

SELECT[®] Online

The Journal of the International Oracle Users Group

Select Journal
Home

Regular Columns

Past Issues

General Information

IOUG Home



Select Magazine - March 2002

Cost-Based Optimizer Problems and Solutions

By Mark Gurry

This article has been written to assist sites with applications having performance problems due to the existing SQL statements which may not be able to be changed. Typical examples of this type of site are those running ERP solutions such as PeopleSoft, Oracle Applications, SAP, Keystone, Siebel and others. The information in this article will also assist application developers who require a more detailed understanding of the cost-based optimizer to ensure that they are producing tuned code.

The DBAs at many sites running ERP packages that I tune are trying to find that magical parameter that solves all of their performance problems. I have seen sites with buffer caches up to a Gigabyte in size where performance was still abysmal. Hardware salespeople love sites like this. Even with the additional hardware, more often than not, the performance problem is not resolved. In almost all cases, when I investigate the performance problem, there are a large number of un-tuned SQL statements that are run often by the irate users.

The cost-based optimizer has been significantly improved from its initial version. My recommendation is that every site new to Oracle should be using the cost-based optimizer. I also recommend that sites currently using the rule-based optimizer have a plan in place for migrating to the cost-based optimizer. There are, however, some issues with the cost-based optimizer that you should be aware of. Each of the problems is listed in this article. The first five problems are cost-based specific. Problem number six and beyond involve both the rule and the cost based optimizers.

Problem 1: The Skewness

Imagine that a site with a table TRANS that has a column called STATUS. The column has two possible values: 'O' for Open Transactions that have not been posted, and 'C' for closed transactions that have already been posted, and which require no further action. There are in excess of 1 million rows that have a status of 'C', but only 100 rows that have a status of 'O' at any point in time.

The site has the following SQL statement that runs many hundreds of times daily. The response time is dismal, and we have been called in to "make it go faster."

```
SELECT acct_no, customer, product, trans_date, amt
FROM trans
WHERE status='O';
Response time = 16.308 seconds
```

In this example, taken from a real-life client engagement of mine, the cost-based optimizer decides that Oracle should perform a full-table scan. This is because the cost-based optimizer is aware of how many distinct values there are for the status column, but is unaware of how many rows exist for each of those values. Consequently, the optimizer will assume an even spread of data for each of the two values: 'O' and 'C'. Given this assumption, Oracle has decided to perform a full table scan to retrieve all open transactions.

If we inform Oracle of the data skewness by specifying the option FOR ALL INDEXED COLUMNS when we run the ANALYZE command, or when we invoke the DBMS_STATS package, Oracle will be made aware of the skewness of the data, that is, the number of rows that exist for each value for each indexed column. In our scenario, the status column is indexed. The following command is used to analyze the table:

ANALYZE TABLE TRANS COMPUTE STATISTICS FOR ALL INDEXED COLUMNS

After analyzing the table and computing statistics for all indexed columns, the cost-based optimizer is aware that there are only 100 or so rows with a status of 'O' and it will accordingly use the index on that column. Use of the index on the status column results in the following, much faster, query response:

`Response Time: 0.259 seconds`

Typically, the cost-based optimizer will perform a full table scan if the value selected for a column has over 12 percent of the rows in the table, and will use the index if the value specified has less than 12 percent of the rows. Although not exactly precise, as a rule of thumb, this is the typical behavior that the cost-based optimizer will follow.

Prior to Oracle9i, if a statement has been written to use bind variables, problems can still occur with respect to skewness even if you use FOR ALL INDEXED COLUMNS. Consider the following example:

```
local_status := 'O';
SELECT acct_no, customer, product, trans_date, amt
  FROM trans
 WHERE status= local_status;
# Response time = 16.608
```

Notice that the response time is similar to that experienced when the FOR ALL INDEXED columns option was not used. The problem here is that the cost-based optimizer isn't aware of the value of the bind variable when it generates an execution plan. As a general rule, to overcome the skewness problem, you should do the following:

- Hard code literals if possible. For example, use WHERE STATUS = 'O', not WHERE STATUS = local_status.
- Always analyze with the option FOR ALL INDEXED COLUMNS

If you are still experiencing performance problems with the cost-based optimizer not using an index due to bind variables being used, and you can't change the source code, try deleting the statistics off the index using a command such as the following:

```
ANALYZE INDEX TRANS_STATUS_NDX DELETE STATISTICS
```

Deleting the index statistics often works because it forces rule-based optimizer behavior, which will always use the existing indexes as illustrated in this next example.

Oracle9i will evaluate the bind variable value prior to deciding upon the execution plan, thus removing the need to hard code literal values.

Problem 2: Analyzing with Wrong Data

I have been invited to many, many sites that have performance problems, and at which I have quickly identified that the tables and indexes have not been analyzed while containing typical volumes of data. The cost-based optimizer requires accurate information, specifically data volumes, to have any chance of creating efficient execution plans.

The times when the statistics are most likely to be forgotten or be out of date are when a table is rebuilt or moved, an index is added, or a new environment is created. For example, a DBA might forget to regenerate statistics after migrating a database schema to a production environment. Other problems typically occur when the DBA does not have solid knowledge of the database that he/she is dealing with, and analyzes a table when it has zero rows, but will often have hundreds of thousand rows shortly afterwards.

How to check the last analyzed date

To observe which tables, indexes, and partitions have been analyzed and when they were last analyzed, you can select the LAST_ANALYZED column from the various user_XXX views, for example, USER_TABLES, and USER_INDEXES. To determine the last analyzed date for all your tables use the following:

```
SELECT table_name, num_rows,
       last_analyzed
FROM user_tables;
```

In addition to user_tables, there are many other views you can select from to view the date that an object was last analyzed. These views include user_indexes and many others. To obtain a full list of views with LAST_ANALYZED dates, you can run the following query.

```
SELECT table_name
FROM all_tab_columns
WHERE column_name = 'LAST_ANALYZED'
```

This is not to say that you should be analyzing with the COMPUTE option as often as possible. Analyzing frequently can cause a tuned SQL statement to become un-tuned.

When and how to analyze

A lot of sites analyze weekly using the compute option. These sites may analyze every Sunday with the analyze process taking more than nine hours. Users are keen to run reports and other jobs during that time, but they are told that they can't because the ANALYZE is running. Many DBAs believe they are doing a marvelous job if they do the Sunday analyze. In my eyes, this is bad practice and often leads to a catastrophe. The ANALYZE will usually improve performance, but can occasionally change its selection for the worse, posing a substantial risk. I've received a number of 8:30AM Monday morning phone calls asking me to come and help because a critical query that was running in one second is now taking three minutes.

Re-analyzing tables and indexes can be dangerous, in fact almost as dangerous adjusting your indexing. You should ideally be making a copy of the production database statistics prior to analyzing. This is possible with Oracle8.1.6 and later using the DBMS_STATS.EXPORT_SCHEMA_STATS procedure. If performance is severely degraded after the ANALYZE, you can quickly restore the pre-analyze statistics using the DBMS_STATS.IMPORT_SCHEMA_STATS procedure.

This is an example of how to export your statistics prior to re-analyzing. Note that you need a table in place to populate with the statistics. You create this using the DBMS_STATS.CREATE_STAT_TABLE procedure.

Step 1. Create the table to store the statistics:

```
EXECUTE SYS.DBMS_STATS.CREATE_STAT_TABLE
(OWNNAME=>'HROA', STATTAB=>'HROA_STAT_TABLE');
```

Step 2. Populate the table with the statistics to be copied:

```
EXECUTE SYS.DBMS_STATS.EXPORT_SCHEMA_STATS
(OWNNAME=>'HROA', STATTAB=>'HROA_STAT_TABLE', STATID=>
'HROA_21SEP_2001');
```

The DBMS_STATS package is only available with Oracle8.1.6 and later, so what do you do if the performance goes off the rails with an earlier version of the optimizer? Basically you need to weave some magic. If a particular index is suddenly not being used (which is the most common problem after a re-analyze) drop the statistics off the index and Oracle will usually use it. If this does not resolve the problem, try analyzing with a different percentage. Despite the resolution of the problem not being an exact science, I have been able to resolve the problems in 100 percent of the cases using these methods.

As mentioned above, some DBAs insist on analyzing with the compute option. I don't dispute that this provides Oracle with the best chance of making an informed decision. However, most sites can't afford to wait nine hours for the ANALYZE to complete. You have two options to overcome this problem: analyze with a lesser

percentage or analyze using DBMS_STATS.GATHER_SCHEMA_STATS with the GATHER STALE. You can also speed up the analyze process by analyzing in parallel.

Prior to Oracle8.1.6, ANALYZE with a percentage of about 5 percent for all tables with greater than or equal to 500,000 rows. I have not experienced cases where analyzing with a lower percentage has caused cost based optimizer decision problems, as long as the table is large. For all tables with less than 500,000 rows, you should typically analyze with COMPUTE.

The DBMS_STATS.GATHER_SCHEMA_STATS DEGREE value specifies the degree of parallelism to use on analyzing. In the following example, we have selected a degree of 4. If we are striped appropriately across multiple disks, this will work nicely for us and speed the process considerably. Unfortunately, only the table is analyzed with a degree of parallelism, not the indexes, but this still beats the ANALYZE command.

```
SYS.DBMS_STATS.GATHER_SCHEMA_STATS
(OWNNAME=>'HROA', ESTIMATE_PERCENT=>10,
DEGREE=>4, CASCADE=>TRUE);
```

DBMS_STATS has a GATHER STALE option that will only analyze tables that have had more than 10 percent of their rows changed. To us it, you first need to turn on monitoring for your selected tables. For example:

```
ALTER TABLE WINNERS MONITORING;
```

You can observe information about the number of table changes for a given table by selecting from the USER_TAB_MODIFICATIONS view. You can see if monitoring is turned on for a particular table by selecting the MONITORING column from USER_TABLES.

With monitoring enabled, you can run the GATHER_SCHEMA_STATS package using the GATHER STALE option:

```
EXECUTE SYS.DBMS_STATS.GATHER_SCHEMA_STATS
(OWNNAME=>'HROA', ESTIMATE_PERCENT=>10, DEGREE=>4, CASCADE=>TRUE, OPTIONS=>
'GATHER STALE');
```

Because GATHER_STALE is specified, tables will only be analyzed if they have had 10 percent or more of their rows changed since the previous analyze. This will also speed the ANALYZE process, and shorten the nine-hour time frame to get those end user reports done after all.

Getting the Data Volumes Right

PeopleSoft is one example of an application that uses temporary holding tables, where the table names typically end with _TMP. When batch processing commences, each holding table will usually have 0 rows. As each stage of the batch process completes, insertions and updates occur against the holding tables.

The final stages of the batch processing populate the major PeopleSoft transaction tables by extracting data from the holding tables. When a batch run completes, all rows are usually deleted from the holding tables. Transactions against the holding tables are not committed until the end of a batch run, when there are no rows left in the table.

When you run ANALYZE on the temporary holding tables, they will usually have zero rows. When the cost-based optimizer sees zero rows, it immediately considers full table scans and Cartesian joins. To overcome this, populate the holding tables and analyze them when they are fully populated with data. You can then truncate the tables and commence normal processing. When you truncate a table, the statistics are maintained.

You can find INSERT and UPDATE SQL statements to use in populating the holding tables by tracing the batch process that usually populates and updates the tables. You can use the same SQL to populate the tables.

The run times of the batch jobs at one large PeopleSoft site in Australia went from more than 36 hours down to less than 10 minutes using this approach. If you do not have the luxury of obtaining the SQL statements and

populating the tables and you are using Oracle8.1.6 and later, you can use the DBMS_STATS.SET_TABLE_STATS and DBMS_STATS.SET_COLUMN_STATS procedures. In the following example from an actual site, notice that as well as setting the number of rows and number of blocks in the table, I have also set the number of distinct values on the critical columns:

```
EXECUTE SYS.DBMS_STATS.SET_TABLE_STATS
(OWNNAME=>'F70PSOFT', TABNAME=>'PS_LED_AUTH_TBL',
NUMROWS=>1000000, NUMBLKS=>6000);
EXECUTE SYS.DBMS_STATS.SET_COLUMN_STATS
(OWNNAME=>'F70PSOFT', TABNAME=>
'PS_LED_AUTH_TBL', COLNAME=>
'OPRID', DISTCNT=>971);
```

If analyzing temporary holding tables with production data volumes does not alleviate performance problems, consider removing the statistics from those tables. This forces SQL statements against the tables to use rule based optimizer behavior. You can delete statistics using the ANALYZE TABLE tname DELETE STATISTICS command. If the statistics are removed, it is important that you do not allow the tables to be joined with tables having valid statistics. It is also important that indexes having statistics are not used to resolve any queries against the unanalyzed tables. If the temporary tables are used in isolation, and only joined with each other, the rule-based behavior is often preferable to that of the cost-based optimizer.

Problem 3: Mixing the Optimizers in Joins

As mentioned in the previous section, when tables are being joined, and one table in the join is analyzed and the other tables are not, the cost-based optimizer performs at its worst.

If you analyze your tables and indexes using the DBMS_STATS.GATHER_SCHEMA_STATS procedure, and the GATHER_TABLE_STATS procedures, which I strongly recommend, be careful to include the CASCADE=>TRUE option. By default, the DBMS_STATS package will gather statistics for tables only. Having statistics on the tables, and not on their indexes, can also cause the cost-based optimizer to make poor execution plan decisions.

One example of this problem that I experienced recently was at a site that had a TRANS table not analyzed, and an ACCT table analyzed. The DBA had re-built the TRANS table to purge data, and had simply forgotten to analyze afterwards. The following example shows the performance of a query joining the two tables:

```
SELECT a.account_name, sum(b.amount)
FROM trans b, acct a
WHERE b.trans_date > sysdate - 7
      AND a.acct_id = b.acct_ID
      AND a.acct_status = 'A'
GROUP BY account_name;
SORT GROUP BY
      NESTED LOOPS
      TABLE ACCESS BY ROWID ACCT
      INDEX UNIQUE SCAN ACCT_PK
      TABLE ACCESS FULL TRANS
Response Time = 410 seconds
```

After the TRANS table was analyzed using the following command, the response time for the query was reduced by a large margin:

```
ANALYZE TABLE trans ESTIMATE STATISTICS
      SAMPLE 5 PERCENT
      FOR ALL INDEXED COLUMNS
```

The new execution plan, and response time, were as follows:

```
SORT GROUP BY
      NESTED LOOPS
      TABLE ACCESS BY ROWID ACCT
```

```

INDEX UNIQUE SCAN ACCT_PK
TABLE ACCESS BY ROWID TRANS
INDEX RANGE SCAN TRANS_NDX1
Response Time = 3.1 seconds

```

One other site that I was asked to tune had been instructed by the vendor of their personnel package to ANALYZE only their indexes and not their tables. The software provider had originally developed the software for Microsoft's SQL Server database, and had ported it to Oracle. The results of just analyzing the indexes caused widespread devastation. For example:

```

SELECT count(*)
  FROM trans
 WHERE acct_id = 9
       AND cost_center = 'VIC';
TRANS_IDX2 is on ACCT_ID
TRANS_NDX3 is on COST_CENTER
Response Time 77.3 Seconds

```

Amusingly, the software developers were blaming Oracle, claiming that it had inferior performance to SQL Server. After analyzing the tables and indexes, the response time of this SQL statement was dramatically improved to 0.415 seconds. The response times for many other SQL statements were also dramatically improved.

The moral of this story could be to let Oracle tuning experts tune Oracle and have SQL Server experts stick to SQL Server. However, with ever more mobile and cross-database trained IT professionals, I would suggest that we all take more care in reading the manuals when we tackle the tuning of a new database.

Problem 4: Choosing an Inferior Index

The cost-based optimizer will sometimes choose an inferior index, despite the obvious fact that another index should be used. Consider the following PeopleSoft WHERE clause:

```

where business_unit      = :5
  and ledger             = :6
  and fiscal_year        = :7
  and accounting_period = :8
  and affiliate          = :9
  and statistics_code    = :10
  and project_id         = :11
  and account            = :12
  and currency_cd        = :13
  and deptid             = :14
  and product            = :15

```

The PeopleSoft system from which I took this example had an index which had every single column in the WHERE clause contained within it. One would have believed that Oracle would definitely use that index when executing the query. Instead, the cost-based optimizer decided to use an index on BUSINESS_UNIT, LEDGER, FISCAL_YEAR, ACCOUNT. When using the correct index shown above, the runtime was over 4 times faster than the runtime obtained when using the index chosen by the cost-based optimizer.

After further investigation, we observed that the index should have been created as a UNIQUE index, but had mistakenly been created as a NON-UNIQUE index as part of a data purge and table rebuild. When the index was rebuilt as a unique index, the index was selected by the cost-based optimizer. The users were obviously happy with their four-fold performance improvement.

However, more headaches were to come. The same index was the ideal candidate for the following statement, which was one of the statements run frequently during end of month and end of year processing:

```

where business_unit      = :5
  and ledger             = :6

```

```

and fiscal_year      = :7
and accounting_period between 1 and 12
and affiliate        = :9
and statistics_code  = :10
and project_id       = :11
and account          = :12
and currency_cd      = :13
and deptid           = :14
and product          = :15

```

Despite the index being correctly created as UNIQUE, the cost-based optimizer was once again ignoring it. The only difference between this statement and the last is that this statement is after a range of accounting periods for the financial year rather than just one accounting period.

This WHERE clause used the same incorrect index as previously mentioned, with the columns (BUSINESS_UNIT, LEDGER, FISCAL_YEAR and ACCOUNT). Once again, we timed the statement using the index selected by the cost-based optimizer against the index that contained all of the columns, and found the larger index to be at least 3 times faster.

The problem was overcome by re-positioning the ACCOUNTING_PERIOD column (originally 3rd in the index) to be last in the index. The new index order was as follows:

```

business_unit
ledger
fiscal_year
affiliate
statistics_code
project_id
account
currency_cd
deptid
Product
accounting_period

```

Another method that can be used to force the cost-based optimizer to use an index is to use one of the hints that allows you to specify an index. This is fine; but often sites are using third-party packages that can't be modified, and consequently hints can't be utilized. However, there may be the potential to create a view that contains a hint, with users then accessing the view. A view is useful if the SQL that is performing badly is from a report or online inquiry that is able to read from views.

As a last resort to force the use of an index, you can delete the statistics on the index. Occasionally you can also ANALYZE ESTIMATE with just the basic 1064 rows being analyzed. Often, the execution plan will change to just the way you want it, but this type of practice is approaching black magic. It is critical that if you adopt such a black magic approach, you clearly document what you have done to improve performance. The same method of analyzing must be used next time the table is rebuilt.

In summary, why does the cost-based optimizer make such poor decisions? First, I must point out that the poor decision making is the exception rather than the rule. The examples in this section indicate that columns are looked at individually rather than as a group. If they were looked at as a group, the cost-based optimizer would have realized in the first example that each row looked at was unique without the DBA having to re-build the index as unique. The second example illustrates that if several of the leading columns in an index have a low number of distinct values, and the SQL is requesting most of those values, the cost-based optimizer will often bypass the index. This happens despite the fact that collectively, the columns are very specific and will return very few rows.

In fairness to the optimizer, queries using indexes with fewer columns will often perform substantially faster than those using an index with many columns.

Problem 5: Incorrect INIT.ORA Parameter Settings

Many sites utilize a pre-production database to test SQL performance prior to moving index and code changes through to production. Ideally the pre-production database will have production volumes of data, and will have the tables analyzed in exactly the same way as the production database. The pre-production database will often be a copy of the actual production data files.

In such a situation, when DBAs test changes they work fine in pre-production, but have problems with a different execution plan being used in production. How can this be? The reason for a different execution plan in production is often different parameter settings in the production INIT.ORA file versus the pre-production INIT.ORA file.

I was at one site that ran the following update command and got a 4-minute response, despite the fact that the statement's WHERE clause condition referenced the table's primary key. Oddly, if we selected from the acct table rather than updating it with the same WHERE clause, the index was used.

```
UPDATE acct SET proc_flag = 'Y'
  WHERE pkey=100;
# Response Time took 4 minutes and
# wouldn't use the primary key
```

We tried re-analyzing the table every which way, and eventually removed the statistics. The statement performed well when the statistics were removed and the rule-based optimizer was used.

After much investigation, we decided to check the INIT.ORA parameters. We discovered that the COMPATIBLE parameter was set to 8.0.0 despite the database version being Oracle 8.1.7. We decided to set COMPATIBLE =8.1.7, and, to our delight, the UPDATE statement correctly used the index and ran in around 0.1 seconds.

COMPATIBLE is not the only parameter that needs to be set the same way in pre-production as in production to ensure that the cost-based optimizer makes consistent decisions. Other parameters include those listed in the following table.

Remember that if any of these parameters are different in your pre-production database as compared to your production database, there is the potential that the execution plans for your SQL statements will be different. Make the parameters identical to ensure consistent behavior.

Critical INIT.ORA Parameters

Parameter/Description

SORT_AREA_SIZE-

The number of bytes allocated on a per user session basis to sort data in memory. If the parameter is set at its default of 64K, NESTED LOOPS will be favored instead of SORT MERGES & HASH JOINS.

HASH_AREA_SIZE

The number of bytes to use on a per user basis to perform hash joins in memory. The default is twice the SORT_AREA_SIZE. Hash joins will often not work unless this parameter is set to at least 1 Meg.

HASH_JOIN_ENABLED

Enables or disables the usage of hash joins. It has a default of TRUE, and usually doesn't need to be set.

OPTIMIZER_MODE

CHOOSE, FIRST_ROWS, or ALL_ROWS. CHOOSE causes the cost-based optimizer to be used if statistics exist. FIRST_ROWS will operate the same way, but will tend to favor NESTED LOOPS instead of SORT MERGE or HASH JOINS. ALL_ROWS will favor SORT MERGES and HASH JOINS in preference to NESTED LOOP joins.

DB_FILE_MULTIBLOCK_READ COUNT

The number of blocks that Oracle will retrieve with each read of the table. If you specify a large value, such as 16 or 32, Oracle will, in many cases, bias towards FULL TABLE SCANS instead of NESTED LOOPS

OPTIMIZER_MODE_ENABLE

Enables new optimizer features to be enabled For example, setting the parameter to 8.1.7 will enable all of the optimizer features up to and including Oracle 8.1.7. This parameter can also automatically enable other parameters such as FAST_FULL_SCAN_ENABLED

Some of the major improvements that have occurred with the various Oracle versions include: 8.0.4 (ordered nested loops, fast full scans), 8.0.5 (many, many optimizer bug fixes), 8.1.6 (improved histograms, partitions, and nested loop processing), 8.1.7 (improved partition handling and subexpression optimization), 9.0.1 (much improved index joins, complex view merging, bitmap improvements, subquery improvements, push join predicate improvements)

OPTIMIZER_INDEX_CACHIN

Tells Oracle the percentage of index data that is expected to be found in memory. This parameter defaults to 0. The range of values is between 0 and 100. The higher the value, the more likely that NESTED LOOPS will be used in preference to SORT MERGE and HASH JOINS. Some sites have reported performance improvements on OLTP systems by setting this value to figures such as 80 or 90.

Parameter/Description**OPTIMIZER_INDEX_a NESTED LOOP_COST_ADJ -**

Affects the cost attributed to join. This parameter has a default of 100. If you lower the parameter to 10, you are telling the cost-based optimizer to lower the cost of a NESTED LOOP to 10/100 of its usual value. You can also set the value to something way beyond 100 to force a SORT MERGE or a HASH JOIN. This parameter is extremely powerful, and should be adjusted with caution. Adjusting it downwards may speed up some OLTP enquiries, but make overnight jobs run forever. And if you increase its value, the reverse may occur. I prefer to set the OPTIMIZER_INDEX_CACHING parameter in preference to this parameter.

STAR__ TRANSFORMATION._ENABLED -

Causes a star transformation to be used to combine bitmap indexes on fact table columns. This is different from the Cartesian join that usually occurs for star queries.

QUERY_REWRITE_ENABLED -

Allows the usage of function-based indexes as_ well as allowing query re-writes for materialized views. The default is FALSE, which may explain why your function indexes are not being used. Set it to TRUE.

PARTITION_VIEW_ENABLED -

Enables the use of partition views. If you are utilizing partitioned views, you will have to set this parameter to TRUE because the default is FALSE. A partition view is basically a view that has a UNION ALL join of tables. Partition views were the predecessor to Oracle partitions, and are used very successfully by many sites for archiving, and to speed performance.

PARALLEL_BROADCAST_ENABLED -

This parameter is used by parallel query_ when small lookup tables are involved. It has a default of FALSE. If set to TRUE, the rows of small tables are sent to each slave process to speed MERGE JOIN and HASH JOIN times when joining a small table to a larger table.

OPTIMIZER_MAX_PERMUTATIONS -

Can be used to reduce parse times. However, reducing the permutations can cause an inefficient execution plan, so this parameter should not be modified from its default setting.

CURSOR_SHARING -

If set to FORCE or SIMILAR, can result in faster parsing, reduced memory usage in the shared pool, and reduced latch contention. This is achieved by translating similar statements that contain literals in the WHERE clause to into statements that have bind variables.

The default is EXACT. We suggest that you consider setting this parameter to SIMILAR with Oracle9 only if you are certain that there are lots of similar statements where the only differences between them are values in the literals. It is far better to write your application to use bind variables if you can.

Setting the parameter to FORCE causes similar statements to use the same SQL area, which can degrade performance. FORCE should not be used.

Note that STAR TRANSFORMATION will NOT work if this parameter is set to SIMILAR or FORCE.

ALWAYS_SEMI_JOIN

My favorite parameter

This parameter can make a dramatic improvement to applications that make heavy usage of WHERE EXISTS. Setting the parameter to MERGE or HASH has produced runtime improvements from hours to minutes at several PeopleSoft sites that I have worked at recently. The default is STANDARD, which means that the main query (not the subquery) drives the execution plan. If you specifically set the parameter, the subquery becomes the driving query.

ALWAYS_ANTI_JOIN

This parameter will change the behavior of NOT IN statements and can speed processing considerably if set to HASH or MERGE. Setting the parameter causes a merge or hash join rather than the ugly and time consuming Cartesian join that will occur with standard NOT INs.

Problem 6: Statement not Written for Indexes

Some SELECT statement WHERE clauses do not use indexes at all. Most such problems are caused by having a function on an indexed column. Oracle8i and later versions allow function-based indexes, which may provide an alternative method of using an effective index which can be used to overcome the problem.

In the examples in this section, for each clause that cannot use an index, I have suggested an alternative approach that will allow you to get better performance out of your SQL statements.

In the following example, the SUBSTR function disables the index when it is used over an indexed column:

Do not use:

```
SELECT account_name, trans_date, amount
   _FROM   transaction
  _WHERE   SUBSTR(account_name,1,7) = 'CAPITAL';
```

Instead, use this code sample:

```
SELECT account_name, trans_date, amount
   FROM   transaction
  WHERE   account_name LIKE 'CAPITAL%';
```

In the following example, the "!=" (not equal) function cannot use an index. Remember that indexes can tell you what is in a table but not what is not in a table. All references to NOT, "!=", and "<>" disable index usage:

Do not use the following:

```
SELECT account_name,trans_date,amount   _FROM   transaction_WHERE   amount != 0;
```

Instead, use this code sample:

```
SELECT account_name,trans_date,amount   _FROM   transaction_WHERE   amount > 0 ;
```

In the following example, the TRUNC function disables the index:

Do not use the following:

```
SELECT account_name, trans_date, amount   _FROM
transaction_WHERE   TRUNC(trans_date) = TRUNC(SYSDATE);
```

Instead, use this code sample:

```
SELECT account_name, trans_date, amount _FROM
transaction_WHERE trans_date BETWEEN TRUNC(SYSDATE)
_ AND TRUNC(SYSDATE) + .99999;
```

In the following example, || is the concatenate function; it strings two character columns together. Like other functions, it disables indexes.

Do not use the following:

```
SELECT account_name, trans_date, amount _FROM
transaction_WHERE account_name || account_type
= 'AMEXA';
```

Instead, use this code sample:

```
SELECT account_name, trans_date, amount _FROM
transaction_WHERE account_name = 'AMEX' _AND
account_type = 'A' ;
```

In the following example, the addition operator is also a function and disables the index. All of the other arithmetic operators (-, *, and /) have the same effect.

Do not use the following command.

```
SELECT account_name, trans_date, amount _FROM
transaction_WHERE amount + 3000 <5000;
```

Instead, use this command.

```
SELECT account_name, trans_date, amount _FROM
transaction_WHERE amount < 2000;
```

In the following example, indexes will not be used when a column or columns appear on both sides of an operator. The result will be a full table scan. The following is an example of what not to do.

```
SELECT account_name, trans_date, amount _FROM
transaction_WHERE account_name = NVL(:acc_name, account_name);
```

Rather, you should follow this example:

```
SELECT account_name, trans_date, amount
_FROM transaction_
WHERE account_name LIKE NVL(:acc_name, '%');
```

As mentioned previously, function indexes can be used if the function in the WHERE clause represents the same function and column on which the function index was created. For function indexes to work, you must have the INIT.ORA parameter QUERY_REWRITE_ENABLED set to TRUE. You must also be using the cost-based analyzer. The statement in the following example uses a function index:

```
CREATE INDEX results_fn_ndx1 ON results(UPPER(owner))
SELECT count(*)
FROM results
WHERE UPPER(owner) = 'MR M A GURRY';
Execution Plan
-----
SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=32)
1 0 SORT (AGGREGATE)
```

```

2      1      INDEX (RANGE SCAN) OF
          'RESULTS_FN_NDX1' (NON-UNIQUE)
          (Cost=1 Card=1 Bytes=32)

```

Another common problem that causes indexes not to be used is when the column indexed is one datatype and the value in the WHERE clause is another data type. Oracle automatically performs a simple column type conversion, or casting, when it compares two columns of different types.) Assume that EMP_TYPE is an indexed VARCHAR2 column:

```

SELECT . . .
FROM emp
WHERE emp_type = 123
Execution Plan
-----
SELECT STATEMENT  OPTIMIZER HINT: CHOOSE
TABLE ACCESS (FULL) OF 'EMP'

```

Because EMP_TYPE is a VARCHAR2 value, and the constant 123 is a numeric constant, Oracle will cast the VARCHAR2 value to a NUMBER value. This statement will actually be processed as:

```

SELECT . . .
FROM emp
WHERE TO_NUMBER(emp_type) = 123

```

The EXPLAIN PLAN utility cannot detect or identify casting problems; it simply assumes that all module bind variables are of the correct type. Programs that are not performing up to expectation may have a casting problem. The EXPLAIN PLAN output will report the SQL statement as correct using the index, but that won't be the case.

A related performance problem that encountered a few weeks ago is relevant here. The code is from the popular Legal ERP package, Keystone. It was from a new release of the software in final testing prior to releasing to production. The index was on the columns (TCO_BILLING_STATUS, PCO_PRJ_OFF_OFFICE and TCO_PRJ_PROJECT_NUM). The WHERE clause had all three columns and used the correct index. The only problem was that the 2nd and 3rd columns (PCO_PRJ_OFF_OFFICE and TCO_PRJ_PROJECT_NUM) in the index had an incorrect datatype in the WHERE clause and the statement took an unacceptable 28 seconds to retrieve a single row of data. Putting the correct data type in the WHERE CLAUSE improved the response time by 28 times.

```

1  SELECT count(*) FROM TCO_COST_TRANSACTION
2     WHERE NVL(TCO_INFO_TRX, 'N') = 'N'
3         AND TCO_ITEM_TABLE IN ('IDO', 'IDT')
4         AND TCO_BILLING_STATUS = 'U'
5         AND TCO_ICC_ICC = '2PP'
6         AND TCO_PRJ_OFF_OFFICE = 03
7         AND TCO_PRJ_PROJECT_NUM = 50193363
8         AND TCO_IVC_DRAFT_KEY IN (0,5001374414)
9*   order by 1

```

```

SQL> /
COUNT(*)
-----

```

1

Elapsed: 00:00:29.08

```

SQL> SELECT count(*) FROM TCO_COST_TRANSACTION
2     WHERE NVL(TCO_INFO_TRX, 'N') = 'N'
3         AND TCO_ITEM_TABLE IN ('IDO', 'IDT')
4         AND TCO_BILLING_STATUS = 'U'
5         AND TCO_ICC_ICC = '2PP'
6         AND TCO_PRJ_OFF_OFFICE = '03'
7         AND TCO_PRJ_PROJECT_NUM = '50193363'

```

```

8      AND TCO_IVC_DRAFT_KEY IN (0,5001374414)
9      order by 1
10
SQL> set timing on
SQL> /
COUNT(*)
-----
1
Elapsed: 00:00:01.22

```

Problem 7: Indexes are Missing or Inappropriate

While it is important to use indexes to reduce response time, the use of indexes can often actually lengthen response times considerably. The problem occurs when more than 10% of a table's rows are accessed using an index.

I am astounded at how many tuners, albeit inexperienced, believe that if a SQL statement uses an index, it must be tuned. You should always ask the questions: Is it the best available index? Could an additional index be added to improve the responsiveness? Would a full table scan produce the result faster?

Another important aspect of indexes is that, depending on their type and composition, an index can affect the execution plan of a SQL statement. This must be considered when adding or modifying indexes.

Indexing Versus Full Table Scans

Using an index may cause more physical reads than performing a FULL TABLE SCAN. Each read from the index entry can have a corresponding physical read from disk for each row retrieved from the table. A full table scan is typically able to read over 100 rows of table information per block. Added to this, the DB_FILE_MULTIBLOCK_READ_COUNT parameter allows Oracle to read many blocks with one physical disk read. You may be reading 800 rows with each physical read from disk.

If an index lookup is retrieving more than 10% of the rows in a table, the full table scan is likely to be a lot faster than index lookups followed by the additional physical reads to the table to retrieve the required data. The exception to this rule is if the entire query can be satisfied by the index without the need to go to the table. In this case, an index lookup can be extremely effective. And if the SQL statement has an ORDER BY clause and the index is ordered in the same order as the columns in the ORDER BY clause, a sort can be avoided, which can further improve performance.

Adding Columns to Indexes

In the following example at a large stock brokerage company, the statement runtime was reduced from 40 seconds to 3 seconds for account_id 100101, which happened to be the largest account in the table. The response time was critical for answering online customer inquiries. The problem was solved by adding all of the columns in the WHERE clause, and those in the SELECT list, into the index. There is the tradeoff that this index now has to be maintained, but the benefits at this site far outweighed the costs.

```

SELECT SUM(val)_
FROM   account_tran_
WHERE  account_id   = 100101_
AND    fin_yr       = '1996'

```

The original index was on (account_id). The new index was on (account_id, fin_yr, val). The result was that the index entirely satisfied the query and the table did not need to be accessed.

Another common problem is that when tables are joined, the leading column of the index is not the column(s) with which the tables are joined. Consider the following example:

```

WHERE A.SURNAME = 'GURRY'
      AND A.ACCT_NO = T.ACCT_NO
      AND T.TRAN_DATE > '01-JUL-97'

```

```
AND T.TRAN_TYPE = 'SALARY'
```

In this situation, many sites will have an index on SURNAME for the ACCT table, and an index on TRAN_DATE and TRAN_TYPE for the TRANS table. To speed the query significantly, it is best to add the ACCT_NO join column as the leading column of the TRANS index. What you really want to have are indexes such as the following:

```
Index ACCT by (SURNAME)
Index TRANS by (ACCT_NO, TRAN_DATE, TRAN_TYPE)
```

Should I Index Small Tables?

Yet another common problem that I see at lots of sites is for small tables not to have any index at all. I often hear heated debates with one person saying that the index is not required because the table is small and the data will be stored in memory anyway. They will often explain that the table can even be created with the cache attribute.

My experience has been that every small table should be indexed. The two reasons for the index are that the uniqueness of the rows in the table can be enforced by a primary or unique key, and, more importantly, the optimizer has the opportunity to work out the optimal execution plan for queries against the table. The example below shows that the response time of a particular query went from 347 seconds elapsed down to 39.72 seconds elapsed when an index was created on the table. The most important thing about not having the index is that the optimizer will often create a less than optimal execution plan without it.

	Without Index		With Index	
	CPU	Elapsed	CPU	Elapsed
PARSE	0.00	0.03	0.01	0.11
EXECUTE	146.14	347.36	18.09	39.60
FETCH	0.00	0.00	0.00	0.00
TOTALS	146.14	347.39	18.10	39.72

Problem 8: Which is faster, IN or EXISTS?

The answer is that either can be faster depending upon the circumstance. If EXISTS is used, the execution path is driven by the tables in the outer select; if IN is used, the subquery is evaluated first, and then joined to each row returned by the outer query.

In the following example, notice that the HORSES table from the outer SELECT is processed first, and it drives the query:

```
SELECT h.horse_name
FROM horses h
WHERE horse_name like 'C%'
AND exists
(SELECT 'x'
FROM WINNERS w
WHERE w.position = 1
AND w.location = 'MOONEE VALLEY'
AND h.horse_name = w.horse_name)
```

Execution Plan

```
-----
0  SELECT STATEMENT Optimizer=CHOOSE
1   0  FILTER
1   1  INDEX (RANGE SCAN) OF
2     'HORSES_PK' (UNIQUE)
3   1  TABLE ACCESS (BY INDEX
4     ROWID) OF 'WINNERS'
5   3  INDEX (RANGE SCAN) OF
6     'WINNERS_NDX1' (NON-UNIQUE)]
```

The situation is reversed when IN is used. The following query produces identical results, but uses IN instead

of EXISTS. Notice that the table in the subquery is accessed first, and that drives the query:

```
SELECT h.horse_name
FROM horses h
WHERE horse_name like 'C%'
AND horse_name IN
(SELECT horse_name
FROM WINNERS w
WHERE w.position = 1
AND w.location = 'MOONEE VALLEY')
```

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE
1      0      NESTED LOOPS
2      1          VIEW OF 'VW_NSO_1'
3      2              SORT (UNIQUE)
4      3                  TABLE ACCESS (BY INDEX
5      4                      ROWID) OF 'WINNERS'
6      1                      INDEX (RANGE SCAN) OF
                          'WINNERS_NDX4' (NON-
                              UNIQUE)
1      1          INDEX (UNIQUE SCAN) OF
                          'HORSES_PK' (UNIQUE)
```

It is fair to say that in most cases, it is best to use the EXISTS rather than the IN. The exception is when a very small number of rows exist in the table in the subquery, and the table in the main query has a large number of rows that are required to be read satisfy the query.

The following example uses a temporary table that typically has only 2000 rows. The table is used in the subquery. The outer table has over 16,000,000 rows. In this example, the subquery is being joined to the main table using all of the primary key columns in the main table. This is an example of where the IN runs considerably faster than the EXISTS.

First the EXISTS-based solution is shown in the following sample.

```
DELETE FROM
  FROM ps_pf_ledger_f00
WHERE EXISTS
  (SELECT 'x'
  FROM ps_pf_led_pst2_t1 b
  WHERE b.business_unit = ps_pf_ledger_f00.business_unit
  AND b.fiscal_year = ps_pf_ledger_f00.fiscal_year
  AND b.accounting_period= ps_pf_ledger_f00.accounting_period
  AND b.pf_scenario_id = ps_pf_ledger_f00.pf_scenario_id
  AND b.source = ps_pf_ledger_f00.source
  AND b.account = ps_pf_ledger_f00.account
  AND b.deptid = ps_pf_ledger_f00.deptid
  AND b.cust_id = ps_pf_ledger_f00.cust_id
  AND b.product_id = ps_pf_ledger_f00.product_id
  AND b.channel_id = ps_pf_ledger_f00.channel_id
  AND b.obj_id = ps_pf_ledger_f00.obj_id
  AND b.currency_cd = ps_pf_ledger_f00.currency_cd);
Elapsed: 00:08:160.51
```

Notice the elapsed time. Next is the IN-based version of the same query. Notice the greatly reduced elapsed execution time:

```
ELETE FROM ps_pf_ledger_f00
WHERE( business_unit,fiscal_year,accounting_period,
      pf_scenario_id ,account,deptid ,cust_id ,
      product_id,channel_id,obj_id,currency_cd)
IN
```

```
(SELECT business_unit,fiscal_year,accounting_period,
        pf_scenario_id ,account,deptid ,cust_id ,
        product_id,channel_id,obj_id,currency_cd
FROM ps_pf_led_pst2_t1 );
Elapsed: 00:00:00.30
```

To help speed up EXISTS processing, you can often utilize the HASH_SJ and MERGE_SJ hints. If you can't add hints, setting the INIT.ORA parameter ALWAYS_SEMI_JOIN to HASH or MERGE will achieve the same execution plan as the hints. These hints allow Oracle to return the rows in the subquery only once. For example:

```
UPDATE PS_JRNL_LN
SET JRNL_LINE_STATUS = 'D'
WHERE BUSINESS_UNIT = 'A023'
  _AND PROCESS_INSTANCE=0001070341 _AND JOURNAL_DATE
IN ( TO_DATE('2001-08-01','YYYY-MM-DD'),_
TO_DATE('2001-08-14','YYYY-MM-DD'))
  _AND LEDGER IN ( 'ACTUALS')
AND JRNL_LINE_STATUS = '0' _AND EXISTS
_(SELECT /*+ HASH_SJ */ 'X' _FROM PS_COMBO_DATA_TBL
  _WHERE SETID='AMP'
  AND PROCESS_GROUP='SERVICE01' _
  AND COMBINATION IN ( 'SERVICE01',
    'SERVICE02', 'STAT_SERV1') _
    AND VALID_CODE='V' _ AND PS_JRNL_LN.JOURNAL_DATE BETWEEN
      EFFDT_FROM AND EFFDT_TO _
      AND PS_JRNL_LN.ACCOUNT =
        ACCOUNT _ AND PS_JRNL_LN.DEPTID = DEPTID)
UPDATE STATEMENT Optimizer=CHOOSE (Cost=9 Card=1 Bytes=80)
UPDATE OF PS_JRNL_LN
  HASH JOIN (SEMI) (Cost=9 Card=1
    Bytes=80)
  TABLE ACCESS (BY INDEX ROWID) OF
    PS_JRNL_LN (Cost=4 Card=1
    Bytes=33)
  INDEX (RANGE SCAN) OF PSDJRN_LN
    (NON-UNIQUE) (Cost=3 Card=1)
  INLIST ITERATOR
  TABLE ACCESS (BY INDEX ROWID) OF
    PS_COMBO_DATA_TBL (Cost=4
    Card=12 Bytes=564)
  INDEX (RANGE SCAN) OF
    PSACOMBO_DATA_TBL (NON-UNIQUE)
    (Cost=3 Card=12)
```

The PeopleSoft example shown was running for 2 hours without the HASH_SJ and reduced to an incredible 4 minutes with the hint. The hint forces the subquery SELECT rows to be read only once and then joined to the table outside the subquery (PS_JRNL_LN). The same effect can be obtained by setting the INIT.ORA parameter ALWAYS_SEMI_JOIN=HASH.

Problem 9: Unnecessary Sorts

Despite a multitude of improvements in the way in which Oracle handles sorts, including bypassing the buffer cache, having tablespaces especially set up as type "temporary," and more effective usage of memory, operations that include sorts can be expensive and should be avoided where practical.

The operations that require a sort include the following:

- CREATE INDEX
- DISTINCT
- GROUP BY
- ORDER BY

- INTERSECT
- MINUS
- UNIONS
- UNINDEXED TABLE JOINS

There are many things that a DBA can do to improve sorting, such as making sure that the sorting tablespace is a TEMPORARY tablespace, having a large SORT_AREA_SIZE to allow more sorts to occur in memory, ensuring that the default INITIAL and NEXT extents on the TEMP tablespace are a multiple of the SORT_AREA_SIZE parameter, and ensuring that all users are correctly assigned to the TEMP tablespace for their sorting. There are also things that you can also do in your SQL to avoid sorts.

Consider UNION ALL in place of UNION

Programmers of complex query statements that include a UNION clause should always ask whether a UNION ALL will suffice. The UNION clause forces all rows returned by the different queries in the UNION to be sorted and merged in order to filter out duplicates before the first row can be returned to the calling module. A UNION ALL simply returns all rows, including duplicates, and does not have to perform any sort, merge, or filtering operations.

Consider the following UNION query:

```
SELECT acct_num, balance_amt
FROM   debit_transactions
WHERE  tran_date = '31-DEC-95'
UNION
SELECT acct_num, balance_amt
FROM   credit_transactions
WHERE  tran_date = '31-DEC-95'
```

To improve performance, replace this code with the following UNION ALL query:

```
SELECT acct_num, balance_amt
FROM   debit_transactions
WHERE  tran_date = '31-DEC-95'
UNION ALL
SELECT acct_num, balance_amt
FROM   credit_transactions
WHERE  tran_date = '31-DEC-95'
```

Of course, if your program depends on duplicate rows being eliminated by the database, you have no choice but to use UNION.

Consider using an index to avoid a sort

Indexes can be used to avoid the need to perform sorts. Indexes are stored in ascending order by default. If the columns in your ORDER BY clause are in the same sequence as the columns in an index, forcing the statement to use that index will cause the data to be returned in the desired order.

To force the usage of the index, you can either add a hint or use a dummy WHERE clause. Consider the following statement that executes against a table having an index on (ACC_NAME, ACC_SURNAME):

```
SELECT acc_name,          acc_surname FROM account acct
ORDER BY acc_name
SELECT /*+ INDEX_ASC(acct acc_ndx1) */ acc_name,
      acc_surname
FROM account acct
```

A dummy WHERE clause, on the other hand, is often placed onto online enquiry screens. The following example uses WHERE acc_name > chr(1) in place of the ORDER BY clause. The WHERE clause forces the use of the index, which results in rows being returned in sorted order. One advantage of eliminating the sort in

an online application is that the first screen full of rows can be returned quickly.

```
SELECT acc_name, acc_surname
       FROM account
WHERE acc_name > chr(1)
```

In a statement like this, there is no need to have the ORDER BY clause. If the ORDER BY clause was specified, and the user put in a BLANK for their selection and pressed the GO button, every single row would need to be sorted before a single row could be returned. This could take a considerable amount of time, and is not desirable behavior in an OLTP environment.

Problem 10: Getting the Indexes on a Table Right

I've visited sites that have a standard in place saying that no table can have more than 6 indexes. This will often cause almost all SQL statements to run beautifully, but a handful of statements to run badly. And indexes can't be added because there are already 6 on the table.

Sometimes indexes may be redundant, such as an INDEX_1 on (A, B), INDEX_2 on (A, B, C) and INDEX_3 on (A, B, C, D). In such cases, DBAs often suggest dropping the first two indexes because they are redundant, i.e. they have the same leading columns, in the same order, as INDEX_3. Dropping redundant indexes, however, may often cause problems with the selection of a new driving table on a join using the rule-based optimizer and can occasionally (although less frequently) cause problems with the cost-based optimizer. There is far less risk associated with dropping redundant indexes when the cost-based optimizer is being utilized.

Having lots of indexes on a table will usually have only a small impact on OLTP systems, because only a few rows are processed in a single transaction, and the impact of updating many indexes is only milliseconds. Having lots of indexes can be extremely harmful for batch update processing with its typically high number of inserts, updates, and deletes. The following table demonstrates this.

Impact of Multiple Indexes on Insert Performance

Number of Inserts & Indexes	Run Time
Inserting 256 rows with 0 indexes	1.101 Seconds
Inserting 512 rows with 0 indexes	1.161 Seconds
Inserting 256 rows with 5 indexes	3.936 Seconds
Inserting 512 rows with 5 indexes	12.788 Seconds
Inserting 256 rows with 10 indexes	12.558 Seconds
Inserting 512 rows with 10 indexes	22.132 Seconds

Some sites overcome the problem of having many indexes on a table by dropping all indexes prior to batch updates, and re-creating them after the batch run is complete. Oracle has added lots of functionality to help speed index re-builds. For example, you can re-build indexes with the NOLOGGING or UNRECOVERABLE options, and you can re-build indexes in parallel. Despite these enhancements, tables may get to a size at which the index rebuild process takes longer than running a batch update with the indexes intact.

My recommendation is to avoid rules stating that a site will not have any more than a certain number of indexes. Oracle9i adds some great new functionality that allows you to identify indexes that are not being used. The command is ALTER INDEX MONITORING USAGE. Take advantage of this command to identify and remove unused indexes.

The bottom line is that all SQL statements must run acceptably. There is always a way of achieving this. If it requires having 10 indexes on a table, then you should put 10 indexes on the table.

Problem 11: Tables with Many Deletes

Oracle is similar to many other databases in that there are performance issues with deletes. Oracle has a high water mark, which represents the highest number of rows ever inserted into the table. This high-water mark can have an impact on performance. Consider the following example, which takes 5.378 seconds to read a table with 151,070 rows:

```
SELECT COUNT(*) FROM YYYY;
151070
real: 5378
```

It just happens that all 150,000 rows have the STATE column set to 'VIC', and that the table has an index on the STATE column. If I use a WHERE clause to force the count to use the index on the STATE, it takes 16.884 seconds:

```
SELECT COUNT(*) FROM YYYY WHERE STATE='VIC';_-----_ 151070
real: 16884
```

Notice that the index scan took about 3 times longer than the full table scan. By the way, this SELECT statement was done using the rule-based optimizer. The cost-based optimizer would have performed a FULL TABLE SCAN.

Now let's delete all of the rows, so that the result is an empty table:

```
DELETE FROM YYYY;
real: 55277
```

Now that we have an empty table, lets count all the rows again:

```
SELECT COUNT(*) FROM YYYY;_-----_ 0
real: 5117
```

Notice that it takes the same amount of time to count zero rows as it took to count from the table when it was completely populated. This is because, when performing a full table scan, Oracle reads as far as the table's high water mark, and the high-water mark has not changed.

Let's count the rows again using the index:

```
SELECT COUNT(*) FROM YYYY WHERE STATE='VIC';
0
real: 16029
```

Just as before, it takes the same amount of time to count zero rows as it took to count the original 150,000. This is because the index entries are logically deleted, but still exist physically.

The table has never had any rows with a state equal to 'NSW' (New South Wales), so let's try counting the 'NSW' rows:

```
SELECT COUNT(*) FROM YYYY WHERE STATE='NSW';
real: 16940
```

The count still takes the same amount time as before, when counting the 'VIC' rows. This is because scanning the index requires Oracle to scan past the logically deleted Victorian ('VIC') entries.

To avoid the types of performance problems I've just demonstrated, rebuild a table and its indexes whenever the table has undergone many deletes. If index columns are frequently updated, you should also re-build the indexes because an update forces a logical delete in the index followed by an insert of the new, updated entry. Some sites go as far as re-building indexes nightly when they have a lot of logical delete activity. To detect which tables have many deletes and updates, run the following SELECT statement:

```
SELECT sql_text, executions
FROM v$sqlarea
WHERE UPPER(sql_text) LIKE 'DELETE%'
OR
UPPER(sql_text) LIKE 'UPDATE%';
```

The output from this statement will contain SQL statements about tables that have many deletes and updates. You should consider regular rebuilds of indexes on these tables.

About the Author

Mark Gurry runs a Company called Mark Gurry and Associates (MGA), which is based in Melbourne and Sydney in Australia as well as Seattle and Newport Beach in the United States. His company is keenly sought for performance tuning work, particularly with sites using PeopleSoft and other ERP solutions. Mark is best known as the co-author of the original best selling Oracle book "Oracle Performance Tuning". His new book "Oracle SQL Tuning" was recently published. Many of the examples in this article are from the book. Mark can be contacted on mgurry@mga.au.com or through his Web site at www.mga.au.com.

[Download Acrobat Reader](#)

Copyright 2003 by the International Oracle Users Group