Managing Statistics for Optimal Query Performance

Karen Morton
Method R Corporation, Beaverton, Oregon, USA
karen.morton@method-r.com

Revised 2009/02/09

Half the battle of writing good SQL is in understanding how the Oracle query optimizer analyzes your code and applies statistics in order to derive the "best" execution plan. The other half of the battle is successfully applying that knowledge to the databases that you manage. The optimizer uses statistics as input to develop query execution plans, and so these statistics are the foundation of good plans. If the statistics supplied aren't representative of your actual data, you can expect bad plans. However, if the statistics are representative of your data, then the optimizer will probably choose an optimal plan.

1 OPTIMIZER BASICS

The Oracle query optimizer is the code path within the Oracle kernel that is responsible for determining the optimal execution plan for SQL. The optimizer must review your query text and iterate over multiple operation choices that would provide the required result set. Each scenario generates a cost value that the optimizer then uses to compare to other scenario costs. The cost is an estimated value that is intended to be proportional to the expected resource use needed to execute the statement with a particular plan. The optimizer calculates a cost of access paths and join orders based on an estimate of the needed computer resources, which includes I/O, CPU, and memory.[1] The final execution plan is the one with the lowest estimated cost.

1.1 The Importance of Statistics

There are many informational inputs into the optimizer's calculations, including: object statistics, system statistics, parameter settings, and physical design choices (e.g., partitioning, clustering, and indexing). But statistics are the primary information source for the optimizer. It is through the use of statistics that the optimizer attempts to determine the most efficient way to use resources to satisfy your query. The more accurately your statistics reflect the true nature of your data, the better the optimizer's plan choice will be, and thus the better your query performance will be.

1.2 Basic Calculations

The importance of statistics is evident in the basic calculations used by the optimizer. Let's take a simple SQL statement and break it down as the optimizer would to review the part statistics play in the derivation of the execution plan.

```
SELECT *
  FROM ord
 WHERE order_no = 256769;
```

The statistics for the `ord` table are:

[1] Oracle Database Performance Tuning Guide 11g Release 1, p. 11-7

© 2009 Method R Corporation. All rights reserved.

```
Statistic       Current value
--------------- ----------------------
# rows          12890
Blocks          180
Empty blocks    0
Avg space       0
Chain Ct        0
Avg Row Len     79
Degree          1
Sample Size     12890
Partitioned     NO
Temporary       N
Row movement    DISABLED
Monitoring      YES
```

The pertinent column statistics for `order_no` column are:

```
# distinct values:  12890
Sample Size      :  12890
Nulls            :  N
# Nulls          :  0
Density          :  .000078
Avg Length       :  5
Histogram Info   :  NONE
Low Value        :  66
High Value       :  999969
```

Given this information, the optimizer will first determine the selectivity of the query predicates. The selectivity is the fraction of rows estimated to match the predicate. A predicate acts as a filter that filters a certain number of rows from a row set. In our example, the selectivity will be computed for condition `order_no = 256769`. Selectivity is calculated for an equality as `1/NDV` (# distinct values) for the column(s) used in the `WHERE` clause. By default, for columns that do not have histograms (statistics that relate to data skew), the statistic called density contains the answer for the selectivity calculation for an equality condition. In our simple example, the density is .000078 or 1/12890.

With this value, the optimizer can now estimate *result set cardinality*. Result set cardinality is the expected number of rows the query should return. It is computed by multiplying the selectivity by the number of rows in the table.

```
cardinality = selectivity × # rows in table
```

For our example this would be: .000078 * 12890 = 1.00542.

The optimizer now must decide how best to retrieve the row(s) it has estimated will be returned. It must check for available indexes, look at parameter settings and consider system statistics (if you're using them) and do some other work to finalize the plan choice.

There are two plan choices the optimizer considers for this query (based on output from an event 10053 optimizer trace collection):

A full table scan…

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    79 |    42   (3)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORD  |     1 |    79 |    42   (3)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ORDER_NO"=256769)
```


© 2009 Method R Corporation. All rights reserved.

And a table access by rowid using an index…

```
-------------------------------------------------------------------------------------
| Id  | Operation                   | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |     1 |    79 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| ORD     |     1 |    79 |     2   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | ORD _PK |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ORDER_NO"=256769)
```

Note how the optimizer's estimated cost for the index access plan (2) is considerably lower than the estimated cost for the full table scan plan (42). You can see how it makes sense that the estimated cost is less for an index scan, as using an index to access the estimated one row will certainly take a lot fewer block accesses than a full table scan would. Given that the index plan has the lower cost, we can conclude that the optimizer would choose that execution plan for the query.

In his book, *Cost-Based Oracle Fundamentals* (Apress, 2006), Jonathan Lewis provides an in-depth view of how the "optimizer tries to efficiently convert your query into an execution plan." For a much more detailed look at how the optimizer computes cost and determines query execution plans than I will cover here, I highly recommend this book. The take-away from this brief example is that the optimizer relies heavily on statistics in choosing an execution plan. If the optimizer considers statistics important, then so should you.

1.3 Execution Plans

Now, let's take a brief look at a bit more detail about executions plans. As I've already stated, the execution plan for a query is the resulting set of operations to be completed based on the optimizer's computations for the different scenarios possible that would derive the result set for the query. Being able to review the actual execution plans for your queries is an invaluable tool for determining how your query is performing and for assisting in diagnosing performance problems. However, many people confuse the output of `EXPLAIN PLAN` with an execution plan. Using `EXPLAIN PLAN` only provides you with the plan that "should be" executed. It is an estimate. It's typically a very good estimate, but the only way to get the exact execution plan for a query is to execute the query and capture the actual plan data. Think of `EXPLAIN PLAN` as being similar to the difference between telling somehow how you'd throw a baseball and actually doing it. You may talk a good game, but can you actually do exactly what you say?

Actual execution plans and their statistics can be captured in several ways. Two of the most common are (1) using extended SQL trace and (2) retrieving the last execution data for a query from `V$SQL_PLAN` or its related views. In Oracle version 10 and above, Oracle provides an easy to use packaged function, `dbms_xplan.display_cursor`, to return a nicely formatted plan that includes execution statistics.

I executed the following example in SQL*Plus, using the simple query shown earlier:

```
SQL>set serveroutput off
SQL>select /*+ gather_plan_statistics */ * from ord where order_no = 256769 ;

  ORDER_NO    CUST_NO ORDER_DAT TOTAL_ORDER_PRICE DELIVER_D DELIVER PA     EMP_NO DELIVER_NAME  ...
---------- ---------- --------- ----------------- --------- ------- -- ---------- ------------- ...
    256769      19314 28-APR-03          42232.66 28-OCT-04 By Noon CC       9875 XFPFMFCXR     ...
SQL>
SQL>select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------------------
SQL_ID  34rzzvhjq0r8w, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ * from ord where order_no = 256769

Plan hash value: 3958543594
-----------------------------------------------------------------------------------------------
| Id  | Operation                   | Name    | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
```

© 2009 Method R Corporation. All rights reserved.

```
|   1 |  TABLE ACCESS BY INDEX ROWID| ORD     |      1 |      1 |      1 |00:00:00.01 |       3 |
|*  2 |   INDEX UNIQUE SCAN         | ORD _PK |      1 |      1 |      1 |00:00:00.01 |       2 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ORDER_NO"=256769)
```

Let me point out a couple of things about this example. In order to get the actual rowsource execution statistics (that's the data in the `A-Rows`, `A-Time`, and `Buffers` columns), you must either use the `gather_plan_statistics` hint in your query or you must set the `statistics_level` parameter to `ALL`. The use of the hint is less inclusive as it only collects rowsource execution statistics for the one query you're testing. If you change the parameter, it collects this information for all queries executed at either the session or system level, depending on how you issue the parameter change, until it is reset (the default value is `TYPICAL`). By setting the parameter to `ALL`, you may incur a greater overall performance penalty than if you were simply collecting the information on a query-by-query basis.

Also notice that this example shows the use of the command `set serveroutput off`. This is something that needs to be done in SQL*Plus to ensure that the execution plan that is displayed is the last query execution plan and not the plan for the execution of the `dbms_xplan.display_cursor` function. Note what happens if `serveroutput` is on:

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
SQL_ID  9babjv8yq8ru3, child number 0

BEGIN DBMS_OUTPUT.GET_LINES(:LINES, :NUMLINES); END;

NOTE: cannot fetch plan for SQL_ID: 9babjv8yq8ru3, CHILD_NUMBER: 0
      Please verify value of SQL_ID and CHILD_NUMBER;
      It could also be that the plan is no longer in cursor cache (check v$sql_plan)
```

Since the `dbms_xplan.display_cursor` function displays the plan data for the last executed statement, you actually see the statement call to `DBMS_OUTPUT` being the last statement when `serveroutput` is on. So, just remember that for SQL*Plus, you should set `serveroutput` off first.

Once you have the execution plan data, you've got a wealth of information to help you determine the true performance of a query. The key data in this display are revealed in the columns `Starts`, `E-Rows`, `A-Rows`, `A-Time`, and `Buffers`.

`Starts`
: The number of times the particular step is executed. Most of the times it is 1, but in the case of a `NESTED LOOPS` rowsource operation, you will likely see a higher number.

`E-Rows`
: The number of rows that the Oracle query optimizer *estimated* would be returned by each rowsource operation in the plan. This is the cardinality calculation that I demonstrated earlier.

`A-Rows`
: The actual number of rows that *were actually* returned by each rowsource operation in the plan.

`A-Time`
: The actual elapsed duration for each plan operation. Note that parent steps include the durations of their child steps.

`Buffers`
: The actual count of access operations upon buffers in the buffer cache (also known as logical I/O operations or LIOs). Note that parent steps include the buffer access counts of their child steps.

The absolutely great thing about this information is that you get to see exactly what your query did. It's not a guess. It's the real execution information for your query. This is invaluable because it shows you specific information you can use when your query is misbehaving. One common problem that can negatively affect performance is when data is skewed—for example, when there's one value in a column that's used in 99% of the rows in a table. As we'll discuss

© 2009 Method R Corporation. All rights reserved.

further in the section ahead, you can compare the E-Rows column value and the A-Rows column value for each plan step to see exactly where you have performance problems induced by skewed data. What if in our example, the E-Rows column had shown 1, but the A-Rows column showed 4,712? The big difference in the optimizer's estimated and actual cardinality values is a vivid indicator of when you have skew in your data.

2 WHAT'S THE REAL QUESTION?

With this foundation information in place, let's get down to business. Almost everywhere I go to consult or teach, people ask me, "What is the best method for collecting statistics?" The answer is: it depends. If there were a single answer that worked for everyone Oracle would've surely implemented it by now. And, in some ways, you can say that Oracle has done just that. In Oracle version 10, statistics collection was automated in that object usage is monitored by default and Oracle schedules a statistics collection whenever data changes by a minimum of 10%. Of course, you can disable this scheduled automated collection process, and many people choose to do just that. But Oracle Corporation's intent was to relieve the DBA of the responsibility for most collections and put it back on the database. Oracle has done a good job of automating the collection of statistics that are consistently representative of the actual data in the database.

Getting better query performance isn't typically a matter of tweaking parameters. The Oracle Optimizer Team says that having good statistics will typically provide performance benefits that are orders of magnitude above any benefit derived from parameter tuning. To that end, they have tried to make the statistics collection process provide a good general solution for most needs.[2] The queries that fall outside this baseline solution will require special attention to refine statistics and query text to allow the optimizer to derive an optimal plan.

Remember this:

> Statistics that don't reasonably describe your actual data (e.g., because they're stale or missing)
>
> …lead to poor cardinality estimates,
>
> …which leads to poor access path selection,
>
> …which leads to poor join method selection,
>
> …which leads to poor join order selection,
>
> …which leads to poor SQL execution times.

Statistics matter! So, what are your options and how do you decide what is best for your environment?

3 COLLECTING STATISTICS

Object statistics can be collected on tables, columns, and indexes. There are several views (I list the most common here) that expose statistics data (these views also have *all_* and *user_* prefixed versions).

dba_tables	dba_tab_cols
dba_tab_statistics	dba_tab_columns
dba_tab_histograms	dba_tab_col_statistics
dba_tab_partitions	dba_tab_subpartitions
dba_indexes	dba_ind_columns
dba_part_col_statistics	dba_part_col_statistics
dba_ind_partitions	dba_ind_subpartitions

[2] *A Practical Approach to Optimizer Statistics in Oracle Database 10g*. Oracle OpenWorld 2005.

© 2009 Method R Corporation. All rights reserved.

Statistics are maintained automatically by Oracle or you can maintain the optimizer statistics manually using the DBMS_STATS package. Oracle recommends for you to collect statistics automatically, but specific application requirements can dictate more complicated collection strategies that you have to execute manually. The problem is that both approaches are generalized, yet your application's needs may be specific.

3.1 Automatic Statistics Collection

Automatic optimizer statistics collection gathers optimizer statistics by calling the *dbms_stats.gather_database_stats_job_proc* procedure. This procedure collects statistics on database objects when the object has no previously gathered statistics or the existing statistics are stale because the underlying object has been modified significantly (more than 10% of the rows). *Dbms_stats.gather_database_stats_job_proc* is an internal procedure, but it operates in a very similar fashion to the *dbms_stats.gather_database_stats* procedure using the GATHER AUTO option. The primary difference is that the automatically executed procedure prioritizes the database objects that require statistics, so that those objects that most need updated statistics are processed first. This ensures that the most-needed statistics are gathered before the maintenance window closes.[3]

It is important to note that the automated procedure for collecting statistics is most functional when data changes at a slow to moderate rate. If your data changes rapidly, causing your statistics to become stale and affect plan choice between maintenance windows, then you will need to consider manual collection alternatives. Oracle suggests that volatile tables and those tables that have large bulk loads are likely candidates for manual collections.

Another important thing to remember is that just because it's automatic doesn't mean it will most accurately handle the needs and nuances of your applications. In my experience, I've found that the automated collection options are often just as inadequate due to particular query constructs and application design issues as they are for volatility and load characteristics. Consider these questions:

- Is it common for your users to get slammed with performance problems shortly after you update your statistics?
- Does performance decline before a 10% data change (enough to trigger an automatic collection) occurs?
- Do low and high values for a column change significantly between automatic collections?
- Does your application performance seem "sensitive" to changing user counts as well as data volume changes?

These are just a few of the questions you need to ask yourself to help determine if automatic collections can be used successfully in your environment. If you answer yes to one or more of these questions, then automatic statistics collection may not serve you as well as needed.

3.1.1 How Automatic Collections Work

As previously stated, the automated statistics collection job will be triggered if data changes by 10% of the rows or more, based on table monitoring data. Once triggered, the job will run during the next normal maintenance window (there is one maintenance window each day). An equivalent collection, if done manually, would be executed from SQL*Plus as follows:

```
SQL> exec DBMS_STATS.GATHER_TABLE_STATS ( ownname=>?, tabname=>?);
```

In other words, other than specifying the owner and table name, all the other parameters would use default values. Those defaults are:

```
partname         NULL                                                    cascade       DBMS_STATS.AUTO_CASCADE
estimate_percent DBMS_STATS.AUTO_SAMPLE_SIZE                             stattab       NULL
block_sample     FALSE                                                   statid        NULL
method_opt       FOR ALL COLUMNS SIZE AUTO                               statown       NULL
degree           1 or value based on number of CPUs and initialization parameters  force         FALSE
granularity      AUTO (value is based on partitioning type)              no_invalidate DBMS_STATS.AUTO_INVALIDATE
```

What that means to you is that Oracle will…

- Select what it thinks is a statistically "good enough" `estimate_percent` value (somewhere between .000001 and 100),

[3] Oracle Database Performance Tuning Guide 11g Release 1, p. 13-2.

© 2009 Method R Corporation. All rights reserved.

- Collect column statistics for all columns and determine if a histogram is warranted based on (1) the distribution of values within the column and (2) the workload (as determined by checking SYS.COL_USAGE$),
- Determine whether index statistics need to be collected or not (this is usually TRUE), and
- Determine whether or not to invalidate dependent cursors (this is usually TRUE).

Sounds pretty good, doesn't it? Or, does it? I think the important thing to keep in mind is the concept of statistically "good enough." You may have heard it said that if a man stands with one foot in a bucket of ice water and the other foot in boiling water, on average he'll feel comfortable.[4] That's a ridiculous conclusion, you might say, and that's exactly my point. When is statistically "good enough" *not* good enough for you? Let's take a look at a great example I recreated from a blog posting by Richard Foote.[5] It shows how one parameter's default value, METHOD_OPT=>FOR ALL COLUMNS SIZE AUTO, can cause problems.

3.1.1.1 Scenario

Imagine a test table with one million rows. It has three columns: a, b, and c. Column a, the primary key, is populated by rownum, and is initially uniformly distributed. To "ruin" the perfect uniform distribution of the first column, let's create a single outlier value in the table: we insert one row with a value of 1,000,000,000. Column b is uniformly distributed with values between 1 and 10; each number appears in the table exactly as many times as each other number. Technically, we can say that column c is also uniformly distributed, but is "special" in that it has only one distinct value.

```
SQL>desc auto_test
 Name                      Null?    Type
 ------------------------- -------- -------
 A                         NOT NULL NUMBER
 B                                  NUMBER
 C                                  NUMBER

Statistic       Current value
--------------- ------------------------------
# rows          1000000
Blocks          2260
Empty blocks    0
Avg space       0
Chain Ct        0
Avg Row Len     10
Degree          1
Sample Size     1000000
Last analyzed   2009-01-27 16:40:59
Partitioned     NO
IOT type
Temporary       N
Row movement    DISABLED
Monitoring      YES

---------- Column Statistics ----------

Column       NDV  Sample Size Nulls  # Nulls   Density Avg Len Histogram Low Value  High Value
------- -------- ------------- ----- -------- --------- ------- --------- ---------- -----------
A        1000000       1000000 N            0   .000001       5 NONE (1)  1          1000000000
B             10       1000000 Y            0   .100000       3 NONE (1)  1          10
C              1       1000000 Y            0  1.000000       3 NONE (1)  100        100

---------- Index Statistics ----------

                                      Leaf Distinct     Num  Clust  Avg Leaf Avg Data
Index Name   Col# Column  Unique? Height Blks Keys         Rows Factor Blks/Key Blks/Key
------------ ----- ------- ------- ------ ---- -------- -------- ------ -------- --------
SYS_C007131     1 A       Y            3 2087  1000000  1000000   2202        1        1
```

[4] Bobby Bragan. http://www.bobbybragan.com.

[5] The full test script and execution results can be found on Richard's blog at http://richardfoote.wordpress.com/2008/01/04/dbms_stats-method_opt-default-behaviour-changed-in-10g-be-careful/.

© 2009 Method R Corporation. All rights reserved.

The test will conduct three different statistics collections and execute a query to review the optimizer's plan choice:

1. 100% collection with no histograms. This test shows the behavior of METHOD_OPT set to FOR ALL COLUMNS SIZE 1 (this was the Oracle 9i default).
2. 100% collection using the SIZE AUTO histogram option. This test shows the behavior of METHOD_OPT set to FOR ALL COLUMNS SIZE AUTO (this is the Oracle 10g and above default).
3. 100% collection collecting a histogram only on column a. This test shows the behavior of METHOD_OPT set to FOR COLUMNS a SIZE 254 (this is a manual collection on a specific column only).

The test query has a single condition searching for rows where the value of column a is greater than 1,000,000:

```
select /*+ gather_plan_statistics */ * from auto_test where a > 1000000;
```

3.1.1.2 Expected results

We know that column a is perfectly distributed except for the one row we changed to have the large outlier value. Therefore, the query will return only 1 row. The best performance option to get one row out of one million, *should* be to use the primary key index on column a to retrieve the single row.

3.1.1.3 Actual results

```
Test 1 (100%, FOR ALL COLUMNS SIZE 1)
SQL>select /*+ gather_plan_statistics */ * from auto_test where a > 1000000;

             A               B               C
-------------- --------------- ---------------
    1000000000               1             100

1 row selected.

SQL>select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
SQL_ID  2614vs0sn81cp, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ * from auto_test where a > 1000000

Plan hash value: 3800464057
-------------------------------------------------------------------------------------------
| Id  | Operation         | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|*  1 |  TABLE ACCESS FULL| AUTO_TEST |      1 |    999K|      1 |00:00:00.05 |    2237 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("A">1000000)
```

Notice the expected rows returned is 999K, not 1 and that the Oracle kernel executed a full table scan. Why? Because the optimizer thought that since the high value for the column is 1,000,000,000 and the low value is 1, when you asked for rows with a > 1000000, it computed the cardinality like this:

```
cardinality = (High value  -  predicate value) / (High value  - Low value) x # rows in table
```

Using this formula, that's a selectivity of .999. Multiply the selectivity by the number of rows in the table (1 million) and there you have your 999K estimate. But *you* know there is only one row that satisfies our query. The problem is our outlier value. The optimizer doesn't understand there is a single value. The optimizer will estimate that the query will return 999K rows even though our actual data doesn't allow that possibility. Shouldn't the optimizer know better? Let's see if using SIZE AUTO will help.

```
Test 2 (100%, FOR ALL COLUMNS SIZE AUTO)
SQL>select /*+ gather_plan_statistics */ * from auto_test where a > 1000000;
```


© 2009 Method R Corporation. All rights reserved.

```
             A               B               C
-------------- --------------- ---------------
    1000000000               1             100

1 row selected.

SQL>select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
SQL_ID  2614vs0sn81cp, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ * from auto_test where a > 1000000

Plan hash value: 3800464057
-----------------------------------------------------------------------------------------------
| Id  | Operation         | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------
|*  1 |  TABLE ACCESS FULL| AUTO_TEST |      1 |    999K|      1 |00:00:00.05 |    2237 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("A">1000000)
```

There is no change. With the `SIZE AUTO` option in effect, the collection actually created histograms on both the `b` and `c` columns but didn't create one for the `a` column (where it likely could've been of some use). Given that we know how the three columns were populated, Oracle has basically fouled up all three columns statistics with the `AUTO METHOD_OPT` option. Now, let's just collect statistics manually to reflect our knowledge that the `a` column statistics need to be reflective of the outlier value and execute our test again.

```
Test 3 (100%, FOR COLUMNS A SIZE 254)
SQL>select /*+ gather_plan_statistics */ * from auto_test where a > 1000000;

             A               B               C
-------------- --------------- ---------------
    1000000000               1             100

1 row selected.

SQL>select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------------------------------
SQL_ID  2614vs0sn81cp, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ * from auto_test where a > 1000000

Plan hash value: 1106764748
-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name        | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID| AUTO_TEST   |      1 |   3937 |      1 |00:00:00.01 |       4 |      2 |
|*  2 |   INDEX RANGE SCAN          | SYS_C007131 |      1 |   3937 |      1 |00:00:00.01 |       3 |      2 |
-----------------------------------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("A">1000000)
```

Although the estimated rows count is not perfect, at least it's now *good enough* to generate a more appropriate execution plan.

The conclusion you can derive from this test is that you need to *know your data*! It is possible that the default behavior for `METHOD_OPT` can fail to gather the correct statistics you need.

© 2009 Method R Corporation. All rights reserved.

3.2 Manual Statistics Collection

Manual collections of statistics were the norm until Oracle 10g's automations came onto the scene. You developed your own scripts and set up regularly scheduled jobs to collect updated statistics. So, the idea of letting Oracle take over and stop using your own, *proven* methods has been a hard change to accept and implement. You may have actually made an attempt, only to have problems arise caused by the new collection defaults, and so you've returned to your old methods.

If automated collections lead the optimizer to make plan choices that give poor performance in your environment, it is perfectly reasonable to go back to the "tried and true" methods. The question you must ultimately answer is: what is the best way to collect statistics in my environment? Remember that the automated collections use default values that *should be* sufficient for *most* environments *most* of the time. It all comes down to whether most of the time is good enough for you.

3.2.1 Deciding Manual Collection Parameters

How do you decide what values to use for the various collection parameters if you find that automated collections don't work for you? I hope your answer is: test, test, test. Unfortunately, the response I get when I ask how a particular strategy was selected is, "Uh…we tried it and it just worked, so we stuck with it." OK. I suppose that if the results are what you want, then perhaps a "good guess" strategy worked (in this case). But, what if you want to make an educated guess versus a shot-in-the-dark guess?

An educated guess requires you to obtain reliable answers to a few key questions:

- What are the most common query predicates for key tables?
- Is it possible to have popular values for one or more columns in key tables?
- Is it possible for data values to be non-uniformly distributed (remember our outlier example from earlier)?
- Do your tables have dependent columns such that if you specify a request for a value from one column, you really don't need to filter on the other column as it would produce the same result?
- Is data loaded randomly, or in a specific order?

This is easiest to do if you actually have data already present. If you're trying to determine a collection strategy without real data and are unable to accurately answer these questions, you're likely going to be just as well covered to use automated collections. Even if you think you can answer the questions with some level of confidence, be prepared to adjust your plan over time if your answers were "off."

3.3 Dynamic Sampling

Another statistics collection option that many people often don't consider as anything more than a temporary fix is dynamic sampling. Dynamic sampling occurs when statistics are missing for an object. For example, object statistics may not exist for newly created objects (particularly for newly created partitions) or for temporary objects. Without statistics, the cost-based optimizer is in quite a bind. So rather than guess (use default values), the optimizer will attempt to collect some statistics during the query optimization process.

The collection that occurs at query optimization time depends on the setting for the `OPTIMIZER_DYNAMIC_SAMPLING` parameter or also can be determined by the presence of the `DYNAMIC_SAMPLING` query hint for a particular table. There are 11 valid settings (values in the range 0–10) for dynamic sampling and in Oracle 10g and above, the setting defaults to 2. A setting of 0 indicates no sampling is to be done and a setting of 10 indicates to sample all blocks in the table. The settings between 0 and 10 indicate increasing sample sizes and fewer restrictions on when dynamic sampling occurs.

The default setting of 2 is "good enough" to make sure a minimal sample is collected when statistics don't exist, but moving up to a level of 4 gives you much better coverage, particularly when you have correlated predicates (which I'll discuss more later). At level 4, the collection will occur for any unanalyzed table but it will also check selectivity estimates for predicates the reference 2 or more columns from the same table. Let's take a look at an example that shows exactly what this level of sampling means to the optimizer's ability to properly calculate row estimates.

First, we'll create a table that has two columns, `c1` and `c2`, that are guaranteed to always contain the same value.

© 2009 Method R Corporation. All rights reserved.

```
SQL>create table depend_test as
  2  select mod(num, 100) c1, mod(num, 100) c2, mod(num, 75) c3, mod(num, 30) c4
  3  from (select level num from dual connect by level <= 10001);

Table created.
```

Then, we'll gather table stats at 100% and no histograms on all columns.

```
SQL>exec dbms_stats.gather_table_stats( user, 'depend_test', estimate_percent => null, method_opt => 'for all columns size 1');

PL/SQL procedure successfully completed.
```

Next, we'll execute a query with a single predicate on `c1` that will return exactly 100 rows.

```
SQL>set autotrace traceonly explain
SQL>select count(*) from depend_test where c1 = 10;

Execution Plan
----------------------------------------------------------
Plan hash value: 3984367388
--------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |     1 |     3 |     8   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |             |     1 |     3 |            |          |
|*  2 |   TABLE ACCESS FULL| DEPEND_TEST |   100 |   300 |     8   (0)| 00:00:01 |
--------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("C1"=10)
```

Note that the row estimate is spot on. Now, let's modify the query to add the dependent/correlated column `c2`. Since both columns contain the same values, the query will still return exactly 100 rows. But, what does the optimizer think?

```
SQL>select count(*) from depend_test where c1 = 10 and c2 = 10;

Execution Plan
----------------------------------------------------------
Plan hash value: 3984367388
--------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |     1 |     6 |     8   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |             |     1 |     6 |            |          |
|*  2 |   TABLE ACCESS FULL| DEPEND_TEST |     1 |     6 |     8   (0)| 00:00:01 |
--------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("C1"=10 AND "C2"=10)
```

Now, we have a problem. Notice the row estimate. It's now 1; not 100. The point is that the optimizer, by default, thinks predicates are independent of each other. When you have dependent columns, as is the case here, the optimizer computes selectivity without understanding the dependency properly and thus underestimates. The cardinality calculation for ANDed predicates is

$$\text{cardinality} = (\text{selectivity c1}) \times (\text{selectivity c2}) \times \#\ \text{rows in table}$$

So, in this case, since each column has a selectivity of .01 (1/10), the cardinality ends up as 1.

Now, let's modify the query to add the `dynamic_sampling` hint at level 4. Note that I also tested levels 0–3, but since the results didn't change until I used level 4, I didn't include that output here to save space.

```
SQL>select /*+ dynamic_sampling (4) */ count(*) from depend_test where c1 = 10 and c2 = 10;

Execution Plan
----------------------------------------------------------
Plan hash value: 3984367388
```

© 2009 Method R Corporation. All rights reserved.

```
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |     1 |     6 |     8   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |             |     1 |     6 |            |          |
|*  2 |   TABLE ACCESS FULL| DEPEND_TEST |   100 |   600 |     8   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("C1"=10 AND "C2"=10)

Note
-----
   - dynamic sampling used for this statement
```

The optimizer's rows estimate is now correct! This test was designed to show you just the cardinality estimate issue. Use this knowledge and imagine how this could affect the optimizer's query plan choices. It could cause the optimizer to choose incorrect access paths and join orders and cause you to end up with a bunch of misbehaving queries.

The level 4 dynamic sampling setting does some powerful magic to help the optimizer really understand dependent predicate correlation. Note, however, that I used the `dynamic_sampling` hint and did not change the dynamic sampling instance parameter for this test. But, the test is representative of the behavior you'll get from the optimizer at this level regardless of whether you choose to set the value instance-wide or do it on a query-by-query basis. When you're deciding how to use dynamic sampling, make sure to carefully consider whether you want all queries to be exposed to dynamic sampling or if you only want to specify certain queries by hinting them.

For those of you already using Oracle 11g or whenever you do upgrade, you have a great new option called extended statistics. You can create extended statistics that describe the correlation between columns to the optimizer. Instead of using a dynamic sampling level 4 hint as I demonstrated using in 10g, you can now use the procedure `dbms_stats.create_extended_stats` to create a derived statistic based on multiple columns.

```
SQL> select dbms_stats.create_extended_stats(ownname=>user,
  2  tabname => 'DEPEND_TEST', extension => '(c1, c2)' ) AS c1_c2_correlation
  3  from dual ;

C1_C2_CORRELATION
--------------------------------------------------------------------------------
SYS_STUF3GLKIOP5F4BOBTTCFTMXOW

SQL> exec dbms_stats.gather_table_stats( user, 'depend_test');

PL/SQL procedure successfully completed.

SQL> set autotrace traceonly explain
SQL> select count(*) from depend_test where c1 = 10 and c2 = 10;

Execution Plan
----------------------------------------------------------
Plan hash value: 3984367388
-------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |     1 |     6 |     9   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |             |     1 |     6 |            |          |
|*  2 |   TABLE ACCESS FULL| DEPEND_TEST |   100 |   600 |     9   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("C1"=10 AND "C2"=10)
```

Dynamic sampling may be an excellent alternative to full statistics collections. The price you pay is found in the extra overhead for the samples to be collected during the initial hard parse of the query. If you set the instance parameter instead of hinting queries individually, keep in mind that you *could* pay the additional price for the samples to be collected for every query that is executed. I say *could* because not every query would use dynamic sampling. That's

© 2009 Method R Corporation. All rights reserved.

why it is important to test and verify the benefits gained from increasing the dynamic sampling level. The resources needed to do the sampling work may add more overhead than may be necessary or desired.

3.4 Setting Statistics Manually

Another method for providing the optimizer with statistics is to manually set the values. In my experience, many people gasp in horror at the prospect of setting statistics themselves. Well, not only is it possible, it's quite easy to do with the `dbms_stats` procedures `set_column_stats`, `set_index_stats`, and `set_table_stats`. The prevailing thinking is that Oracle will do a better job executing a formal collection and somehow our manual settings will be less accurate. Or, perhaps it's just the desire to be able to point the finger at Oracle if statistics are "bad". Whatever the reason, you should know that statistics are not the Holy Grail. You can touch and change them as you wish and it's really OK. Even Oracle supports that it is perfectly reasonable to set statistics manually in Metalink note 157276.1 (from way back in 2003).

The best reason to set statistics manually is to provide accurate and current values for some objects to the optimizer without having to burn the resources to run a collection. For example, let's say that you just loaded 5 million rows into a table and you know from the load process the exact counts and distribution of values within key columns for the new data. Why not modify the statistics for that table directly and not incur the expense of doing a `gather_table_stats` procedure?

There are times when the collection process, even if you do it at 100%, will not paint the perfectly correct picture of your data. A common misconception is that 100% collections should be "perfect." This isn't necessarily true. Index clustering factor is one statistic that can be collected at 100% and cause plans that are totally wrong. Clustering factor is a measure of how well ordered the data is as compared to the index order and is used to help the optimizer determine estimated block accesses.

Let's say that there were only a few blocks in your table but the rows had been inserted into these blocks in such a way that inserts were made on an alternating basis. In other words, row 1 was inserted in block 100, row 2 was inserted in block 101, row 3 was inserted in block 100, row 4 was inserted in block 101, and so on.

Data Block 100
Row 1 (Adams)
Row 3 (Cole)
Row 5 (Farmer)

Data Block 101
Row 2 (Brown)
Row 4 (Davis)
Row 6 (Higgins)

Now, create an index and note how the order of entries has this alternating pattern back and forth between the two blocks.

Index Value	ROWID
Adams	Block 100 Row 1
Brown	Block 101 Row 2
Cole	Block 100 Row 3
Davis	Block 101 Row 4
Farmer	Block 100 Row 5
Higgins	Block 101 Row 6

During a statistics collection, the index clustering factor is calculated by checking the `rowid` for each value and adding to a count whenever the block number changes. In our scenario, the block number will change for every value. This causes the clustering factor to be very high. When clustering factor is high, the optimizer will cost the use of that index for a range scan much higher than it should because it will compute that many, many different blocks will need to be accessed. But the truth is that only two blocks really need to be visited.

You can tell the optimizer the "real truth" by manually setting the clustering factor for that index yourself. You'd need to do a bit more work to determine the best setting, but by doing so the optimizer will have better data with which to make a better choice to use the index when it may have chosen to avoid the index otherwise.

© 2009 Method R Corporation. All rights reserved.

Don't shy away from manually setting statistics. Manual control is an important tool in your toolbox for helping the optimizer do the best job it possibly can to derive execution plans for your queries.

4 COMMON PERFORMANCE PROBLEMS

Statistics that don't accurately represent your actual data are the root cause for many common performance problems. When statistics are stale or non-existent, and misrepresent the actual state of your data, the optimizer can make improper choices for execution plan operations. Plan choices go wrong because the optimizer can't compute accurate enough cardinality estimates.

We've already looked at two problem cases where statistics don't properly represent the data. We reviewed an example where an outlier value that wasn't originally recognized in the statistics data caused the optimizer to use a full table scan when an index scan would've been a better choice. We also saw how the optimizer assumes that all predicates are independent and needs help to compute the correct cardinality when multiple columns are correlated to each other.

Let's look at three other areas where non-representative statistics cause problems:

- data skew
- bind peeking
- incorrect high and low value statistics

4.1 Data Skew

By default, the optimizer assumes that the values for a column are uniformly distributed. As long as this is true, the optimizer can make a fairly accurate computation of the percentage of rows containing each value. For example, if there are 4 distinct values for the column named color and there are 100 records in the table, as we've already seen, the optimizer will compute that regardless of which color you wish to select, 25 rows will be returned (`1/4 × 100 = 25`).

However, if the values for a column are not uniformly distributed, the optimizer will be unable to accurately compute how many rows match a given value. In order to get the calculations correct, the optimizer needs to have a histogram statistic for such columns.

You collect histograms using the `METHOD_OPT` parameter of the statistics gathering procedures in *DBMS_STATS*. When properly collected, the selectivity for a given value is more accurate. This simple example using a test table built using the `ALL_OBJECTS` view illustrates my point. The table contains approximately 1.6 million rows and an index on the `object_type` column. Statistics were collected at 100% with no histograms and the `object_type` column has 36 distinct values. The first query's predicate wants to retrieve rows where the `object_type='PROCEDURE'`, and the second query wants to retrieve rows where the `object_type='SYNONYM'`.

```
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------------
SQL_ID  16yy3p8sstr28, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ owner, object_name, object_type, object_id, status from
obj_tab where object_type = 'PROCEDURE'

Plan hash value: 2862749165
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID| OBJ_TAB      |      1 |  44497 |   2720 |00:00:00.06 |    1237 |
|*  2 |   INDEX RANGE SCAN          | OBJ_TYPE_IDX |      1 |  44497 |   2720 |00:00:00.02 |     193 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("OBJECT_TYPE"='PROCEDURE')
```

Notice the difference between the `E-Rows` and `A-Rows` values. The optimizer calculated that 44,497 rows would be returned based on the premise that the column values were uniformly distributed. Based on this result, the actual

© 2009 Method R Corporation. All rights reserved.

number of rows returned was more than 16 times less than the estimate. But, even with this poor estimate, the optimizer chose to use an index scan that was the correct operation in this case.

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------
SQL_ID  9u6ppkh5mhr8v, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ owner, object_name, object_type, object_id, status from obj_tab
where object_type = 'SYNONYM'

Plan hash value: 2862749165

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID| OBJ_TAB      |      1 |  44497 |    640K|00:00:14.25 |     104K|  19721 |
|*  2 |   INDEX RANGE SCAN          | OBJ_TYPE_IDX |      1 |  44497 |    640K|00:00:04.52 |   44082 |   1696 |
-----------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OBJECT_TYPE"='SYNONYM')
```

In the second case, the estimate is still the same (as you'd expect), but the actual number of rows returned was over 14 times higher. As far as the optimizer is concerned, the index scan was still the best plan choice. But, what would the optimizer do if a histogram existed on the `object_type` column that could accurately reflect the true distribution of values? Let's collect a histogram, using the default `SIZE AUTO` for `METHOD_OPT` and see what happens.

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------
SQL_ID  16yy3p8sstr28, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ owner, object_name, object_type, object_id, status from obj_tab
where object_type = 'PROCEDURE'

Plan hash value: 2862749165
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID| OBJ_TAB      |      1 |   2720 |   2720 |00:00:00.07 |    1237 |     29 |
|*  2 |   INDEX RANGE SCAN          | OBJ_TYPE_IDX |      1 |   2720 |   2720 |00:00:00.02 |     193 |      0 |
-----------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("OBJECT_TYPE"='PROCEDURE')
```

This time the row-count estimate is exact. The optimizer still chose the index scan, which is what we had hoped for.

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
SQL_ID  9u6ppkh5mhr8v, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ owner, object_name, object_type, object_id, status
from obj_tab where object_type = 'SYNONYM'

Plan hash value: 2748991475
-------------------------------------------------------------------------------------------------
| Id  | Operation         | Name    | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------------
|*  1 |  TABLE ACCESS FULL| OBJ_TAB |      1 |    640K|    640K|00:00:03.36 |   64263 |  21184 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("OBJECT_TYPE"='SYNONYM')
```

The rows estimate is perfect for our second query too. Because it's right on the money, notice how the optimizer chose a full table scan operation instead of an index scan. The performance difference is evident if you compare the

© 2009 Method R Corporation. All rights reserved.

original plan using the index. Using the index, the query took 14.25 seconds to execute and had to read 104,000 buffers. With the histogram in place and the choice to use a full table scan, the query execution time dropped to 3.36 seconds and only 64,263 buffer accesses.

As you can see, when the optimizer has the *right* statistics, it is able to make better plan choices that give you the kind of performance you want and need. Your job is to make sure you provide the optimizer with what it needs. That means you need to know your data well enough to know which columns may experience skew and collect histograms on those columns. Of course, you can always let Oracle determine which columns need histograms by using the `METHOD_OPT=>SIZE AUTO` option. Just remember to confirm Oracle's decisions against your own knowledge of the data and look for issues like the outlier value problem we reviewed earlier.

4.2 Bind Peeking

Since Oracle 9i, the optimizer has been able to "peek" at the values contained in bind variables used in your query during the hard parse phase. Prior to version 9, Oracle did not check bind values until after the plan decision had already been made. What this meant was that if you had collected histograms and then used a bind variable instead of a literal in your query, the optimizer would use a default selectivity value for the column and be unable to use the histogram. If you think about it, it makes sense. The literal value in our example above was checked against the histogram to determine the specific selectivity for that value. But, if the optimizer didn't know the value at parse time, it had to "guess" at the selectivity. The guesses usually favored the use of an index since the default guess was 5% selectivity.

But, as of version 9, the optimizer checks the value of the bind variable and is able to compute selectivity as if it were coded as a literal. This is both a good thing and a not-so-good thing. It's obviously good because we know the optimizer will be able to use actual column statistics to determine the best plan choice. But, where it goes wrong is in exactly the same place it goes right!

Let's use our previous example. First, assume the query is executed with a bind variable instead of a literal and the bind variable is set to `PROCEDURE`. This should give us the same plan as we got when using the literal (an index scan). But, the problem comes in when we run the same query a second time and set the bind variable to `SYNONYM`. What do you want the optimizer to do?

Since the query is identical, except for the bind variable value, the optimizer will not attempt to hard parse the query a second time. Instead, it will retrieve the previously selected plan and use that one (i.e., it will do a soft parse). Since the stored plan used was an index scan operation, that's what will be used to execute our request to retrieve `SYNONYM` rows. And, as we have already seen, the index scan is a very poor performing choice in that case.

In cases where bind variables are used for columns with uniformly distributed values, the plan choice wouldn't change with different bind values. The problems with bind peeking are primarily related to columns with skewed data that require histograms to make proper plan choices based on specific values. The good news is that in Oracle version 11, intelligent cursor sharing is available. This feature allows cursors using bind variables the possibility of getting new plans when the bound value changes.

When a query executes that contains a predicate comparing a skewed column (i.e., it has a histogram) to a bind variable twice (once for a skewed column value and once for a non-skewed value), the query is marked as bind sensitive. You can see this notation in the `is_bind_sensitive` column of `v$sql`. Since the presence of the histogram indicates that the column is skewed, different values of the bind variable may call for different plans.

In addition to bind sensitivity, there is one other new feature that comes into the picture: bind-aware cursors. A cursor is bind aware when a query is executed with two different bind values that should require a different plan (in our previous example, one would need an index scan and the other would need a full table scan). As queries execute, Oracle monitors the behavior of the queries and determines whether different bind values cause the data volumes manipulated by the query to be significantly different. If so, Oracle adapts its behavior so that the same plan is not always shared for this query. Instead of reusing the original plan, a new plan is generated based on the current bind value for subsequent executions.

In Oracle version 11, we can see how intelligent cursor sharing kicks in and provides the correct plan for both bind variable values when data skew is present. This example, like our data skew example above from Oracle 10, uses a test table built using the `ALL_OBJECTS` view with approximately 2.1 million rows and an index on the `object_type`

© 2009 Method R Corporation. All rights reserved.

column. Statistics are collected at 100% with a histogram on the `object_type` column. We'll query first for where the `object_type='PROCEDURE'` and the second query will retrieve rows where the `object_type='SYNONYM'`. The histogram tells the optimizer that there are very few rows containing the value PROCEDURE and many rows containing the value SYNONYM.

```
SQL> variable objtype varchar2(19)
SQL> exec :objtype := 'PROCEDURE';

PL/SQL procedure successfully completed.

SQL> select /*+ gather_plan_statistics */ count(*) ct
  2  from big_tab
  3  where object_type = :objtype ;

            CT
--------------
          4416

1 row selected.

SQL>
SQL> select * from table(dbms_xplan.display_cursor('211078a9adzak',0,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID  211078a9adzak, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ count(*) ct from big_tab where
object_type = :objtype

Plan hash value: 154074842
-----------------------------------------------------------------------------------------------------
| Id  | Operation         | Name            | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE   |                 |      1 |      1 |      1 |00:00:00.03 |      16 |
|*  2 |   INDEX RANGE SCAN| BIG_OBJTYPE_IDX |      1 |   4416 |   4416 |00:00:00.01 |      16 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("OBJECT_TYPE"=:OBJTYPE)

SQL> select child_number, executions, buffer_gets, is_bind_sensitive, is_bind_aware, is_shareable
  2  from v$sql where sql_id = '211078a9adzak' ;

   CHILD_NUMBER      EXECUTIONS     BUFFER_GETS IS_BIND_SENSITIVE IS_BIND_AWARE   IS_SHAREABLE
--------------- --------------- --------------- ----------------- --------------- ---------------
              0               1              16 N                 N               Y

SQL> exec :objtype := 'SYNONYM';

PL/SQL procedure successfully completed.

SQL>
SQL> select /*+ gather_plan_statistics */ count(*) ct
  2  from big_tab
  3  where object_type = :objtype ;

            CT
--------------
         854176

1 row selected.

SQL>
SQL> select * from table(dbms_xplan.display_cursor('211078a9adzak',0,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID  211078a9adzak, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ count(*) ct from big_tab where object_type = :objtype
```

© 2009 Method R Corporation. All rights reserved.

```
Plan hash value: 154074842

-------------------------------------------------------------------------------------------------
| Id  | Operation         | Name            | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE   |                 |      1 |      1 |      1 |00:00:04.33 |    2263 |
|*  2 |   INDEX RANGE SCAN| BIG_OBJTYPE_IDX |      1 |   4416 |    854K|00:00:01.71 |    2263 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("OBJECT_TYPE"=:OBJTYPE)

SQL> select child_number, executions, buffer_gets, is_bind_sensitive, is_bind_aware, is_shareable
  2  from v$sql where sql_id = '211078a9adzak' ;

   CHILD_NUMBER      EXECUTIONS     BUFFER_GETS IS_BIND_SENSITIVE IS_BIND_AWARE   IS_SHAREABLE
--------------- --------------- --------------- ----------------- --------------- ---------------
              0               2            2279 Y                 N               Y

1 row selected.
```

Notice how both executions of the query used the same index range scan plan. But due to the skewed distribution, the estimated row count of 4,416 was substantially too low. Also note that the count of buffer gets went up from 16 to 2,263. Based on this big difference, Oracle now marks the query as bind sensitive. Let's execute the query again with the same bind value (SYNONYM).

```
SQL> select /*+ gather_plan_statistics */ count(*) ct
  2  from big_tab
  3  where object_type = :objtype ;

            CT
---------------
        854176

1 row selected.

SQL>
SQL> select * from table(dbms_xplan.display_cursor('211078a9adzak',1,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
----------------------------------------
SQL_ID  211078a9adzak, child number 1
-------------------------------------
select /*+ gather_plan_statistics */ count(*) ct from big_tab where
object_type = :objtype

Plan hash value: 1315022418
-----------------------------------------------------------------------------------------------------
| Id  | Operation             | Name            | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE       |                 |      1 |      1 |      1 |00:00:04.44 |    6016 |
|*  2 |   INDEX FAST FULL SCAN| BIG_OBJTYPE_IDX |      1 |    854K|    854K|00:00:01.85 |    6016 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("OBJECT_TYPE"=:OBJTYPE)

SQL> select child_number, executions, buffer_gets, is_bind_sensitive, is_bind_aware, is_shareable
  2  from v$sql where sql_id = '211078a9adzak' ;

   CHILD_NUMBER      EXECUTIONS     BUFFER_GETS IS_BIND_SENSITIVE IS_BIND_AWARE   IS_SHAREABLE
--------------- --------------- --------------- ----------------- --------------- ---------------
              0               2            2279 Y                 N               N
              1               1            6016 Y                 Y               Y

2 rows selected.
```


© 2009 Method R Corporation. All rights reserved.

This time, the optimizer computed a new plan (index fast full scan) for this bind value. A new child cursor has been added and marked as bind aware. Also notice that the original child cursor (0) has been marked as not shareable. The original cursor was discarded when the cursor switched to bind aware mode. This is a one-time overhead. The cursor is marked as not shareable which means that this cursor will be among the first to be aged out of the cursor cache, and that it will no longer be used. In other words, it is just waiting to be garbage collected.

Now, if we execute the query with a bind value that is more selective, we should get the index range scan plan again. But, since that cursor has been marked as not shareable, what happens?

```
SQL> exec :objtype := 'PROCEDURE';

PL/SQL procedure successfully completed.

SQL>
SQL> select /*+ gather_plan_statistics */ count(*) ct
  2  from big_tab
  3  where object_type = :objtype ;


            CT
---------------
          4416

1 row selected.

SQL>
SQL> select * from table(dbms_xplan.display_cursor('211078a9adzak',2,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
----------------------------------------
SQL_ID  211078a9adzak, child number 2
-------------------------------------
select /*+ gather_plan_statistics */ count(*) ct from big_tab where
object_type = :objtype

Plan hash value: 154074842
-------------------------------------------------------------------------------------------------------
| Id  | Operation         | Name            | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE   |                 |      1 |      1 |      1 |00:00:00.04 |      16 |
|*  2 |   INDEX RANGE SCAN| BIG_OBJTYPE_IDX |      1 |   4416 |   4416 |00:00:00.02 |      16 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("OBJECT_TYPE"=:OBJTYPE)

SQL>
SQL> select child_number, executions, buffer_gets, is_bind_sensitive, is_bind_aware
  2  from v$sql where sql_id = '211078a9adzak';


   CHILD_NUMBER      EXECUTIONS     BUFFER_GETS IS_BIND_SENSITIVE IS_BIND_AWARE   IS_SHAREABLE
--------------- --------------- --------------- ----------------- --------------- ---------------
              0               2            2279 Y                 N               N
              1               1            6016 Y                 Y               Y
              2               1              16 Y                 Y               Y

3 rows selected.
```

The answer is that we get a new child cursor for the index range scan plan. Oracle version 11 uses additional cursors for such queries because when a new bind value is used, the optimizer tries to find a cursor that it thinks will be a good fit, based on similarity in the bind value's selectivity. If it cannot find such a cursor, it will create a new one. If the plan for the new cursor is the same as one of the existing cursors, the two cursors will be merged, to save space in the cursor cache. This will result in one being left behind that is in a not shareable state. This cursor will be aged out first if there is crowding in the cursor cache, and will not be used for future executions.

© 2009 Method R Corporation. All rights reserved.

The whole bind peeking problem seems to be solved with Oracle version 11. And, in the strictest terms, you could say that bind peeking isn't really a statistics issue. …It's a coding issue. Until you get to version 11, you'll need to make sure to properly code your applications to account for the effects of bind peeking. If you know what values are skewed, you may choose to code conditional logic that tests for the known skewed values and hard-codes those literals into your SQL. If the value isn't one that is skewed, you'll simply branch to a SQL statement that uses a bind variable. The key thing to keep in mind is the number of distinct values that may be skewed. If the number is low, then hard-coding shouldn't create too many unique cursors in the shared pool. If the number is high, you'll have to weigh the cost of having many unique cursors in the shared pool (and the associated overhead of the hard parse activity) against the benefit of having the optimizer be able to choose the proper plan for different values.

4.3 Incorrect High and Low Value Statistics

High and low value statistics for a column are used in the optimizer's calculations for determining selectivity for range predicates. For example, if there is a low value of 0 and a high value of 10 for `colx` and you ask for where `colx > 8`, the computation will be

```
selectivity = (high value – colx value) / (high value - low value)
```

or selectivity=.2. Therefore, if the table had 100 rows, the optimizer would estimate that 20 rows (.2 × 100) would be returned. A similar formula would be in effect if you asked for `colx < 8`. The computation will be

```
selectivity = (colx value – low value) / (high value – low value)
```

or selectivity=.8. Therefore, our resulting row estimate would be 80 rows (.8 × 100).

Of course, there are other operators to consider such as <= or >=. Again, I'll recommend that you read Jonathan Lewis' *Cost-Based Oracle Fundamentals* book (Apress, 2006) for a much more detailed treatment of these calculations.

What I'd like you to understand is what happens if you execute a query that uses a predicate outside the range of the low to high value statistics. In other words, what if your statistics do not include the correct low and high column values? The main problem is that the optimizer's cardinality estimate will not accurately represent the current data. Typically, that means the estimate will be too low. For example, I have a table with 100,000 rows and a uniformly distributed range of values in column `colx` between 9 and 18. If I query outside this range on either side, Oracle will compute the selectivity in a slightly different manner. If I query for where `colx < 8`, the computation will be

```
selectivity = ((colx value – low value) / (high value – low value) + 1) × .10
```

or selectivity=.088889. Therefore, our resulting row estimate would be 8889 rows (.088889 × 100,000).

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
SQL_ID  9fah3fcf24nx5, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ count(a) from hi_lo_t where b < 8

Plan hash value: 3460024886

-----------------------------------------------------------------------------------------------
| Id  | Operation         | Name       | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE   |            |      1 |      1 |      1 |00:00:00.01 |       2 |
|*  2 |   INDEX RANGE SCAN| HILO_B_IDX |      1 |   8889 |      0 |00:00:00.01 |       2 |
-----------------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("B"<8)
```

The interesting thing to notice about this calculation is that as the value of `colx` moves farther away from the actual low (or high) value, the cardinality estimate moves toward 1. At some point, as the estimate gets smaller and smaller, this can cause the optimizer to make very different choices about the execution plan, particularly in relation to its choice of join order.

© 2009 Method R Corporation. All rights reserved.

Oracle typically does a good job of getting fairly accurate low and high value statistics during a normal collection. However, you can run into problems if you've recently added a significant amount of data and the statistics are now stale.

This seems to be particularly problematic with partitioned tables when partition statistics are copied from another partition and then manually adjusted. One process I've seen that can cause issues with correct plan choice occurs when a new partition is added and the statistics are copied from the previous partition. In order for this process to be complete, the statistics should then be adjusted with values matching the newly added data. That means that the `low_value`, `high_value`, `num_distinct`, and other statistics need to be changed to correctly represent the data in the new partition. If that doesn't happen, or if not all the proper statistics get modified, the optimizer can have a very hard time generating an optimal execution plan. In this particular case, dynamic sampling is likely a better choice than risking problems caused by incorrectly setting statistics manually.

Another case where problems arise is when statistics are collected using a `METHOD_OPT` choice that doesn't collect column statistics for all columns. For example, if you choose `METHOD_OPT=>'FOR ALL INDEXED COLUMNS'`, only statistics for indexed columns will be collected. If you then use a column that isn't indexed in a query, the optimizer has no choice but to use a default "guess" for that column's statistics.

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------
SQL_ID  6nawp6qk6yqcy, child number 1
-------------------------------------
select /*+ gather_plan_statistics */ count(a) from hi_lo_t where b = 12

Plan hash value: 3307858660

-------------------------------------------------------------------------------------------
| Id  | Operation          | Name    | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |         |      1 |      1 |      1 |00:00:00.07 |     184 |
|*  2 |   TABLE ACCESS FULL| HI_LO_T |      1 |   1000 |  10000 |00:00:00.05 |     184 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("B"=12)
```

In this example, column `B` didn't have an index and therefore no statistics were collected for it when using `METHOD_OPT=>'FOR ALL INDEXED COLUMNS'`. So, the optimizer used a default of 1%. Again, the problem is that the optimizer doesn't have proper representative information about the data and can easily make a wrong plan choice.

The main point to remember about the low and high value statistics as well as the other column statistics—`num_distinct`, `num_nulls`, and `num_buckets`—that affect the optimizer's ability to select an optimal execution plan is that inaccuracies in these values can cause the optimizer to compute cardinality estimates that are orders of magnitude different than actual. The more inaccurate the cardinality estimates are, the more likely it is that the optimizer will choose the wrong access path or join order. You must carefully consider your statistics collection choices to make sure they provide the optimizer with the most representative picture of your data as possible.

5 STATISTICS COLLECTION STRATEGY

The statistics collection process can be intimidating, especially when there are so many options, both automatic and manual, for you to take into consideration. Often, how statistics are collected isn't strategically determined but rather a case of "we tried it, it worked, so we just leave it that way". Based on our review so far, I hope it is clear that having representative statistics is a must if you want your applications to be stable and perform well.

As Oracle's automated collections of statistics has matured, it is becoming more and more feasible to start with what Oracle determines is best (i.e., use automated collection features) and adjust as needed from there. Oracle has documented significant improvements for version 11 in the quality of automated collections as well as notable

© 2009 Method R Corporation. All rights reserved.

improvements in the time it takes to complete a collection.[6] Perhaps we are actually getting closer to a near-automatic solution. However, if any of the issues we've discussed here are present in your environment (e.g., data skew, outlier values, dependent predicates), then you will need to perform some amount of manual collections or manual setting of statistics to cover these situations. And of course, there's always dynamic sampling to keep in mind as well.

5.1 Using Dynamic Sampling as Part of your Overall Strategy

I've become more convinced in recent months that dynamic sampling is an excellent alternative to implement for longer-running queries, particularly when the objects those queries reference have missing or stale statistics quite frequently. Robyn Sands, Oak Table member and friend, was kind enough to execute some dynamic sampling tests for me during her testing of a new application release. Robyn has written a paper on using variance as a tool for performance measurement,[7] so her tests show the variance in response time and CPU usage between different environments. There were eleven processes executed under 4 different environment scenarios. The two scenarios I was most interested in were the two runs that compared having no statistics present and using only dynamic sampling at level 4 versus having a 100% collection with the `optimizer_dynamic_sampling` parameter set to 4. The following data shows the comparison of the two runs and how the response times (total and CPU) varied over 6 executions of each process.

Variance between 100% Statistics Collection and Dynamic Sampling (level 4)
Run 3 = Dynamic sampling level 4 only
Run 4 = 100% collection plus dynamic sampling level 4

Job#	Run#	Avg Elapsed Time	Variance	VMR*	COV*	Avg Elapsed CPU Time	CPU Variance	CPU VMR	CPU COV
1	3	184.105	15467.349	84.014	0.676	183.798	15431.0000	83.816	0.675
1	4	182.778	16014.809	87.619	0.692	182.003	16013.4460	87.611	0.692
2	3	8.603	0.762	0.089	0.101	0.330	0.0010	0.000	0.003
2	4	9.205	8.016	0.871	0.308	0.318	0.0010	0.000	0.003
3	3	121.448	774.920	6.381	0.229	93.925	480.4410	3.956	0.180
3	4	108.087	2462.103	22.779	0.459	81.168	1959.2970	18.127	0.410
4	3	252.977	1225.972	4.846	0.138	51.935	39.3670	0.156	0.025
4	4	252.972	1218.028	4.815	0.138	51.547	40.6930	0.161	0.025
5	3	55.330	336.283	6.078	0.331	55.197	334.3020	6.042	0.330
5	4	54.725	327.892	5.992	0.331	54.668	327.4130	5.983	0.331
6	3	158.192	5435.606	34.361	0.466	0.487	0.0080	0.000	0.001
6	4	158.972	5524.136	34.749	0.468	0.487	0.0070	0.000	0.001
7	3	2.533	0.472	0.186	0.271	0.138	0.0004	0.000	0.008
7	4	2.397	0.706	0.295	0.351	0.133	0.0007	0.000	0.012
8	3	85.302	2006.418	23.521	0.525	0.193	0.0004	0.000	0.000
8	4	92.003	2510.789	27.290	0.545	0.190	0.0005	0.000	0.000
9	3	10.697	1.624	0.152	0.119	0.183	0.0001	0.000	0.001
9	4	10.632	1.543	0.145	0.117	0.180	0.0001	0.000	0.001
10	3	82.005	1413.944	17.242	0.459	81.735	1404.8478	17.131	0.457
10	4	81.582	1298.805	15.920	0.442	81.388	1295.5376	15.880	0.441
11	3	31.465	46.875	1.490	0.218	0.420	0.0008	0.000	0.001
11	4	32.793	59.968	1.829	0.236	0.408	0.0005	0.000	0.001

*VMR - Variance to Mean Ratio. A measure of dispersion (randomness) in a given phenomena. High VMR is an indicator for code with the greatest potential for improvement and can be used to prioritize targets for performance improvements.

*COV - Coefficient of Variance. A measure of volatility (risk) you are assuming in comparison to the expected return (average). High COV is an indicator showing how much changes in the quantity of data will affect response time.

I was very pleased to see how well using only dynamic sampling performed, particularly in terms of variances. More often than not, Run 3, that used only dynamic sampling, had a variance in both total elapsed time and CPU time across

[6] Inside the Oracle Optimizer Blog. *Improvement of AUTO sampling statistics gathering feature in Oracle 11g.* January 22, 2008. http://optimizermagic.blogspot.com/2008/01/improvement-of-auto-sampling-statistics.html

[7] Robyn's paper and presentation on variance can be found at http://optimaldba.com/papers.

© 2009 Method R Corporation. All rights reserved.

all executions that were less than Run 4 that used a 100% collection. I really like using the concept of variance as a performance indicator. And with statistics playing such a key role in the optimizer's ability to determine the best execution plans *most of the time,* I think Robyn has hit on a great way to monitor how well your statistics deliver consistent plan performance. I know that you can't draw a definitive conclusion from a single test case like this one, but I do think it speaks very highly of how well dynamic sampling can work.

5.2 What To Do and What *Not* To Do

Selecting the best statistics management strategy is often a matter of simply *not* doing things that set you up for problems. If you consider all the areas where statistics can go wrong as we've discussed here, the bottom-line is one of my company's favorite sayings, "Why *guess* when you can *know.*"

Use the following as a checklist of what *not* to do:

- Don't guess why you're currently using the statistics strategy you are; find out. The person who originally created the method in use for statistics collection may not even be there any more. Find out not only what is being done, but why, and document it thoroughly.
- Don't make any changes to your statistics collection process without thorough testing. This particularly applies to upgrading Oracle versions. Remember that automatic statistics collection options change with each version.
- Don't wait till you have problems to look at your statistics. Review the statistics after each collection, compare previous values to current values, and monitor response times and variances for at least a key subset of your application's most common queries. Starting in Oracle 10.2.0.4 the `DBMS_STATS` package contains a set of functions named `DIFF_TABLE_STATS_*` that can be used to produce difference reports between different sets of statistics. If you're using an earlier version, you'll have to write your own.
- Don't ignore your data. Good statistics collection strategies are founded on a thorough knowledge of your data. Understanding data distributions and how data is most commonly queried will help you monitor key adjustment areas (i.e., objects that need special treatment).

Note that this is not a checklist of items to put on your to-do list. It's intended to be a list of behaviors that if you choose to implement them, you'll find more success in managing statistics in your environment. Don't let anyone fool you! There is no single strategy that works best for everyone. Oracle is doing its best to provide automated statistics collection options that cover most common environments most of the time. But, a 100% solution (or at least as close to 100% as you can get) can only be derived with careful study and testing on your part.

6 SUMMARY

I believe the Oracle optimizer is *the* critical player in determining how well your application will perform. There are many touch points in the Oracle database that, if not managed properly, can lead to performance bottlenecks. But your applications rely on getting data from the database and it's the optimizer's job to decide on what operations will be used to deliver that data back to you. Your knowledge about how the optimizer uses statistics, how different statistics collection methods obtain the data the optimizer will use, and how well you know your own data, all combine to provide the foundation of stable and consistent application performance.

Oracle continues to improve automated mechanisms for collecting statistics and how the optimizer uses the statistics to produce optimal execution plans. But automated processes are still lacking the ability to collect data that correctly paints a complete picture of your data in all cases. If Oracle can get you 60%, 70%, 80%, or more of the way towards an optimal collection, that's all the better. The remaining part of the journey is up to you!

With this paper, I hope I have illuminated the following key points for you:

- Statistics that reasonably represent your actual data are critical to the optimizer's ability to determine stable and consistently performing execution plans for your application SQL.
- Understanding basic optimizer statistics computations helps you determine when statistics are "good" or "bad."
- Oracle's automated statistics collection methods are intended to provide statistically "good enough" information for most environments most of the time.

© 2009 Method R Corporation. All rights reserved.

- There are statistics related to data skew, outlier data, predicate dependency, and others, where automated collections using default options do not provide what the optimizer needs to correctly determine the optimal execution plan.
- Dynamic sampling provides statistics to the optimizer when none exist, when statistics are stale, or when dependent predicates exist.
- Manually setting column statistics allows precise control over specific statistic values without requiring the resource utilization of a full collection.
- Testing and monitoring of statistics should be an integral part of your application performance reviews.
- Knowing your data is critical to designing your statistics collection and management strategy.

I hope this paper encourages you to learn more about how the Oracle optimizer works, how statistics affect the optimizer's ability to produce optimal execution plans, how well you know your data, and how you can use this knowledge to create a strategy for statistics management that is best for you.

7 BIBLIOGRAPHY

Foote, Richard. *DBMS_STATS METHOD_OPT default behavior changed in 10g – be careful.* http://richardfoote.wordpress.com/2008/01/04/dbms_stats-method_opt-default-behaviour-changed-in-10g-be-careful/.

Inside the Oracle Optimizer Blog. *Improvement of AUTO sampling statistics gathering feature in Oracle 11g*. January 22, 2008. http://optimizermagic.blogspot.com/2008/01/improvement-of-auto-sampling-statistics.html.

Lewis, Jonathan. *Cost-Based Oracle Fundamentals*. Berkeley, CA: Apress, 2006.

Sands, Robyn. *An Industrial Engineer's Approach to Managing Oracle Databases*. Paper. http://optimaldba.com/papers/IEDBMgmt.pdf.

Sands, Robyn. *Using Variance as a Tool: Measuring for Robust Performance*. Presentation. http://optimaldba.com/papers/IEApproachToOracle.pdf.

Oracle Corporation. *Oracle Database Performance Tuning Guide 11g Release 1*. 2008. http://download.oracle.com/docs/cd/B28359_01/server.111/b28274/toc.htm.

—. *A Practical Approach to Optimizer Statistics in Oracle Database 10g*. Oracle OpenWorld 2005. http://www.oracle.com/technology/deploy/performance/pdf/PS_S961_273961_106-1_FIN_v2.pdf.

8 ABOUT THE AUTHOR

Karen Morton teaches application optimization to companies around the world, in both classroom settings and shoulder-to-shoulder consulting engagements. She is the director of education and services for Method R Corporation (*http://method-r.com*), a Southlake, Texas company founded by Cary Millsap. Method R offers consulting services, education courses, and software tools—including the Method R Profiler—that help you optimize your software performance.

For over 20 years, Karen has worked in information technology starting out as a mainframe programmer, developer, DBA, data architect and now as a researcher, educator and consultant. Having used Oracle since the early 90's, she began teaching others how to use Oracle over a decade ago.

She is a frequent speaker at conferences and user groups, an Oracle ACE, and a member of the OakTable Network (an informal association of "Oracle scientists" that are well known throughout the Oracle community). She blogs at http://karenmorton.blogspot.com.

© 2009 Method R Corporation. All rights reserved.