                                          Cost   Rows   Bytes
---------------------------------------------- -------  -------  -----------
SELECT STATEMENT on                               3     55   1650
 NESTED LOOPS on                                  3     55   1650
```

```
  HASH JOIN on                                3     55    1540
   TABLE ACCESS FULL on TYPE                  1      6      54
  TABLE ACCESS FULL on COA                    1     55    1045
   INDEX UNIQUE SCAN on XPKSUBTYPE                  4       8
```

Once again in the following example the Optimizer has chosen to read the index for the very small Subtype table. However, the GeneralLedger table has its index read because it is very large and the Optimizer considers that more efficient. The reason for this is that the GeneralLedger table does have an index on the COA# column and thus the index is range scanned.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT t.text, coa.text
      FROM type t JOIN coa USING(type)
            JOIN subtype st USING(subtype)
                  JOIN generalledger gl ON(gl.coa# =
                    coa.coa#);
```

```
Query                                       Cost     Rows      Bytes
------------------------------------------- -------  --------  ----------
SELECT STATEMENT on                           58   752740   31615080
 NESTED LOOPS on                              58   752740   31615080
  NESTED LOOPS on                              3       55       1980
   HASH JOIN on                                3       55       1870
    TABLE ACCESS FULL on TYPE                  1        6         54
    TABLE ACCESS FULL on COA                   1       55       1375
   INDEX UNIQUE SCAN on XPKSUBTYPE                      4          8
  INDEX RANGE SCAN on XFK_GL_COA#              1   752740    4516440
```

The most efficient type of inner join will generally be one retrieving very specific rows such as that in the next example. Most SQL is more efficient when retrieving very specific, small numbers of rows.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT t.text, st.text, coa.text, gl.dr, gl.cr
      FROM generalledger gl JOIN coa ON(gl.coa# = coa.coa#)
            JOIN type t ON(t.type = coa.type)
                  JOIN subtype st ON(st.subtype =
                    coa.subtype)
      WHERE gl.generalledger_id = 100;
```

Note how all tables in the query plan are accessed using unique index hits.

```
Query                                       Pos  Cost  Rows  Bytes
------------------------------------------- ---  ----  ----  -----
SELECT STATEMENT on                           6     6     1     64
 NESTED LOOPS on                              1     6     1     64
```

```
NESTED LOOPS on                          1     5     1    55
 NESTED LOOPS on                         1     4     1    45
  TABLE ACCESS BY INDEX ROWID on GENE    1     3     1    20
   INDEX UNIQUE SCAN on XPKGENERALLED    1     2     1
  TABLE ACCESS BY INDEX ROWID on COA     2     1    55  1375
   INDEX UNIQUE SCAN on XPKCOA           1          55
 TABLE ACCESS BY INDEX ROWID on SUBTY    2     1     4    40
  INDEX UNIQUE SCAN on XPKSUBTYPE        1           4
TABLE ACCESS BY INDEX ROWID on TYPE      2     1     6    54
 INDEX UNIQUE SCAN on XPKTYPE            1           6
```

Self Joins

A self join joins a table to itself. Sometimes self-joining tables can be handled with hierarchical queries. Otherwise a self join is applied to a table containing columns within each row which link to each other. The Employee table in the Employees schema is such a table. Since both the MANAGER_ID and EMPLOYEE_ID columns are indexed it would be fairly efficient to join the tables using those two columns.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT manager.name, employee.name
      FROM employee manager, employee employee
      WHERE employee.manager_id = manager.employee_id;
```

In the query plan the Employee table is fully scanned twice because all the data is read and the Optimizer considers this faster because the Employee table is small.

```
Query                                    Cost     Rows      Bytes
---------------------------------------- -------- --------- -----------
SELECT STATEMENT on                         3       110       2970
 HASH JOIN on                               3       110       2970
  TABLE ACCESS FULL on EMPLOYEE             1       111       1554
   TABLE ACCESS FULL on EMPLOYEE            1       111       1443
```

Equi-Joins and Range Joins

An equi-join uses the equals sign (=) and a range join uses range operators (<, >, <=, >=) and the BETWEEN operator. In general, the = operator will execute an exact row hit on an index and thus use unique index hits. The range operators will usually require the Optimizer to execute index range scans. BTree (binary tree) indexes,

the most commonly used indexes in Oracle Database, are highly amenable to range scans. A BTree index is little like a limited depth tree and is optimized for both unique hits and range scans.

Going back into the Accounts schema, this first query uses two unique index hits. The filter helps that happen.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT coa.*, gl.*
      FROM generalledger gl JOIN coa ON(gl.coa# = coa.coa#)
      WHERE generalledger_id = 10;
```

```
Query                                          Cost   Rows   Bytes
---------------------------------------------- -----  -----  -----
SELECT STATEMENT on                               4      1     51
 NESTED LOOPS on                                  4      1     51
  TABLE ACCESS BY INDEX ROWID on
    GENERALLEDG                                   3      1     26
   INDEX UNIQUE SCAN on XPKGENERALLEDGER          2      1
  TABLE ACCESS BY INDEX ROWID on COA              1     55   1375
   INDEX UNIQUE SCAN on XPKCOA                          55
```

This second query uses a range index scan on the GeneralLedger table as a result of the range operator in the filter. Do you notice that the join clause inside the ON clause is where the range join operator is placed? Well there isn't really much point in joining ON (gl.coa# >= coa.coa#). I do not think I have ever seen an SQL join joining using a range operator. The result would be a very unusual type of outer join perhaps. Thus there is no need for a query plan.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT coa.*, gl.*
      FROM generalledger gl JOIN coa ON(gl.coa# >= coa.coa#)
      WHERE generalledger_id = 10;
```

ⓘ A Cartesian product is generally useless in a relational database, so is a range join.

6.5.3 Inefficient Joins

What is an inefficient join? An inefficient join is an SQL query joining tables which is difficult to tune or cannot be tuned to an acceptable level of performance. Certain types of join queries are inherently

both poor performers and difficult if not impossible to tune. Inefficient joins are best avoided.

Cartesian Products

The ANSI join format calls a Cartesian product a Cross Join. A Cross Join is only tunable as far as columns selected match indexes such that rows are retrieved from indexes and not tables.

The following second query has a lower cost than the first because the selected columns match indexes on both tables. I have left the Rows and Bytes columns in the query plans as overflowed numbers replaced with a string of # characters. This is done to stress the pointlessness of using a Cartesian product in a relational database.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM coa, generalledger;
```

```
Query                                    Cost      Rows       Bytes
---------------------------------------- --------- ---------- ----------
SELECT STATEMENT on                     27116   #######   #########
 MERGE JOIN CARTESIAN on                27116   #######   #########
  TABLE ACCESS FULL on COA                  1        55        1375
   BUFFER SORT on                       27115    752740    19571240
    TABLE ACCESS FULL on
      GENERALLEDGER                       493    752740    19571240
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT coa.coa#, gl.generalledger_id FROM coa,
        generalledger gl;
```

```
Query                                    Cost      Rows       Bytes
---------------------------------------- --------- ---------- ----------
SELECT STATEMENT on                     11936   #######   #########
 MERGE JOIN CARTESIAN on                11936   #######   #########
  INDEX FULL SCAN on XPKCOA                 1        55         330
  BUFFER SORT on                        11935    752740     4516440
   INDEX FAST FULL SCAN on
     XPKGENERALLEDGER                     217    752740     4516440
```

Outer Joins

Tuning an outer join requires the same approach to tuning as with an inner join. The only point to note is that if applications require a

large quantity of outer joins there is probably potential for data model tuning. The data model could be too granular. Outer joins are probably more applicable to reporting and data warehouse type applications.

An outer join is not always inefficient. The performance and to a certain extent the indication of a need for data model tuning depends on the ratio of rows retrieved from the intersection to rows retrieved outside the intersection. The more rows retrieved from the intersection the better.

My question is this: Why are outer joins needed? Examine the data model first.

Anti-Joins

An anti-join is always a problem. An anti-join simply does the opposite of a requirement. The result is that the Optimizer must search for everything not meeting a condition. An anti-join will generally always produce a full table scan as seen in the first example following. The second example uses one index because indexed columns are being retrieved from one of the tables. Again the Rows and Bytes columns are left as overflowing showing the possibly folly of using anti-joins.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT t.text, coa# FROM type t, coa WHERE
        t.type != coa.type;
```

```
Query                                    Cost       Rows       Bytes
---------------------------------------- -------    -------    ----------
SELECT STATEMENT on                         7        275        4675
 NESTED LOOPS on                            7        275        4675
  TABLE ACCESS FULL on TYPE                 1          6          54
   TABLE ACCESS FULL on COA                 1         55         440
```

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT coa.coa#, gl.generalledger_id FROM coa,
        generalledger gl
      WHERE coa.coa# != gl.coa#;
```

```
Query                              Pos   Cost     Rows      Bytes
---------------------------------- -------- --------- ---------- ----------
SELECT STATEMENT on              27116  27116  ####### #########
 NESTED LOOPS on                     1  27116  ####### #########
```

```
INDEX FULL SCAN on XPKCOA      1      1       55        330
 TABLE ACCESS FULL on
   GENERALLEDGER               2    493   752740    9032880
```

Mutable and Complex Joins

A mutable join is a join of more than two tables. A complex join is a mutable join with added filtering. We have already examined a complex mutable join in the section on intersection joins and various other parts of this chapter.

6.5.4 How to Tune a Join

So how can a join be tuned? There are a number of factors to consider.

- Use equality first.
- Use range operators only where equality does not apply.
- Avoid use of negatives in the form of != or NOT.
- Avoid LIKE pattern matching.
- Try to retrieve specific rows and in small numbers.
- Filter from large tables first to reduce rows joined. Retrieve tables in order from the most highly filtered table downwards; preferably the largest table has the most filtering applied.

ⓘ The most highly filtered table is the table having the smallest percentage of its rows retrieved, preferably the largest table.

- Use indexes wherever possible except for very small tables.
- Let the Optimizer do its job.

6.6 Using Subqueries for Efficiency

Tuning subqueries is a highly complex topic. Quite often subqueries can be used to partially replace subset parts of very large mutable joins, with possible enormous performance improvements.

6.6.1 Correlated versus Noncorrelated Subqueries

A correlated subquery allows a correlation between a calling query and a subquery. A value for each row in the calling query is passed into the subquery to be used as a constraint by the subquery. A noncorrelated or regular subquery does not contain a correlation between calling query and subquery and thus the subquery is executed in its entirety, independently of the calling query, for each row in the calling query. Tuning correlated subqueries is easier because values in subqueries can be precisely searched for in relation to each row of the calling query.

A correlated subquery will access a specified row or set of rows for each row in the calling query. Depending on circumstances a correlated subquery is not always faster than a noncorrelated subquery. Use of indexes or small tables inside a subquery, even for noncorrelated subqueries, does not necessarily make a subquery perform poorly.

6.6.2 IN versus EXISTS

We have already seen substantial use of IN and EXISTS in the section on comparison conditions. We know already that IN is best used for small tables or lists of literal values. EXISTS is best used to code queries in a correlated fashion, establishing a link between a calling query and a subquery. To reiterate it is important to remember that using EXISTS is not always faster than using IN.

6.6.3 Nested Subqueries

Subqueries can be nested where a subquery can call another subquery. The following example using the Employees schema shows a query calling a subquery, which in turn calls another subquery.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
SELECT * FROM division WHERE division_id IN
      (SELECT division_id FROM department WHERE
        department_id IN
             (SELECT department_id FROM project));
```

Notice in the query plan how the largest table is scanned using an INDEX FAST FULL SCAN. The Optimizer is intelligent enough to

analyze this nested query and discern that the Project table is much larger than both of the other two tables. The other two tables are so small that the only viable option is a full table scan.

```
Query                                      Cost     Rows     Bytes
-----------------------------------------  -------  -------  -------
SELECT STATEMENT on                          14       10       590
 HASH JOIN SEMI on                           14       10       590
  TABLE ACCESS FULL on DIVISION               1       10       460
  VIEW on VW_NSO_1                            8    10000    130000
   HASH JOIN on                               8    10000     60000
    TABLE ACCESS FULL on DEPARTMENT           1      100       400
    INDEX FAST FULL SCAN on
      XFKPROJECT_DEPT                         4    10000     20000
```

Nested subqueries can be difficult to tune but can often be a viable and sometimes highly effective tool for the tuning of mutable complex joins, with three and sometimes many more tables in a single join. There is a point when there are so many tables in a join that the Optimizer can become less effective.

6.6.4 Replacing Joins with Subqueries

For very large complex mutable joins it is often possible to replace joins or parts of joins with subqueries. Very large joins can benefit the most because they are difficult to decipher and tune. Some very large joins are even beyond the intelligence of the Optimizer to assess in the best possible way. Two ways in which subqueries can replace joins in complex mutable joins are as follows:

- A table in the join not returning a column in the primary calling query can be removed from the join and checked using a subquery.
- FROM clauses can contain nested subqueries to break up joins much in the way that PL/SQL would use nested looping cursors.

Certain aspects of SQL coding placed in subqueries can cause problems:

- An ORDER BY clause is always applied to a final result and should not be included in subqueries if possible.
- DISTINCT will always cause a sort and is not always necessary. Perhaps a parent table could be used where a unique value is present.

- When testing against subqueries retrieve, filter, and aggregate on indexes not tables. Indexes usually offer better performance.
- Do not be too concerned about full table scans on very small static tables.

ⓘ Instances where joins can be replaced with subqueries often involve databases with heavy outer join requirements. Excessive use of SQL outer joins is possibly indicative of an over-granular data model structure. However, it could also indicate orphaned child table rows or the opposite: redundant static data. Cleaning out redundant or orphaned rows can sometimes help performance immensely by negating the need for outer joins.

Remove Tables Without Returned Columns Using EXISTS

Going back to the Accounts schema once again look at the following complex mutable join. We are joining four tables and selecting a column from only one of the tables. EXISTS comparisons can be placed into the WHERE clause to force index access, removing three tables from the join.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT c.name
      FROM customer c JOIN orders o USING(customer_id)
            JOIN ordersline ol USING(order_id)
                  JOIN transactions t USING(customer_id)
                        JOIN transactionsline tl
                          USING(transaction_id)
      WHERE c.balance > 0;
```

Its query plan is a little scary. There are three full table scans and two full index scans. The objective is to remove full table scans and change as many index scans as possible into unique index scans.

```
Query                                    Cost       Rows       Bytes
---------------------------------------- ---------- ---------- ----------
SELECT STATEMENT on                      ######     #######    #########
 MERGE JOIN on                           ######     #######    #########
  SORT JOIN on                           ######     #######    #########
   MERGE JOIN on                           4988     #######    #########
```

```
    SORT JOIN on                    3209    100136    3805168
     HASH JOIN on                    762    100136    3805168
      TABLE ACCESS FULL on
        CUSTOMER                      9      2690      69940
      MERGE JOIN on                 316    100237    1202844
       INDEX FULL SCAN on
         XFK_ORDERLINE_ORDER         26    540827    2704135
       SORT JOIN on                 290     31935     223545
        TABLE ACCESS FULL on
          ORDERS                    112     31935     223545
   SORT JOIN on                    1780    188185    1317295
    TABLE ACCESS FULL on
      TRANSACTIONS                  187    188185    1317295
 SORT JOIN on                       5033    570175    2850875
  INDEX FAST FULL SCAN on
    XFK_TRANSLINE_TRANS               4    570175    2850875
```

Only the Customer.NAME column is selected. This query is an extreme case but we can actually remove every table from the join except the Customer table. Let's show this in two stages. Firstly, I will remove the transaction tables from the join.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT c.name
      FROM customer c JOIN orders o ON(c.customer_id =
        o.customer_id)
            JOIN ordersline ol USING(order_id)
      WHERE c.balance > 0
      AND EXISTS(
            SELECT t.transaction_id FROM transactions t
            WHERE t.customer_id = c.customer_id
            AND EXISTS(
                  SELECT transaction_id FROM
                    transactionsline
                  WHERE transaction_id = t.transaction_id
            )
      );
```

We have now reduced the full table scans to two, have a single full index scan and most importantly index range scans on both of the transaction tables.

```
Query                                    Cost     Rows     Bytes
---------------------------------------- -------- -------- ----------
SELECT STATEMENT on                       359     5007     190266
 FILTER on
  HASH JOIN on                            359     5007     190266
   TABLE ACCESS FULL on CUSTOMER            9      135       3510
```

```
 MERGE JOIN on                          316  100237  1202844
  INDEX FULL SCAN on
    XFK_ORDERLINE_ORDER                  26  540827  2704135
  SORT JOIN on                          290   31935   223545
   TABLE ACCESS FULL on ORDERS          112   31935   223545
NESTED LOOPS on                          72     212     2544
 TABLE ACCESS BY INDEX ROWID on
  TRANSACTIONS                            2      70      490
 INDEX RANGE SCAN on
  XFX_TRANS_CUSTOMER                      1      70
INDEX RANGE SCAN on
  XFK_TRANSLINE_TRANS                     1  570175  2850875
```

Now let's get completely ridiculous and remove every table from the join but the Customer table.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT c.name FROM customer c
      WHERE c.balance > 0
      AND EXISTS(
            SELECT o.order_id FROM orders o
            WHERE o.customer_id = c.customer_id
            AND EXISTS(
                  SELECT order_id FROM ordersline
                  WHERE order_id = o.order_id
            )
      )
      AND EXISTS(
            SELECT t.transaction_id FROM transactions t
            WHERE t.customer_id = c.customer_id
            AND EXISTS(
                  SELECT transaction_id FROM
                    transactionsline
                  WHERE transaction_id = t.transaction_id
            )
      );
```

This is about the best that can be done with this query, now no longer a join. This final result has full table access on the Customer table only, along with four index range scans. We could possibly improve the query further by decreasing the number of Customer rows retrieved using filtering.

```
Query                                  Cost      Rows     Bytes
-------------------------------------- --------  -------  -------
SELECT STATEMENT on                       9         7       182
 FILTER on
  TABLE ACCESS FULL on CUSTOMER           9         7       182
```

```
NESTED LOOPS on                      66       201      2412
 TABLE ACCESS BY INDEX ROWID on
   ORDERS                             2        64       448
  INDEX RANGE SCAN on
    XFK_ORDERS_CUSTOMER               1        64
 INDEX RANGE SCAN on
   XFK_ORDERLINE_ORDER                1    540827   2704135
NESTED LOOPS on                      72       212      2544
 TABLE ACCESS BY INDEX ROWID on
   TRANSACTIONS                       2        70       490
  INDEX RANGE SCAN on
    XFX_TRANS_CUSTOMER                1        70
 INDEX RANGE SCAN on
   XFK_TRANSLINE_TRANS                1    570175   2850875
```

FROM Clause Subquery Nesting

Now what we want to do is to retrieve columns from different tables. Columns cannot be retrieved from an EXISTS comparison in the WHERE clause. We have to use another method. Nested subqueries in the FROM clause allow retrieval of columns.

In this example I am adding extra filtering to the TransactionsLine table; at over 500,000 rows it is the largest table in the query. Since the TransactionsLine table is larger than the Customer table it is filtered first.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT c.name, tl.amount FROM customer c
       JOIN orders o USING(customer_id)
        JOIN ordersline ol USING(order_id)
         JOIN transactions t USING(customer_id)
          JOIN transactionsline tl USING(transaction_id)
      WHERE tl.amount > 3170
      AND c.balance > 0;
```

We start with three full table scans, one index range scan and a unique index hit.

```
Query                                  Cost       Rows      Bytes
-------------------------------------- ---------- --------- ---------
SELECT STATEMENT on                    1860         605      33880
 NESTED LOOPS on                       1860         605      33880
  HASH JOIN on                         1667         193       9843
   TABLE ACCESS FULL on ORDERS          112       31935     223545
   MERGE JOIN CARTESIAN on              436       43804    1927376
```

```
    NESTED LOOPS on                    292        16       288
     TABLE ACCESS FULL on
       TRANSACTIONSLINE                276        16       176
     TABLE ACCESS BY INDEX ROWID on
       TRANSACTIO                        1    188185   1317295
      INDEX UNIQUE SCAN on
        XPKTRANSACTIONS                       188185
    BUFFER SORT on                     435      2690     69940
     TABLE ACCESS FULL on CUSTOMER       9      2690     69940
  INDEX RANGE SCAN on
    XFK_ORDERLINE_ORDER                1    540827   2704135
```

Firstly, some appropriate simple tuning can be done. Since the TransactionsLine table is the largest table with the smallest relative filtered result, it should be selected from first.

ⓘ The first table to be processed should be the largest table with the largest relative row reduction filter. In other words, the biggest table with the lowest number of rows retrieved from it. This applies to both the FROM clause and the WHERE clause. Always reduce rows to be joined first.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT c.name, tl.amount FROM transactionsline tl
       JOIN transactions t USING(transaction_id)
        JOIN customer c USING(customer_id)
         JOIN orders o USING(customer_id)
          JOIN ordersline ol ON(ol.order_id = o.order_id)
      WHERE tl.amount > 3170
      AND c.balance > 0;
```

Appropriate simple tuning yields one full table scan, two index range scans, and two unique index hits.

```
Query                                  Cost      Rows      Bytes
-------------------------------------- --------  --------  ----------
SELECT STATEMENT on                    1381         3267    182952
 NESTED LOOPS on                       1381         3267    182952
  NESTED LOOPS on                       340         1041     53091
   NESTED LOOPS on                      308           16       704
    NESTED LOOPS on                     292           16       288
     TABLE ACCESS FULL on
       TRANSACTIONSLINE                 276           16       176
     TABLE ACCESS BY INDEX ROWID
       on TRANSACTIO                      1        33142    231994
```

```
      INDEX UNIQUE SCAN on
        XPKTRANSACTIONS                      33142
    TABLE ACCESS BY INDEX ROWID on
      CUSTOMER                        1      2690    69940
     INDEX UNIQUE SCAN on
       XPKCUSTOMER                           2690
   TABLE ACCESS BY INDEX ROWID on
     ORDERS                           2    172304  1206128
    INDEX RANGE SCAN on
      XFK_ORDERS_CUSTOMER             1    172304
  INDEX RANGE SCAN on
    XFK_ORDERLINE_ORDER               1    540827  2704135
```

Now let's use the FROM clause to create nested subqueries. The trick is to put the largest table with the most severe filter at the deepest nested level, forcing it to execute first. Thus we start with the TransactionsLine table.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT c.name, b.amount
      FROM customer c,
      (
            SELECT t.customer_id, a.amount
            FROM transactions t,(
                  SELECT transaction_id, amount FROM
                    transactionsline
                  WHERE amount > 3170
            ) a
            WHERE t.transaction_id = a.transaction_id
      ) b, orders o, ordersline ol
      WHERE c.balance > 0
      AND c.customer_id = b.customer_id
      AND o.customer_id = c.customer_id
      AND ol.order_id = o.order_id;
```

The cost is reduced further with the same combination of scans because fewer rows are being joined.

```
Query                                  Cost      Rows     Bytes
-------------------------------------- --------- -------- ------------
SELECT STATEMENT on                     533       605     33880
 NESTED LOOPS on                        533       605     33880
  NESTED LOOPS on                       340       193      9843
   NESTED LOOPS on                      308        16       704
    NESTED LOOPS on                     292        16       288
     TABLE ACCESS FULL on
       TRANSACTIONSLINE                 276        16       176
```

```
      TABLE ACCESS BY INDEX ROWID
        on TRANSACTIO                    1    33142   231994
       INDEX UNIQUE SCAN on
         XPKTRANSACTIONS                      33142
     TABLE ACCESS BY INDEX ROWID
       on CUSTOMER                       1     2690    69940
      INDEX UNIQUE SCAN on
        XPKCUSTOMER                           2690
    TABLE ACCESS BY INDEX ROWID on
      ORDERS                             2    31935   223545
     INDEX RANGE SCAN on
       XFK_ORDERS_CUSTOMER               1    31935
  INDEX RANGE SCAN on
    XFK_ORDERLINE_ORDER                  1   540827  2704135
```

Now let's combine WHERE clause comparison subqueries and FROM clause embedded subqueries.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT c.name, b.amount
      FROM customer c,
      (
            SELECT t.customer_id, a.amount
            FROM transactions t,(
                  SELECT transaction_id, amount FROM
                    transactionsline
                  WHERE amount > 3170
            ) a
            WHERE t.transaction_id = a.transaction_id
      ) b
      WHERE c.balance > 0
      AND EXISTS(
            SELECT o.order_id FROM orders o
            WHERE o.customer_id = c.customer_id
            AND EXISTS(
                  SELECT order_id FROM ordersline
                  WHERE order_id = o.order_id
            )
      );
```

Using EXISTS makes the query just that little bit faster with lower cost since the number of tables joined is reduced.

```
Query                                          Cost      Rows     Bytes
---------------------------------------------- --------- -------- ----------
SELECT STATEMENT on                            420       2190     91980
 FILTER on
  NESTED LOOPS on                              420       2190     91980
```

```
 MERGE JOIN CARTESIAN on            420      2190     81030
  TABLE ACCESS FULL on
    TRANSACTIONSLINE                276        16       176
  BUFFER SORT on                    144       135      3510
   TABLE ACCESS FULL on CUSTOMER      9       135      3510
 INDEX UNIQUE SCAN on
   XPKTRANSACTIONS                          188185    940925
NESTED LOOPS on                      66       201      2412
 TABLE ACCESS BY INDEX ROWID on
   ORDERS                             2        64       448
  INDEX RANGE SCAN on
    XFK_ORDERS_CUSTOMER              1        64
 INDEX RANGE SCAN on
   XFK_ORDERLINE_ORDER               1    540827   2704135
```

6.7 Using Synonyms

A synonym is, as its name implies, another name for a known object. Synonyms are typically used to reference tables between schemas. Public synonyms make tables contained within schemas available to all schemas. Apart from the obvious security issues there can be potential performance problems when over-using synonyms in highly concurrent environments. It may not necessarily be effective to divide functionality between different schemas only to allow users global or semi-global access to all of the underlying schemas. Simplicity in development using objects like synonyms often leads to complexity and performance problems in production. Additionally too many metadata objects can cause problems with the shared pool.

6.8 Using Views

Views are application- and security-friendly. Views can also be used to reduce complexity, particularly in development. In general, views are not conducive to good performance. A view is a logical overlay on top of one or more tables. A view is created using an SQL statement. A view does not contain data itself. The biggest problem with a view is that whenever it is queried its defining SQL statement is re-executed. It is common in applications for a developer to query a view and add additional filtering. The potential results are views containing large queries where programmers will then execute small row number

retrievals from the view. Thus two queries are executed, commonly with the view query selecting all the rows in the underlying table or join.

Let's try to prove that views are inherently slower than direct table queries. Firstly, I create a view on my largest Accounts schema table.

```
CREATE VIEW glv AS SELECT * FROM generalledger;
```

Now let's do some query plans. I have four queries and query plans listed. The first two retrieve a large number of rows from the view and then the table. It is apparent that the query plans are identical in cost.

Selecting from the view:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM glv WHERE coa# = '40003';
```

```
Query                                          Cost      Rows     Bytes
---------------------------------------------- -------  -------  -------
SELECT STATEMENT on                             165     150548   3914248
 TABLE ACCESS BY INDEX ROWID on
   GENERALLEDGE                                 165     150548   3914248
  INDEX RANGE SCAN on XFK_GL_COA#                 5     150548
```

Selecting from the table:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger WHERE coa# = '40003';
```

```
Query                                          Cost      Rows     Bytes
---------------------------------------------- -------  -------  -------
SELECT STATEMENT on                             165     150548   3914248
 TABLE ACCESS BY INDEX ROWID on
   GENERALLEDGE                                 165     150548   3914248
  INDEX RANGE SCAN on XFK_GL_COA#                 5     150548
```

Now let's filter and return much fewer rows. Once again the query plans are the same.

Selecting from the view:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM glv WHERE generalledger_id = 500000;
```

```
Query                                          Cost   Rows    Bytes
---------------------------------------------- -----  -----  -------
SELECT STATEMENT on                              3       1       26
 TABLE ACCESS BY INDEX ROWID on
   GENERALLEDGE                                  3       1       26
  INDEX UNIQUE SCAN on XPKGENERALLEDGER          2       1
```

Selecting from the table:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM generalledger WHERE generalledger_id =
        500000;

Query                                          Cost  Rows   Bytes
---------------------------------------------------------------------------
SELECT STATEMENT on                             3      1      26
 TABLE ACCESS BY INDEX ROWID on
   GENERALLEDGE                                 3      1      26
  INDEX UNIQUE SCAN on XPKGENERALLEDGER         2      1
```

So let's now try some time tests. The COUNT function is used as a wrapper and each query is executed twice to ensure there is no conflict between reading from disk and memory.

```
SELECT COUNT(*) FROM(SELECT * FROM glv WHERE coa# = '40003');
SELECT COUNT(*) FROM(SELECT * FROM generalledger WHERE
  coa# = '40003');
SELECT COUNT(*) FROM(SELECT * FROM glv
      WHERE generalledger_id = 500000);
SELECT COUNT(*) FROM(SELECT * FROM generalledger
      WHERE generalledger_id = 500000);
```

In the first two instances retrieving from the view is faster than reading from the table.

```
SQL> SELECT COUNT(*) FROM(SELECT * FROM glv WHERE
  coa# = '40003');

 COUNT(*)
----------
     66287

Elapsed: 00:00:04.04

SQL> SELECT COUNT(*) FROM(SELECT * FROM generalledger WHERE
  coa# = '40003');

 COUNT(*)
----------
     66287

Elapsed: 00:00:04.09

SQL> SELECT COUNT(*) FROM(SELECT * FROM glv WHERE
  generalledger_id = 500000);

 COUNT(*)
----------
         1
```

```
Elapsed: 00:00:00.00

SQL> SELECT COUNT(*) FROM(SELECT * FROM generalledger WHERE
  generalledger_id = 500000);

 COUNT(*)
----------
         1

Elapsed: 00:00:00.00
```

For a single table and a view on that table there is no difference in query plan or execution time.

Now let's go and re-create our view and re-create it with a join rather than just a single table. This code drops and re-creates the view I created previously.

```
DROP VIEW glv;
CREATE VIEW glv AS
      SELECT gl.generalledger_id, coa.coa#, t.text AS type,
            st.text AS subtype, coa.text as coa, gl.dr,
              gl.cr, gl.dte
      FROM type t JOIN coa USING(type)
            JOIN subtype st USING(subtype)
                  JOIN generalledger gl ON(gl.coa# =
                    coa.coa#);
```

When retrieving a large percentage of rows in the following two queries the cost in the query plan is much better when retrieving using the tables join rather than the view.

Selecting from the view:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM glv WHERE coa# = '40003';
```

Query	Cost	Rows	Bytes
SELECT STATEMENT on	**168**	30110	2107700
NESTED LOOPS on	168	30110	2107700
NESTED LOOPS on	3	1	44
NESTED LOOPS on	2	1	34
TABLE ACCESS BY INDEX ROWID on COA	1	1	25
INDEX UNIQUE SCAN on XPKCOA		1	
TABLE ACCESS BY INDEX ROWID on TYPE	1	6	54
INDEX UNIQUE SCAN on XPKTYPE		6	
TABLE ACCESS BY INDEX ROWID on SUBTYPE	1	4	40

```
    INDEX UNIQUE SCAN on XPKSUBTYPE              4
  TABLE ACCESS BY INDEX ROWID on
    GENERALLEDG                            165  150548  3914248
   INDEX RANGE SCAN on XFK_GL_COA#           5  150548
```

Selecting from the table join:

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT gl.generalledger_id, coa.coa#, t.text AS type,
            st.text AS subtype, coa.text as coa, gl.dr,
              gl.cr, gl.dte
      FROM type t JOIN coa USING(type)
            JOIN subtype st USING(subtype)
                  JOIN generalledger gl ON(gl.coa# =
                    coa.coa#)
      WHERE gl.coa# = '40003';
```

```
Query                                          Cost    Rows    Bytes
---------------------------------------------- -------- ---------- ---------------
SELECT STATEMENT on                               5   30110  2107700
 NESTED LOOPS on                                  5   30110  2107700
  NESTED LOOPS on                                 3       1       44
   NESTED LOOPS on                                2       1       34
    TABLE ACCESS BY INDEX ROWID
      on COA                                      1       1       25
     INDEX UNIQUE SCAN on XPKCOA                          1
    TABLE ACCESS BY INDEX ROWID on
      TYPE                                        1       6       54
     INDEX UNIQUE SCAN on XPKTYPE                         6
   TABLE ACCESS BY INDEX ROWID on
     SUBTYPE                                      1       4       40
    INDEX UNIQUE SCAN on XPKSUBTYPE                       4
  TABLE ACCESS BY INDEX ROWID on
    GENERALLEDG                                   2  150548  3914248
   INDEX RANGE SCAN on XFK_GL_COA#                1  150548
```

Let's try some timing tests. The first timing test retrieves a large set of rows from the view.

```
SQL> SELECT COUNT(*) FROM(SELECT * FROM glv WHERE coa# =
  '40003');

  COUNT(*)
----------
     66287

Elapsed: 00:00:04.02
```

The second timing test retrieves the same large set of rows from the table join and is obviously much faster.

```
SQL> SELECT COUNT(*) FROM(
  2  SELECT gl.generalledger_id, coa.coa#, t.text AS type,
       st.text AS subtype, coa.text as coa, gl.dr, gl.cr,
       gl.dte
  3  FROM type t JOIN coa USING(type)
  4    JOIN subtype st USING(subtype)
  5     JOIN generalledger gl ON(gl.coa# = coa.coa#)
  6  WHERE gl.coa# = '40003');

  COUNT(*)
----------
     66287

Elapsed: 00:00:00.07
```

Comparing times to retrieve a single row, there is no difference between the view and the retrieval from the join. In a highly active concurrent environment this would probably not be the case.

```
SQL> SELECT COUNT(*) FROM(SELECT * FROM glv WHERE
  generalledger_id = 500000);

  COUNT(*)
----------
         1

Elapsed: 00:00:00.00

SQL> SELECT COUNT(*) FROM (SELECT gl.generalledger_id,
  coa.coa#, t.text
AS type, st.text AS subtype, coa.text as coa, gl.dr, gl.cr,
gl.dte
  2  FROM generalledger gl JOIN coa ON(gl.coa# = coa.coa#)
  3   JOIN type t USING(type)
  4    JOIN subtype st USING(subtype)
  5  WHERE generalledger_id = 500000);

  COUNT(*)
----------
         1

Elapsed: 00:00:00.00
```

Views can now have constraints, including primary and foreign key constraints. These may help performance of data retrieval from views. However, assuming that views are created for coding development simplicity and not security, adding complexity to a view would negate the simplification issue.

The exception to views re-executing every time they are queried is a Materialized View. A Materialized View is a separate database

object in Oracle Database and stores the results of a query. Thus when a Materialized View is queried, data is extracted from the view and not the underlying objects in the query. Materialized Views are read only and intended for use in data warehousing and replication.

Views are not performance friendly! For the sake of performance do not use views. Some applications are built with multiple layers of views. This type of application design is often application convenient and can produce disappointing results with respect to database performance. There is simply too much metadata in the shared pool. A brute force method of resolving selection of rows from multiple layered sets of views is to use a form of the FROM clause in the SELECT statement with the ONLY clause included as shown in the following syntax. The ONLY clause will not retrieve rows from subset views.

```
SELECT … FROM ONLY (query) …
```

6.9 Temporary Tables

In years past in traditional relational databases temporary tables were created as shell structures to contain temporarily generated data, usually for the purposes of reporting. Oracle Database has the ability to create tables containing temporary rows. Rows created in temporary table objects are created locally to a session, somewhat managed by the database. It is more efficient to use Oracle Database temporary table objects in some circumstances. Creating traditional temporary tables, filling them, emptying them, and storing or deleting their rows is unlikely to perform better than Oracle Database temporary table objects.

6.10 Resorting to PL/SQL

PL/SQL is an acronym for Programming Language for SQL. From a purely programming perspective PL/SQL is really SQL with programming logic wrapper controls, amongst numerous other bells and whistles. PL/SQL is a very primitive programming language at best. Beware of writing entire applications using PL/SQL. SQL is designed to retrieve sets of data from larger sets and is not procedural, sequential,

or object-like in nature. Programming languages are required to be one of those or a combination thereof. PL/SQL has its place as a relational database access language and not as a programming language. Additionally PL/SQL is interpretive which means it is slow!

10g PL/SQL objects are now stored in compiled form in binary object variables or BLOB objects. This helps PL/SQL procedure execution performance.

Does PL/SQL allow for faster performance? The short answer is no. The long answer is as follows. PL/SQL allows much greater control over SQL coding than simple SQL does. However, modern Oracle SQL is much more sophisticated than it was years ago and some of the aspects of PL/SQL used in the past allowing better performance in PL/SQL than SQL are effectively redundant, particularly with retrieval of data.

There are a number of reasons for resorting to PL/SQL:

- PL/SQL will not provide better performance but will allow a breakdown of complexity. Breaking down complexity can allow easier tuning of SQL statements through better control of cursors.
- An obvious benefit of PL/SQL is the ability to build stored procedures and triggers. Triggers should be used sparingly and avoided for implementation of Referential Integrity; constraints are much faster. In the case of stored procedures the obvious benefits are centralized control and potential performance increases because stored procedure code is executed on the server. Execution on a database server reduces network traffic.
- There are some situations where it is impossible to code SQL code using SQL alone and thus PL/SQL has to be resorted to. As Oracle Corporation has developed SQL this has become less frequent.
- Some exceedingly complex SQL code can benefit from the use of PL/SQL instead of SQL. Some SQL code can become so complex that it is impossible to tune or even code in SQL and hence PL/SQL becomes the faster option. Using the DECODE function is similar to control structures such as IF and CASE statements. PL/SQL allows all the appropriate control structures.
- PL/SQL packages can be cached into memory to avoid re-parsing.

PL/SQL can provide better program coding control of cursors plus execution and maintenance from a central, single point. Perhaps the most significant benefit of resorting to PL/SQL is a potential reduction in complexity. Once again, as with views, reducing complexity is not necessarily conducive to performance tuning. In fact simplicity, as with over-Normalization can often hurt performance in a relational database. Thus further discussion of tuning SQL retrieval code using PL/SQL is largely irrelevant.

Tuning PL/SQL is a programming task and has little to do with coding well-performing SQL code. The obvious programming points to note are as follows:

- Do not process more in a loop or a cursor than is necessary. Break out of loops when no more processing is required.
- Large IF statements should be replaced with CASE statements or appropriate breaks should be used. Traditionally CASE statements are faster than IF statements.
- Recursion using embedded procedures creates very elegant code but in PL/SQL can cause performance problems, especially where transactions are not completed. PL/SQL is not executed as compiled code in the sense that compiled C code is. PL/SQL compilation involves syntax parsing and interpretation; no binary copy is created. When PL/SQL code is executed it is interpreted and not executed from a compiled binary form.

(10g) PL/SQL objects are now stored in compiled form in binary object variables or BLOB objects. This helps PL/SQL procedure execution performance.

- The WHERE CURRENT OF clause can refer to a cursor ROWID, giving direct pointer access to a cursor row in memory. The RETURNING INTO clause can help with reducing the need for subsequent SELECT statements further on in a procedure by returning values into variables.
- Explicit cursors can be faster than implicit cursors but are more difficult to code.
- PL/SQL has three parameter formats for passing variables in and out of procedures. IN and OUT pass values in and out of

procedures, respectively. IN OUT passes a pointer both in and out of a procedure. Use only what is needed.

- As with SQL code embedded in applications, PL/SQL procedures of any kind can use bind variables. Using bind variables can have a profound effect on increasing performance by lowering parsing requirements in the library cache.

6.10.1 Tuning DML in PL/SQL

There are two interesting points to note in relation to performance of DML statements in PL/SQL. Other than what has already been covered in this chapter there is little else which can be done to tune DML (INSERT, UPDATE, DELETE, MERGE) or SELECT statements in PL/SQL.

The RETURNING INTO Clause

In the following example the RETURNING clause prevents the coder having to code an extra SELECT statement to find the identifier value required for the second INSERT statement. The same applies similarly to the second INSERT statement returning a second value for the DBMS_OUTPUT.PUT_LINE procedure.

```
DECLARE
      division_id division.division_id%TYPE;
      division_name division.name%TYPE;
      department_name department.name%TYPE;
BEGIN
      INSERT INTO division(division_id, name)
      VALUES(division_seq.NEXTVAL, 'A New Division')
      RETURNING division_id, name INTO division_id,
        division_name;

      INSERT INTO department(department_id, division_id, name)
      VALUES(department_seq.NEXTVAL, division_id, 'A New
        Department')
      RETURNING name INTO department_name;

      DBMS_OUTPUT.PUT_LINE('Added : '||division_name
            ||', '||department_name);
END;
/
```

(10g) The RETURNING clause can be used to return collections.

6.10.2 When to Resort to PL/SQL and Cursors

In general, SQL in Oracle Database is now powerful enough to deal with almost any requirement. Some occasions do call for substituting SQL code nested loop type queries with PL/SQL. PL/SQL provides better programming control and allows much easier management of highly complex SQL code. It is sometimes the case that better performance will be gained using PL/SQL to control nested SQL statements.

PL/SQL can sometimes be faster when compiled, pinned, and executed on the database server, depending on the application. The goal of stored procedures is to minimize on network traffic. With the speed of modern network connections available this is not necessarily an issue anymore.

So what do I mean by resorting to PL/SQL and cursors? After all I did not just state "Resorting to PL/SQL". So what am I talking about? Look at this example.

```
EXPLAIN PLAN SET statement_id='TEST' FOR
      SELECT * FROM coa NATURAL JOIN generalledger;
```

This query plan shows the COA table in the outer loop and the GeneralLedger table in the inner loop.

```
Query                                    Cost      Rows      Bytes
---------------------------------------- --------  --------  --------
1.   SELECT STATEMENT on                  1642   1068929   50239663
2.    HASH JOIN on                       1642   1068929   50239663
3.     TABLE ACCESS FULL on COA             2        55       1320
3.      TABLE ACCESS FULL on
          GENERALLEDGER                  1128   1068929   24585367
```

To convert the SQL statement shown previously to PL/SQL I would need PL/SQL code something similar to that shown here.

```
DECLARE
   CURSOR cCOA IS SELECT * FROM coa;
   TYPE tGL IS REF CURSOR RETURN generalledger%ROWTYPE;
   cGLs tGL;
   rGL generalledger%ROWTYPE;
```

```
BEGIN
   FOR rCOA IN cCOA LOOP
       OPEN cGLs FOR SELECT * FROM generalledger WHERE
         coa# = rCOA.coa#;
       LOOP
          FETCH cGLs INTO rGL;
          EXIT WHEN cGLs%NOTFOUND;
          DBMS_OUTPUT.PUT_LINE(
             rCOA.coa#||' '||
             TO_CHAR(rGL.dte)||' '||
             TO_CHAR(rGL.dr)||' '||
             TO_CHAR(rGL.cr));
       END LOOP;
       CLOSE cGLs;
   END LOOP;
EXCEPTION WHEN OTHERS THEN CLOSE cGLs;
END;
/
```

In general, PL/SQL should only replace SQL when coding simply cannot be achieved in SQL, it is too complex and convoluted for SQL, or centralization on the database server is required.

6.10.3 Java or PL/SQL?

Java can be used to construct stored procedures in much the same way that PL/SQL can. When to use Java? That is a question with a very simple answer. Use Java when not accessing the database and writing code which is computationally heavy. What does this mean? When answering a question such as this I always prefer to go back into the roots of a programming language or a database. In other words, what were PL/SQL and Java originally built for? What is their purpose? Let's start by looking at PL/SQL.

PL/SQL is effectively a primitive programming language and is purely an extension of SQL. SQL was originally built purely for the purpose of accessing data in a relational database. Therefore, it follows that PL/SQL is of the same ilk as SQL. PL/SQL was originally devised to create stored procedures in Oracle Database. Stored procedures were devised for relational databases in general, not just Oracle Database, to allow for coding of self-contained blocks of transaction-based SQL code. Those blocks of code are executed on the database server thus minimizing on network traffic. PL/SQL is

much richer than stored procedure languages used in other relational databases. PL/SQL can be used to code complexity. This takes us to Java.

Why use Java? Java is an object-oriented programming language built for coding of highly complex front-end and back-end applications. If you know anything about objects at all you will understand that objects in programming are superb at handling complexity. It is in the very nature of objects to handle complexity by breaking everything down into its most simplistic parts. Java can be used to handle highly complex coding sequences and can be used to create Oracle Database stored procedures, much in the same way that PL/SQL can. Java is much more powerful than PL/SQL in doing lots of computations. Java is better than PL/SQL at anything that does not involve accessing the database, especially complex code. Coding requirements when accessing a database are trivial in relation to the complex routines and functions required by applications-level coding.

There is one small but fairly common problem with using Java. Java is object oriented. Oracle Database is relational. Object and relational methodologies do not mix well. Many Java applications, due to their object nature, break things into very small, easily manageable pieces. Object-oriented design is all about easy management of complexity. In relational terms management of complexity by severe break-down and object "black boxing" is an incredible level of Normalization. Too much Normalization leads to too much granularity and usually very slow performance. What commonly happens with Java applications is one or all of a number of things. These are some of the possibilities.

- Pre-loading of large data sets.
- Separation of parts of SQL statements into separate commands sent to the database. For instance, a SELECT statement could be submitted and then filtering would be performed in the Java application itself. This leads to lots of full table scans and a plethora of other performance problems.
- Sometimes object design is imposed onto the database to such an extent as to continually connect to and disconnect from the database for every SQL code execution. This is very heavy on database resource consumption.

So what's the answer to the nagging question of using Java or PL/SQL? The answer is to use both, if your skills set permits it. *Java can be used to handle any kind of complex code not accessing the database. PL/SQL should be used to access the database*. If you cannot or do not wish to use a mix and prefer Java then be aware that a relational database is not able to manage the complete and total "black box" breakdown into easily definable and understandable individual objects. Do not attempt to impose an object structure on a relational or furthermore even an object-relational database: they are not designed for that level of granularity.

6.11 Replacing DELETE with TRUNCATE

The DDL TRUNCATE command is faster than using the DELETE command when deleting all the rows from a table. The TRUNCATE command automatically commits and does not create any log entries or rollback. The obvious problem with TRUNCATE is that an error cannot be undone.

6.12 Object and Relational Conflicts

Relational and object data model structures are completely different to each other. There is great potential for conflict when combining these two methodologies into what is called an object-relational database. These conflicts can hamper performance in a relational database such as Oracle Database.

6.12.1 Large Binary Objects in a Relational Database

The biggest benefit to the combination of objects and relations is that of including large binary objects into a relational structure, such as multimedia objects. However, there is a twist. In both relational and object databases the most efficient storage method for multimedia objects is to use a BFILENAME pointer. A BFILENAME pointer does not store the object itself in the database but contains a path and file name for the object. The object is stored externally to the database in the file system. Storing large binary objects in any database

is inefficient because those binary objects can span multiple blocks, both Oracle Database blocks and operating system blocks. Even with Oracle9*i* Database multiple block sizes and specific storage structures for LOB datatypes large multimedia objects will span more than one block. Storing data into a database which spans more than a single block is effectively row chaining where a row is "chained" from one block to another. Chaining is not really an issue if the blocks are physically next to each other. Also the DB_FILE_MULTIBLOCK_READ_COUNT parameter can help to counteract this. The fact is contiguous, defragmented block storage is extremely unlikely in any database environment. It is usually best to store large singular binary objects in the file system.

10g BIGFILE tablespaces can possibly help to alleviate these issues somewhat.

6.12.2 Object-Relational Collections

Including TABLE and VARRAY collections inside relational tables and within PL/SQL is generally a very bad idea. Collections are an object methodological approach to object data abstraction and encapsulation. Object methodologies are completely opposed to those of relational databases. An object structure is spherical allowing access from any point to any point within an entire database. A relational database is more two-dimensional in nature and requires that access to data be passed through or down semi-hierarchical structures or into subset parts of a data model. Since PL/SQL is merely an extension of SQL, and SQL is a purely relational database access language, any type of object coding is best avoided using PL/SQL. If you want to write object code use Java. Java is built for object structures.

The same approach applies to including object structures in tables. There are various instances in which collection substructures can be utilized in relational database tables. 1st Normal Form master detail and 2nd Normal Form foreign key static table inclusion relationships are possibilities. However, storage structures remain applicable to a relational methodology and are still ultimately stored in rows (tuples). Tables and rows are two-dimensional. Object structures are

multi-dimensional. The two do not fit together. If you must contain collections in tables in Oracle Database there are various object collection types that could be used. Associative arrays are the most efficient.

- *Nested Table*. A nested table is a dynamic array. A dynamic array is a pointer. A pointer does not require a specific size limit and thus the use of the term dynamic. Dynamic implies that it can have any number of rows or entries in its array.
- *VARRAY*. A VARRAY is a fixed-length array. A fixed-length array is a reserved chunk of memory saved for multiple array rows. Unlike a dynamic array a fixed-length array has space in memory reserved for the whole array structure.
- *Associative Array*. An associative array is an indexed dynamic array and can potentially be accessed faster than a nested table because it can be accessed using an index.

Problems with objects in relational models are more often than not a misunderstanding of object-modeling techniques. It is common knowledge that the approach required for object data modeling is completely different to that of relational modeling. The same is very likely to apply to object-relational modeling using object types such as nested table, VARRAY or associative array collections in Oracle Database tables. Using these collection data types requires object and relational modeling skills, not only relational modeling skills.

That ends this chapter on examining the basics of how to create efficient SQL. In the next chapter we will look at indexing and start to delve a little deeper into Oracle Database specifics.

21

Tools and Utilities

There are a multitude of tools available to both monitor and tune Oracle installations. This chapter will simply introduce some of those tools and categorize what each can do. It is best to explain some of the various tools graphically. Let's begin with Oracle Enterprise Manager. In Chapter 10 of Part II we looked at the SQL code tuning aspects of Oracle Enterprise Manager; this chapter attempts to focus on the physical tuning aspects.

21.1 Oracle Enterprise Manager

Oracle Enterprise Manager in relation to physical tuning can be divided into two parts. These two parts contain diagnostic and tuning tools. Diagnosis usually implies monitoring for potential problems and sometimes repairing them; tuning and analysis can even automatically repair possible performance problems.

21.1.1 Diagnostics Pack

With respect to physical tuning the interesting parts of the diagnostics pack are as follows:

- *Event Monitoring*. Allows definition, automated detection of and potential automated "FixIt" jobs, which can automatically correct problems.
- *Lock Monitoring*. Allows monitoring and resolution of locks in a database.

- *TopSessions*. Detects and displays the top best- or worst-performing sessions ordered by a large number of different possible sorting parameters.
- *TopSQL*. Detects and displays the top best- and worst-performing SQL statements, again ordered by a large number of different possible sorting parameters.
- *Performance Manager*. Performance Manager wraps everything together such as details contained in the TopSessions and TopSQL tools, with extra functionality and GUI usability.

ⓘ The ultimate of Oracle Enterprise Manager is a GUI window interface into the Oracle Database Wait Event Interface.

Event Monitoring

Event monitoring covers areas such as performance, resource usage, how space is used, amongst many other areas. There are an absolute of plethora of events which can be included for automated detection and potential "FixIt" action to resolve problems. Figure 21.1 shows a very small subsection of the fully user definable event detection tool. The event detection construction tool is accessible from the Event menu in the Oracle Enterprise Manager Console.

10g Oracle Enterprise Manager repository, events and jobs are automatically configured.

Lock Monitoring

Figure 21.2 shows lists of locks on a very busy database. Note the drilldown menu with various options including the option to kill off a session. Sometimes it is necessary to kill a session causing potentially damaging problems.

TopSessions

The TopSessions tool allows ordered visualization of statistical performance information where the best or worst sessions can be

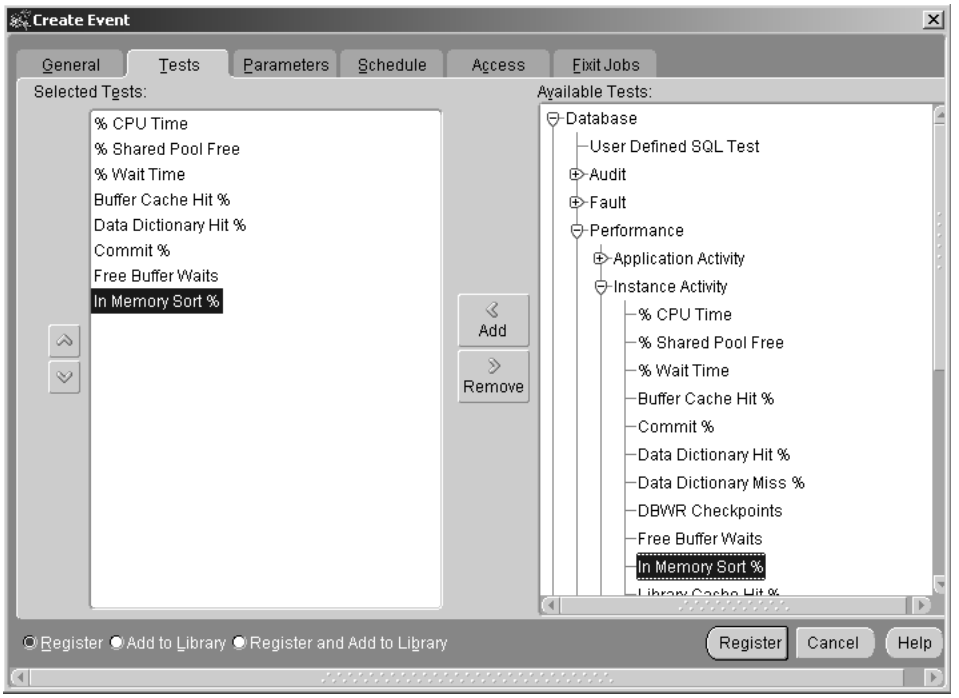


Figure 21.1 *Defining Events for Detection and Resolution*

displayed first. Figure 21.3 shows the first 10 sessions executing the most physical reads.

TopSQL

TopSQL is similar to the TopSessions tool where the TopSQL tool detects and displays the best- or worst-performing SQL statements in a large number of different possible orders. An example of TopSQL is shown in Figure 21.4.

10g TopSQL is much improved in relation to more usable reporting capabilities.

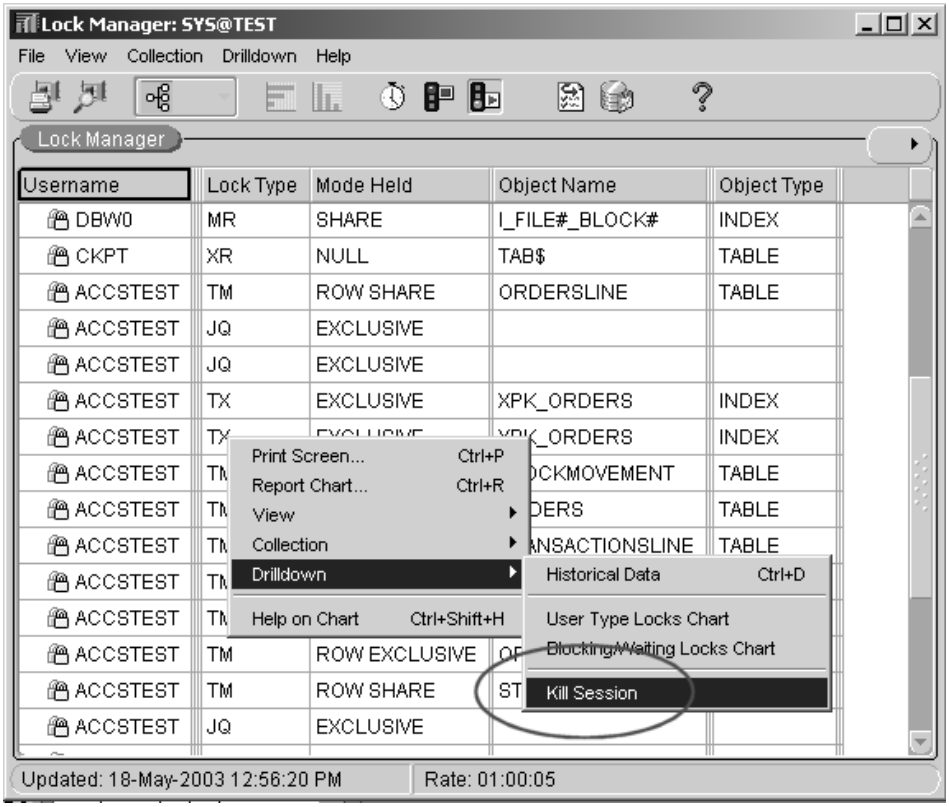


Figure 21.2 *Monitoring and Managing Locking*

Performance Manager

The Performance Manager is a very comprehensive tool. It allows an intensive GUI picture of an environment with every possible monitoring and performance metric imaginable. The Performance Manager appears complex but it is comprehensive and very easily usable.

Figure 21.5 shows the Performance Overview screen. This screen is accessible from the Oracle Enterprise Manager Console on the Tools menu. Select the Diagnostic Pack submenu. The Performance Overview interface gives an overall picture of database performance

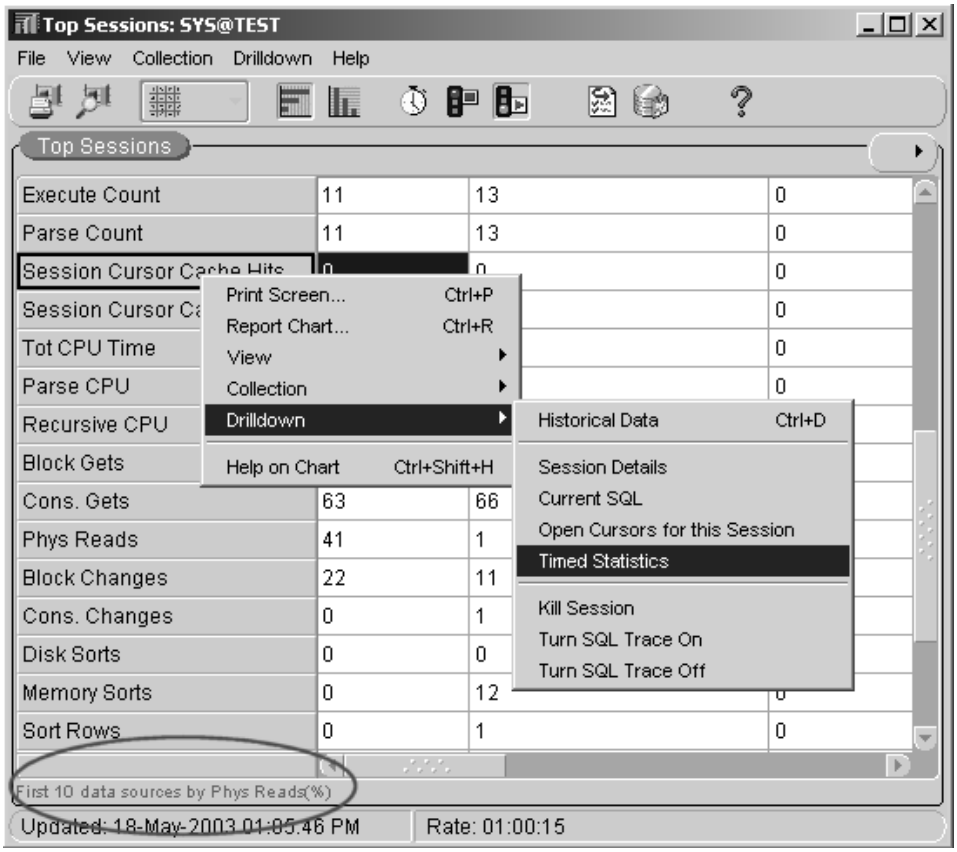


Figure 21.3 *TopSessions Monitoring in a Specified Order*

and is highly adaptable depending on requirements and what you want to monitor.

10g Performance Overview charts are much improved allowing monitoring and diagnostics checking from a central point using an HTML interface.

The Performance Manager allows simple drill-down further into the separate subset areas of those shown in Performance Overview interface, to the point and beyond of drilling down

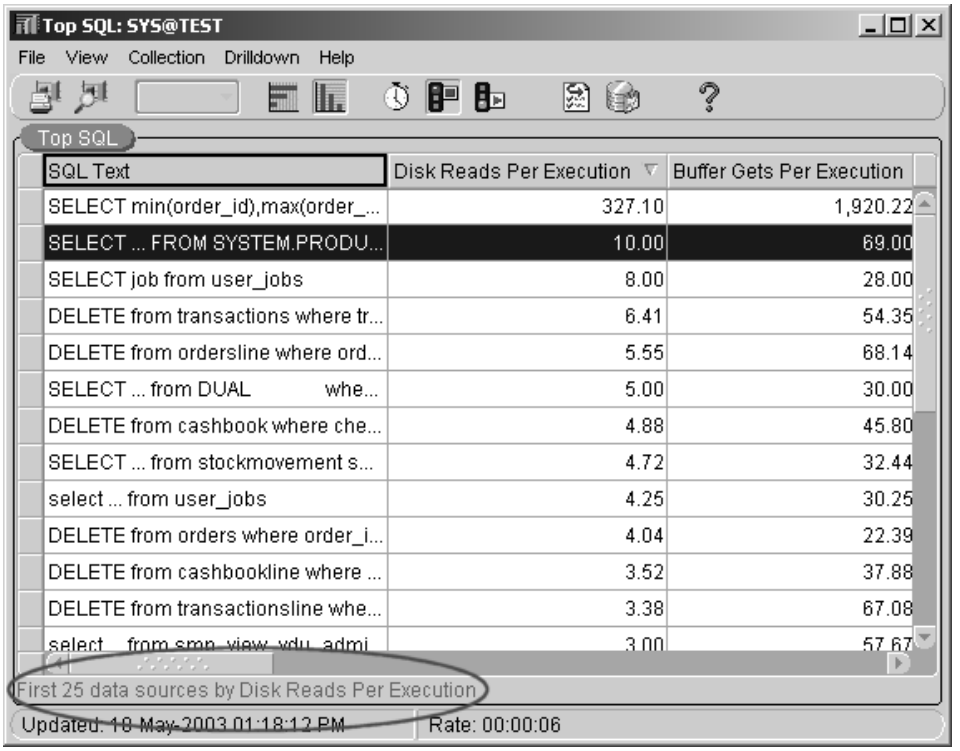


Figure 21.4 *TopSQL Statements*

into the Oracle Database Wait Event Interface. The Oracle Database Wait Event Interface will be covered in detail in the next chapter. Figure 21.6 shows a small part of the possible types of information which can be monitored with the Performance Manager.

Figure 21.7 shows a drilldown into latch get/miss rates.

Finally Figure 21.8 shows a detailed analysis of latches.

21.1.2 Tuning Pack

As far as physical tuning is concerned only the Tablespace Map and Reorg Wizard tools are of interest. Other parts of the Oracle

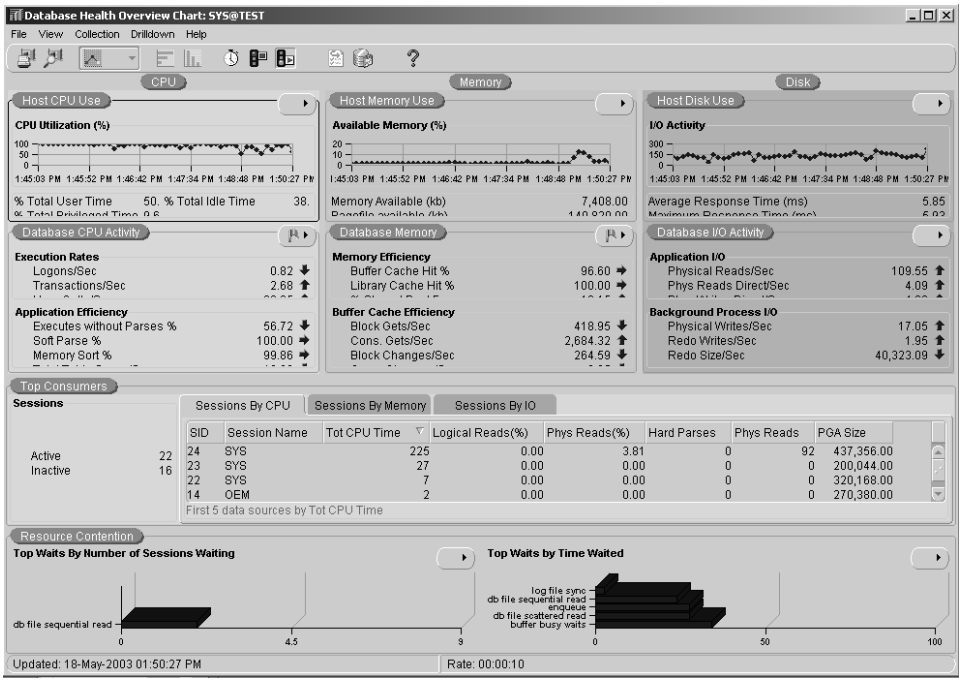


Figure 21.5 *The Performance Overview Tool*

Enterprise Manager tuning pack were covered in Chapter 10 of Part II of this book.

Tablespace Map and the Reorg Wizard

The tablespace map presents a graphical representation of physical data distribution within tablespaces. After executing a tablespace analysis in the INDX tablespace of my database, two indexes are recommended for reorganization, as shown in Figure 21.9.

As can be seen in the drop-down menu in Figure 21.9 various options are available:

- *Tablespace Analysis*. Analyze a tablespace for problems such as fragmentation.
- *Reorganize Tablespace*. Execute the Reorg Wizard.

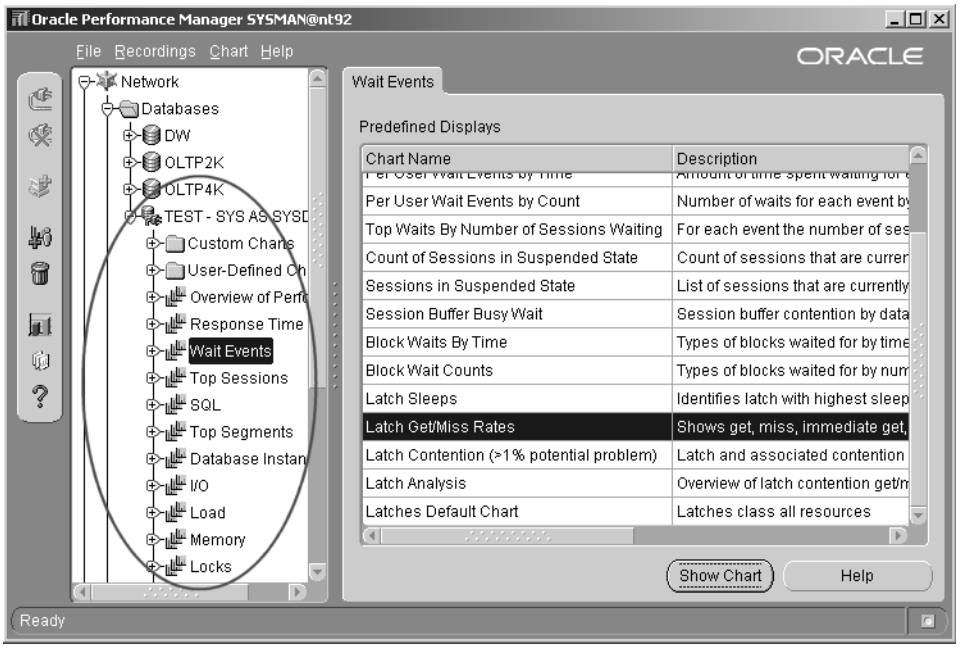
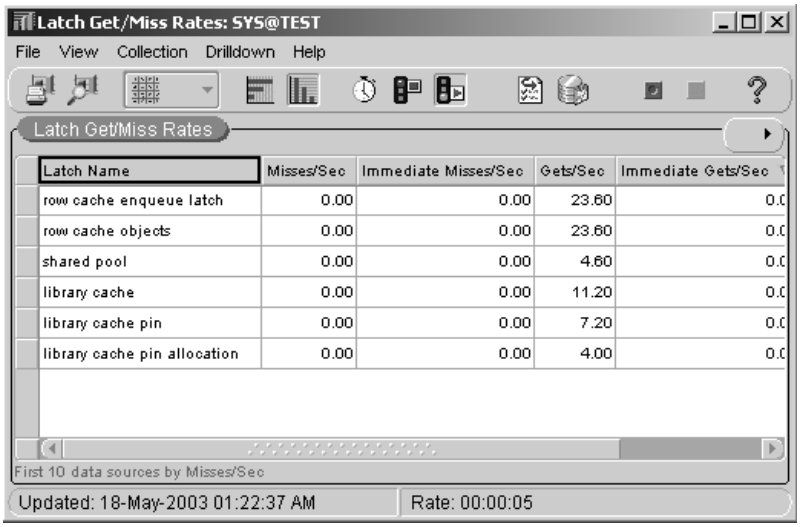


Figure 21.6 *The Performance Manager Main Screen*

Figure 21.7 *Latch Get/Miss Rates Drilldown*



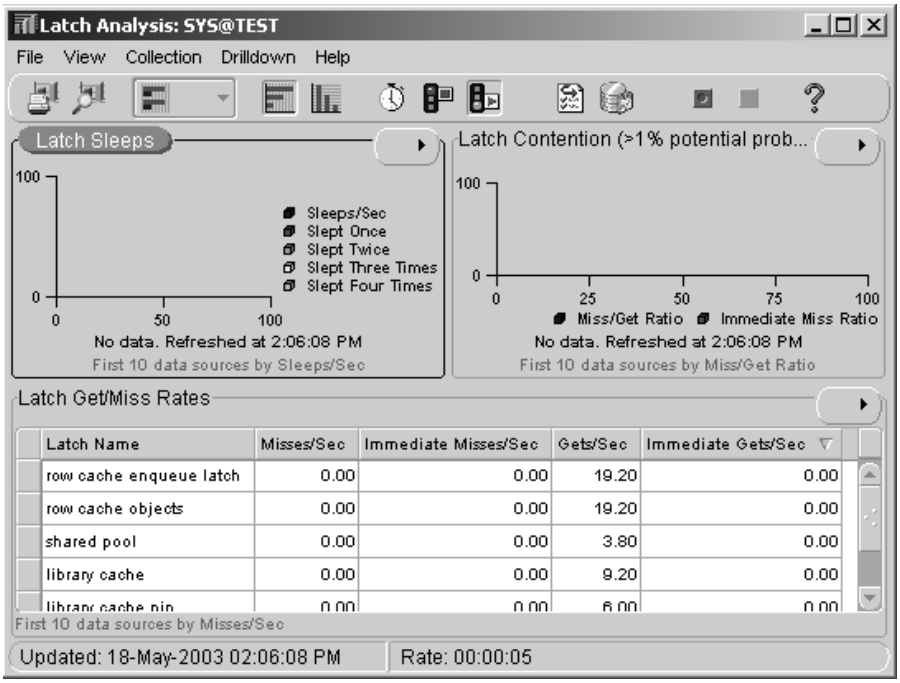


Figure 21.8 *Latch Analysis*

- *Reorganize Selected Segments*. Execute reorganization on selected indexes (segments) only.
- *Coalesce Free Extents*. Coalescence attempts to merge empty adjacent physical areas into single reusable chunks of disk storage space. In the past, I have not found coalescence to be particularly useful.

Figure 21.10 simply shows the same screen as in Figure 21.9 except with the Tablespace Analysis Report tab selected. The tablespace analysis report describes exactly why the table analysis process considers these indexes the cause for reorganization, namely fragmentation.

The result of execution of the Reorg Wizard and tablespace coalescence on the INDX tablespace is shown in Figure 21.11. The actual reorganization process is created as an instantly executed or

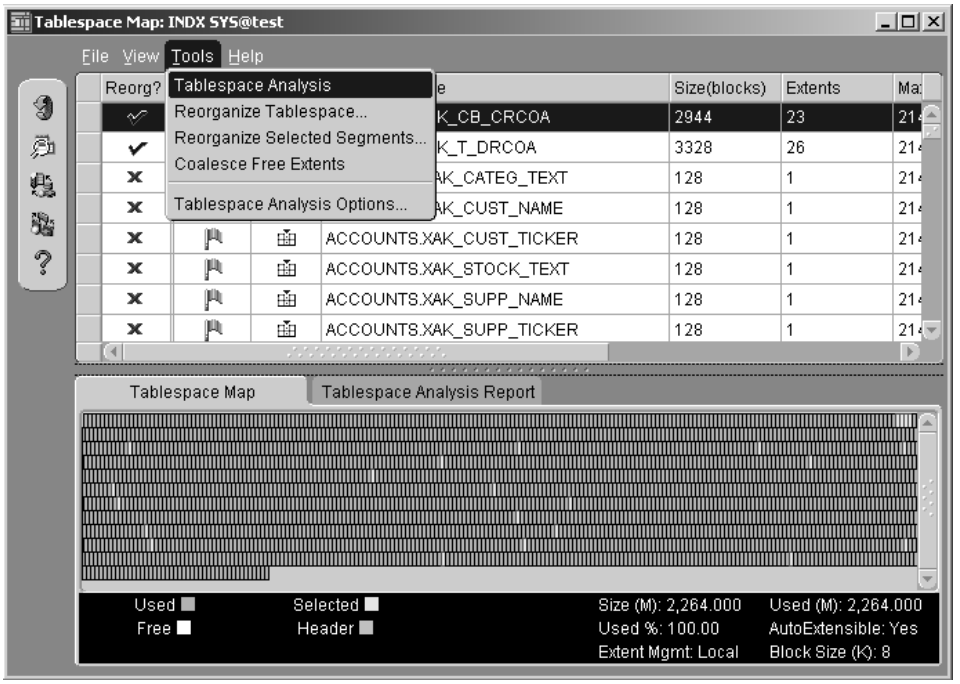


Figure 21.9 *After Execution of Tablespace Analysis*

future scheduled job, to be executed inside the job control scheduling system, visible in the Oracle Enterprise Manager Console job scheduling screen.

That's enough about Oracle Enterprise Manager. The best use of Oracle Enterprise Manager for physical and configuration tuning is diagnosis and analysis of potential bottlenecks using the GUI drill-down access paths into the Oracle Database Wait Event Interface. The Oracle Database Wait Event Interface will be covered in detail in the next chapter. There are many other non-Oracle Corporation tools for monitoring and tuning Oracle databases. Spotlight is one of these tools.

21.2 Spotlight

Spotlight on Oracle is an excellent tool for real-time monitoring and reactive, not predictive, tuning of busy production databases.

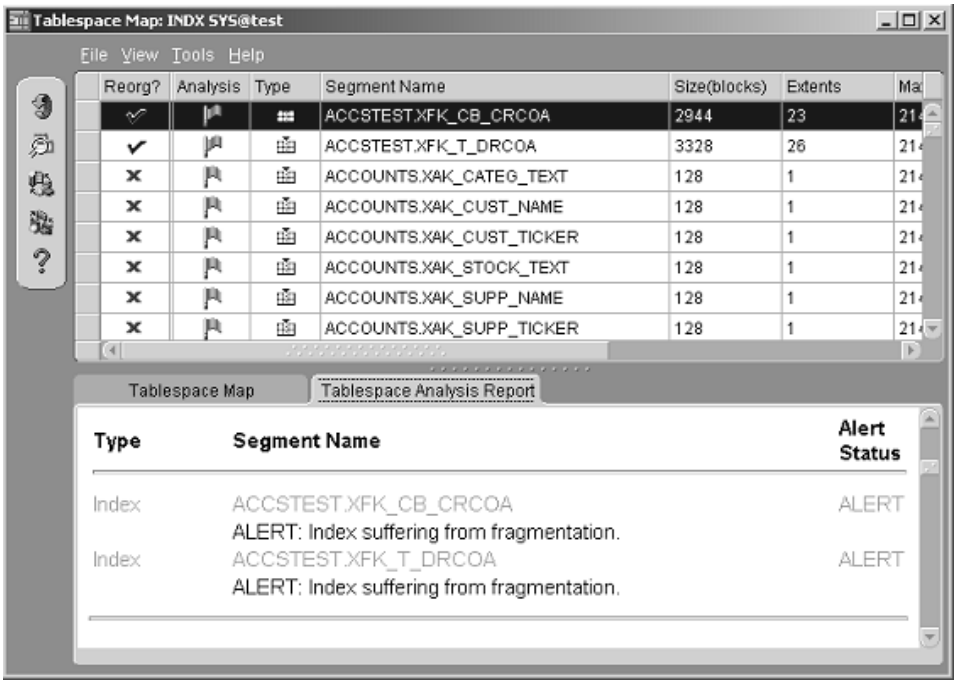


Figure 21.10 *What Should be Reorganized and Why*

With respect to physical tuning Spotlight can do much and its GUI is of excellent caliber. Spotlight can be used for SQL code tuning as well. However, it is best used as a monitoring and physical tuning advisory tool.

Figure 21.12 shows the primary screen in Spotlight on Oracle, used for monitoring a busy Oracle production database. Different colors represent different levels of alert: green being OK, red being serious, or just panic!

There are an enormous number of subset screens in Spotlight on Oracle, for every aspect of Oracle Database tuning imaginable. Figure 21.13 shows a screen displaying real time usage of the Shared Global Area (SGA) for a database.

Figure 21.14 shows a mixture of various views from Spotlight with both textual and instantly recognizable graphical displays.

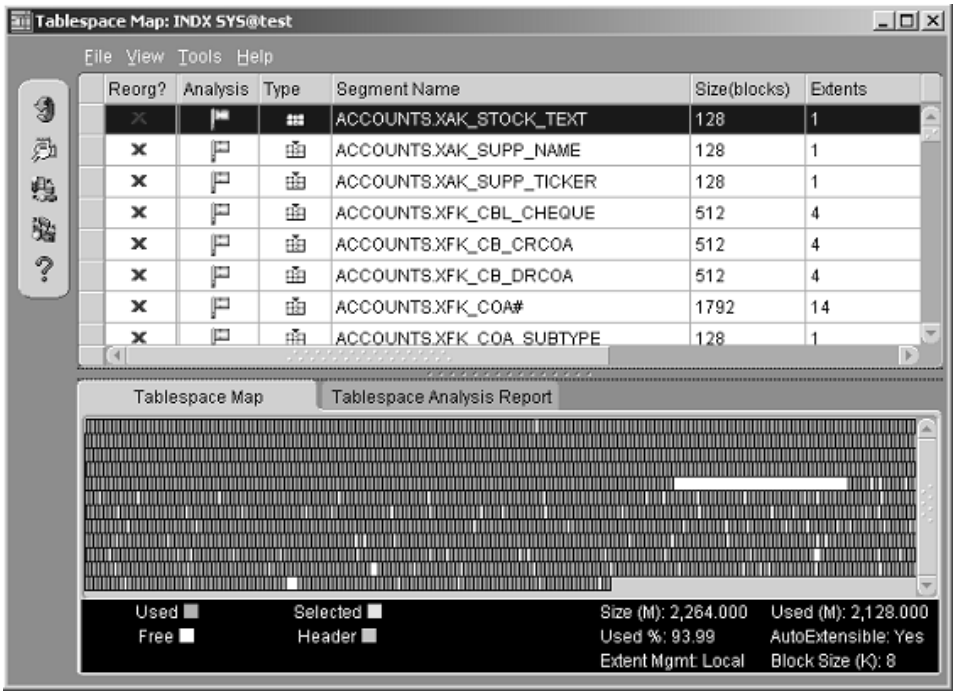


Figure 21.11 *After Reorganization and Coalescence on the INDX Tablespace*

Spotlight is a very comprehensive tuning and monitoring tool for Oracle Database, especially in the area of real time production database monitoring. As already stated many non-Oracle Corporation produced Oracle database monitoring and tuning tools are available. Spotlight is one of the most comprehensive and useful of those tools I have used in relation to its price.

21.3 Operating System Tools

21.3.1 Windows Performance Monitor

The Windows Performance Monitor is useful for monitoring operating system-level performance. The GUI can be started up on a Windows 2K server in the Administrative Tools icon on the

Figure 21.12 *Spotlight on Oracle Main Monitoring Screen*

Control Panel. This tool allows graphical display of all aspects of hardware and operating system-level activity. Figure 21.15 shows a picture of a dual-CPU server displaying both processors, memory, and network activity for a relatively idle database server.

Figure 21.16 on the other hand shows the Windows Performance Monitor GUI for a highly active single-CPU system. The Windows Task Manager snapshot is added to show how busy this system was when these screenshots were taken.

There are a multitude of command line and GUI tools used for operating system monitoring and tuning in other operating systems such as Unix or Linux.

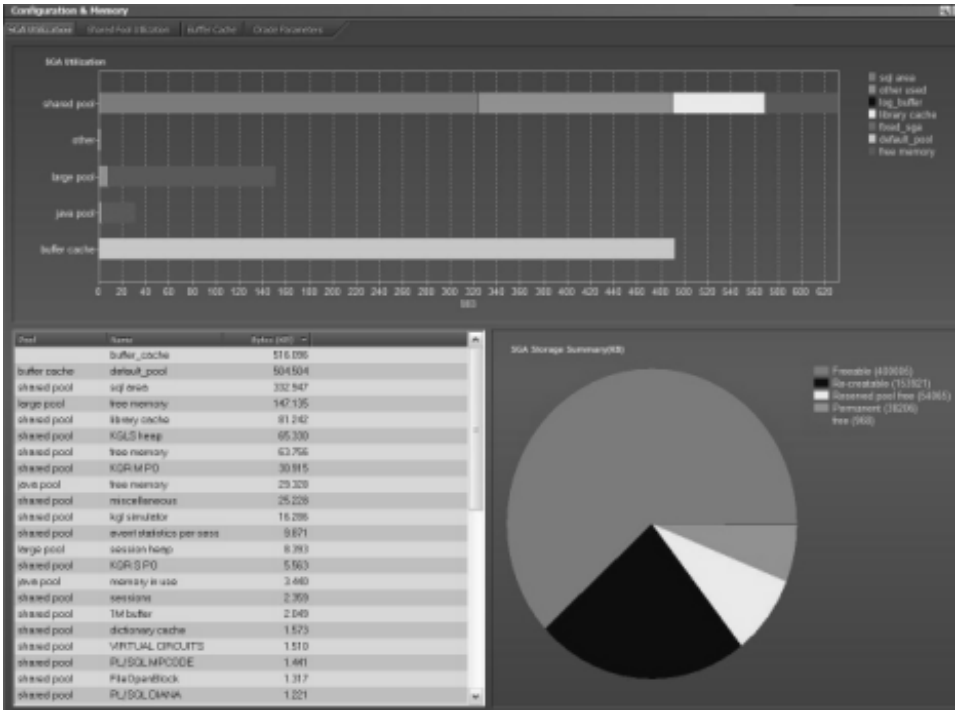


Figure 21.13 *Spotlight SGA View*

21.3.2 Unix Utilities

There are a plethora of tools used in all flavors of Unix and Linux operating systems, both command line utilities and GUI-based monitoring tools. These tools are some of the command line monitoring tools used on Solaris.

- *CPU Usage*. sar, vmstat, mpstat, iostat.
- *Disk I/O Activity*. sar, iostat.
- *Memory Usage*. sar, vmstat.
- *Network Usage*. netstat.

Since I am not an experienced systems administrator I see no point in attempting to explain the detailed syntax and use of these tools.

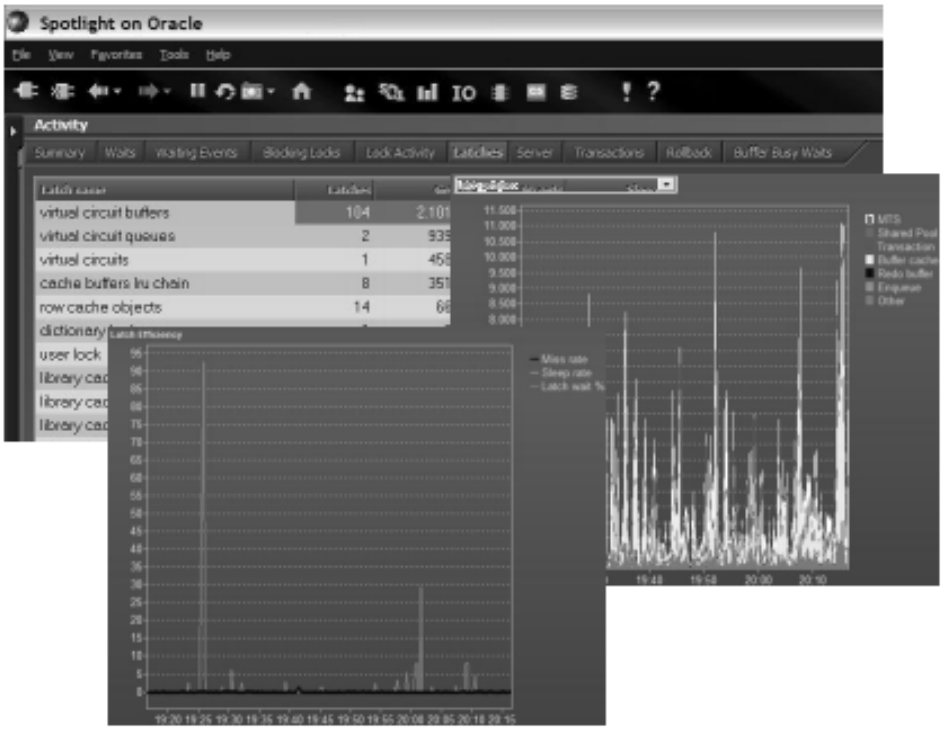


Figure 21.14 *Various Spotlight Views*

21.4 Other Utilities and Tools

21.4.1 Import, Export, and SQL*Loader

The Export and SQL*Loader utilities can be executed in DIRECT mode. DIRECT mode implies that the SQL engine, executing INSERT statements, is completely bypassed and data is appended directly to datafiles on disk. DIRECT mode is much faster than passing through the SQL engine. Import and Export have a parameter called BUFFER. The BUFFER parameter limits chunks of rows processed. In very busy environments using the BUFFER parameter can limit the impact on other database services.

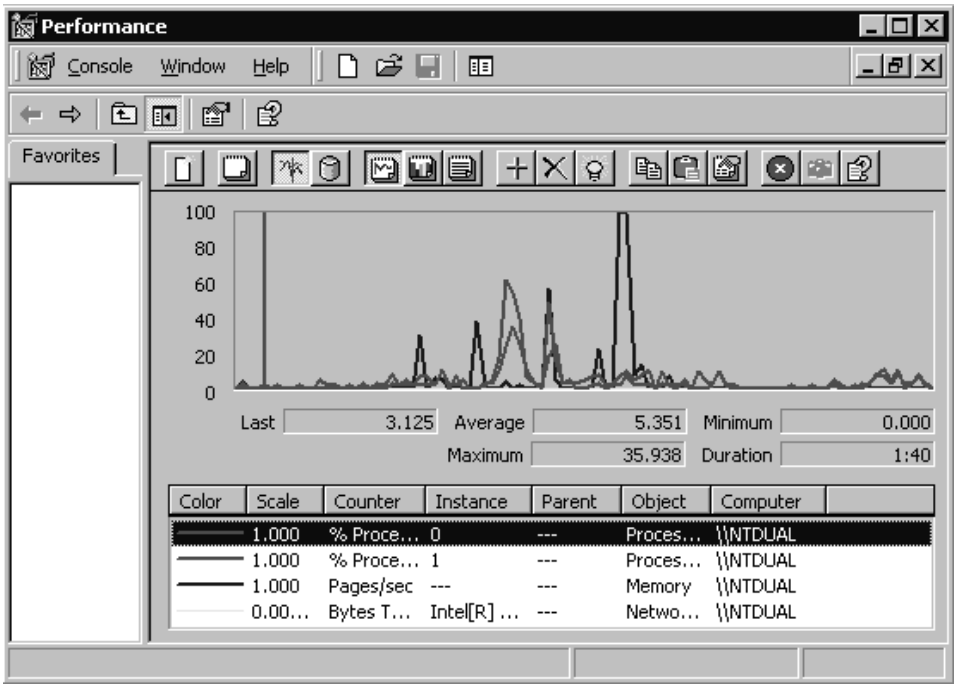


Figure 21.15 *The Windows Performance Monitor*

ⓘ Do not use the Import and Export Utilities for Backup and Recovery. Generally only use Export to copy individual static tables you are absolutely sure will not change. Additionally using a previously FULL Export backup file to Import and recreate a lost database is incredibly inefficient. I have seen a 200 Gb database reconstruction using a FULL database Export take an entire weekend to complete. This particular database was apparently destroyed by a junior DBA executing a script to drop all users in cascade mode, by mistake of course. Oops!

10g Data Pump Import and Data Pump Export have much better performance than the Import and Export utilities. Data Pump allows parallel execution, is self-tuning, and does not require compression into a single extent.

SQL*Loader is an excellent tool for loading data into a database. SQL*Loader is very easy to use and can map data files in text

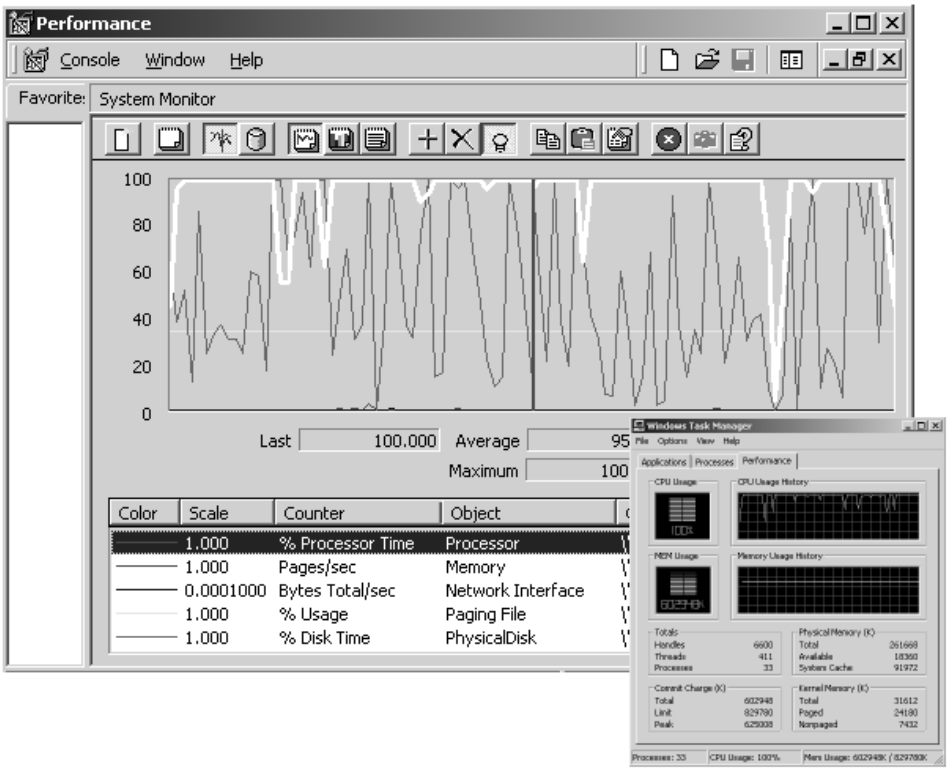


Figure 21.16 *This is a Very Busy Server*

format directly into database tables. Use SQL*Loader as an alternative option to that of loading tables using large quantities of INSERT statements, it can give performance increases in the thousands. Use SQL*Loader as it is very easy to learn to use.

10g SQL*Loader can now cache dates reducing conversions on duplicated date values.

21.4.2 Resource Management and Profiling

Resources can be managed and effectively shared in Oracle Database to create round-robin-type queues. Hardware resources such as CPU time can be evenly distributed across multiple profiles.

Profiles contain groups of users or functionality sets. Different activities can be given priority at different times of day. Perhaps OLTP activity can be favored during the daytime and backup or batch processing activities favored after-hours. This of course assumes that you do not have to maintain a constant level of 24×7 database availability. There is a small amount of resource overhead with respect to Oracle Database internal implementation and execution of resource management.

21.4.3 Recovery Manager (RMAN)

RMAN or Recovery Manager is a slightly more power-user-oriented backup utility when compared to using backup mode tablespace datafile copies. However, RMAN simplifies backup and recovery processing and DBA involvement. Additionally RMAN can be executed in parallel.

21.4.4 STATSPACK

People have written and published entire books about STATSPACK. STATSPACK is a comprehensive statistics monitoring and tuning analysis tool, the next generation of the UTLBSTAT.sql and UTLESTAT.sql tuning scripts. In addition STATSPACK can be used to store statistical information in the database in a special repository for later comparison and analysis between different statistics set collections. Therefore, when a performance problem occurs a current snapshot can be compared against a previously obtained baseline snapshot, allowing for easy comparison and thus rapid diagnosis of the problem.

ⓘ STATSPACK is useful for analysis and detection of bottlenecks but using the Oracle Database Wait Event Interface and the Capacity Planner in Oracle Enterprise Manager is better. The Oracle Database Wait Event Interface will be covered in the next chapter, along with some use of STATSPACK.

STATSPACK is very easy to install, configure, and to use. The problem with it is that it produces enormous quantities of what

could amount to superfluous information. Wading through all the mounds of information produced by STATSPACK could be somewhat daunting to the tuning novice or someone who is trying to solve a problem in a hurry. Large amounts of information can even sometimes hide small things from an expert. The expression "trying to find a needle in a haystack" comes to mind.

STATSPACK is more useful for general performance tuning and analysis as opposed to attempting to find specific bottlenecks. The Oracle Database Wait Event Interface in Oracle Enterprise Manager is much more capable than STATSPACK in rapidly isolating problems.

To use STATSPACK a specific tablespace must be created:

```
CREATE TABLESPACE perfstat DATAFILE
      'ORACLE_HOME/<SID>/perfstat01.dbf'
      SIZE 25M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED
      EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT
        AUTO;
```

Do not create a user because the scripts will crash! Additionally you might want to set the ORACLE_SID variable if your database server has multiple databases. The following script will create STATSPACK goodies.

```
@ORACLE_HOME/rdbms/admin/spcreate.sql;
```

If the installation completely freaks out, the following script will drop everything created by SPCREATE.SQL so that you can start all over again.

```
@ORACLE_HOME/rdbms/admin/spdrop.sql;
```

Once installed take a snapshot of the database by executing these commands in SQL*Plus.

```
CONNECT perfstat/perfstat[@tnsname];
EXEC STATSPACK.SNAP;
```

The DBMS_JOBS package can be used to automate STATSPACK SNAP procedure executions on a periodical basis, as in the example script shown overleaf, which executes every 5 min. Different snapshot levels can be used between 0 and 10. 0 is not enough, 10 far too much, 5 is the default and 6 provides query execution plans. Since the large majority of performance problems are caused by poorly written SQL code I would recommend starting at snapshot level 6.

Snapshot levels are as follows:

- 0: simple performance statistics.
- 5: include SQL statements, which is the default level.
- 6: include SQL plans.
- 7: include segment statistics.
- 10: include parent and child latches.

ⓘ Running STATSPACK will affect performance so do not leave it running constantly.

```
DECLARE
 jobno NUMBER;
 i INTEGER DEFAULT 1;
BEGIN
 DBMS_JOB.SUBMIT(jobno,' STATSPACK.SNAP(I_SNAP_LEVEL=>6);'
       ,SYSDATE,'SYSDATE+1/288');
COMMIT;
END;
/
```

Remove all jobs using an anonymous procedure such as this.

```
DECLARE
      CURSOR cJobs IS SELECT job FROM user_jobs;
BEGIN
      FOR rJob IN cJobs LOOP
            DBMS_JOB.REMOVE(rJob.job);
      END LOOP;
END;
/
COMMIT;
```

Run a STATSPACK report using the following script. The script will prompt for two snapshots to execute between, comparing sets of statistics with each other.

```
@ORACLE_HOME/rdbms/admin/spreport.sql;
```

An SQL report can be executed on one SQL statement, searching for a bottleneck, where the hash value for the SQL code statement is found in the STATSPACK instance report.

```
@ORACLE_HOME/rdbms/admin/sprepsql.sql;
```

There are a number of other "SP*.SQL" STATSPACK scripts used for various functions. Two of those scripts are SPPURGE.SQL and SPTRUNCATE.SQL. Purging allows removal of a range of snapshots. If the database is bounced then snapshots cannot be taken across the database restart. Truncate simply removes all STATSPACK data.

STATSPACK can also have threshold value settings such that characteristics below threshold values will be ignored. Default STATSPACK parameter settings including threshold values are stored in the STATS$STATSPACK_PARAMETER table.

This is enough information about physical tuning tools and use of utilities. The next chapter will look at physical and configuration tuning from the perspective of finding bottlenecks using the Oracle Database Wait Event Interface.