



Judicious Use of Histograms in SQL Tuning



By John Kanagaraj

Oracle enables SQL performance tuning via the use of Column Histograms. This procedure is neither well documented nor well understood, although the functionality exists and is used within the Database structures. In this article, we will describe how histograms can be used to tune SQL without major changes in SQL under certain circumstances. We will also showcase this with a real-life example in order to help the reader understand how to use histograms.



Introduction to the Optimizer and Histograms

Histograms are used in Oracle to describe the “skew” in the distribution of data in columns that hold non-unique, repeating values. In other words, when a few distinct values in that column form a sizeable portion of the row count, and this count is not uniformly distributed, then histograms are able to describe this distribution mathematically. Thus, when histograms are created on key columns such as those that contain a set of repeating values such as Department ID on an Employee Table, they are able to quantify the spread of distinct values. Under certain circumstances that we will explain later, histograms exert a great influence on Oracle’s Cost Based Optimizer (CBO) and help determine the optimal access path to data. An efficient access path usually results in good performance for the SQL statement being executed, which ultimately is the goal of tuning.

The Optimizer is at the heart of the Oracle kernel’s access layer and determines how the required data should be accessed. Until Oracle 7, the Rule Based Optimizer (RBO) was the only optimizer in use. In Oracle 7 and onward, Oracle Corporation made the CBO available in the Database. The difference is simple to explain: The RBO instructs the Access layer to get to the data based on a set of rules, while the CBO performs this using the *Estimated Cost of Access*. It does this by performing calculations based on

statistics about objects such as Tables, Indexes, Columns and Histograms, and decides to use the best access route or *execution plan* among many alternative paths that it has calculated, and chooses the plan that has the lowest *cost* of access. (The CBO in fact is turned on by the value *CHOOSE* in the *OPTIMIZER_MODE* initialization parameter.) This Execution plan can be determined using the *EXPLAIN PLAN* statement. When the SQL is traced, it can also be seen in raw form in the Trace file or a processed form in the *TKPROF* output after the trace file is processed. The executed plan can also be seen via the *V\$SQL_PLAN* view on Oracle 9i databases and above immediately after the completion of the execution.

Both optimizers produce execution plans. However, the RBO cannot use information about the object in generating an execution plan while the CBO is flexible enough to adapt its access path according to this information. Thus, if the size and nature of the objects varies, the CBO is able to adapt and adjust its execution plan accordingly. The CBO is also able to cater to new types of objects such as function-based indexes, partitioned tables, bitmap indexes, etc., as well as perform new types of access such as hash joins while the RBO does not. For these reasons, Oracle has been discouraging the use of the RBO and has in fact fully de-supported it from Oracle Database 10g onwards.

One problem though, is that by its very nature, the CBO is very sensitive to the veracity of object statistics—stale, incorrect or missing statistics will produce the wrong plan. As well, bugs and misunderstandings in the algorithms used to perform this “cost of access” calculations can also result in incorrect plans. The CBO thus needs to be fed the right statistics, and the statistical information in histograms is a crucial part of that. Indeed, this is the reason we are discussing the optimizer in such depths here!

Before we move on, let us state that histograms are essentially “buckets” that specify a range of values in a column. The Oracle kernel sorts the non-null values in the column and groups them into the specified number of these buckets so that each bucket holds the same number of data points, bounded by the end point value of the previous bucket. Histograms can be either height balanced or width balanced: Simply stated, if the distinct sets of values are less than the number of histograms that are required to be collected, a width-balanced histogram is created. Histograms can be created and used on numeric, character and date columns.

Histograms and the Execution Plan

SQL statements from any interface or tool need to be parsed before they are executed. This is the phase when syntax, semantic and access rights are checked, and the “execution plan” is created and stored in the Shared Pool section of the System Global Area (SGA). Once created, this plan drives the execution of the SQL and the data is accessed. In simple terms, when the CBO considers the value of the key that is to be used for retrieval (as used in a SQL WHERE clause), and histograms are available for that key column, the CBO calculates the “cost” of access for both an Indexed read as well as a full table scan (FTS), based on the frequency of occurrence of that key value. Thus, the CBO may decide to use a FTS rather than an indexed read for popular values that occur frequently, as this option may be less costly in terms of *total I/O*. On the other hand, the CBO may decide to use indexed reads for less popular keys, again based on the *total I/O*. In a slightly more complex setting, the CBO considers histograms while determining the optimal order in which tables should be joined. For example, when presented with the need to join multiple tables using join conditions on columns that contain histograms, the CBO is able to determine the order in which the tables should be joined, executing the join that is expected to produce the least number of rows first, followed by joins that produce successively larger numbers of rows.

All this under-the-hood rearrangement occurs without the programmer manipulating the SQL, and this concept thus lends itself to reduced program “tuning fixes” as data patterns change. This in fact is a key strength of the CBO and is an important reason for considering histograms (and reading this article!). Indeed the CBO considers much more than the factors described above, but for the purpose of this article, we will look only at this simple explanation.

As usual, there are some caveats. Up until Oracle 9i, the CBO will consider histograms during the SQL parse phase only if literal values (of the form WHERE KEY_ID = '123') rather than bind variables (of the form WHERE KEY_ID = :key_value) are used in SQL conditional clauses. From Oracle 9i Release 9.0.1 and above, the CBO is able to peek at the value of the bind variable prior to the parse, and use Histograms when they are available. The downside though in Oracle 9i and above is that the execution plan is generated based on the current value of the bind variable at the time of parsing and is thus “fixed” during the first parse, hence subsequent changes in the value of the bind variable does not change the path. This has resulted in a number of performance issues, and hence we need to know and keep this warning in mind. At this point, let us state that this functionality is turned on by default in Oracle 9i and above, but can be manually disabled via the undocumented parameter `_optim_peek_user_binds`. Of course, changes should be done only under Oracle Support’s direction!

Histograms are not automatically created on all key columns, nor should they be. Histograms are collected using the DBMS_STATS inbuilt PL/SQL package, which collects object statistics for tables, indexes and columns for use by the CBO. Histograms can also be collected by the ANALYZE Statement. However, Oracle recommends the use of the DBMS_STATS command as the latter is able to collect additional and more accurate information. Histograms can be viewed using the ALL_HISTOGRAMS, ALL_PART_HISTOGRAMS, ALL_TAB_HISTOGRAMS and their DBA_% equivalents. Columns containing histograms can be determined from the NUM_BUCKETS column in the ALL_TAB_COLUMNS – this value should be greater than one for Histograms to exist. Note that all columns that have statistics attached to them by virtue of being analyzed will have at least two entries in the ALL_TAB_COLUMNS view. Histogram information is available for those columns that have more than two rows in this view.

While histograms are generally beneficial to performance, they do impose overheads. These overheads occur both during Statistics collection for the extra processing and storage required, as well as during parse for extra processing, loading and storage in the Shared pool section of the SGA. In addition, the CBO might now have to calculate additional access path alternatives when considering histograms. When histograms are stale and do not reflect the underlying data pattern due to data pattern changes, they can also result in incorrect access paths that are not appropriate for that time. Thus, it is essential that the right amount of histograms be collected and be kept up to date as data changes. Since histograms can be collected along with other objects statistics at the same time, and the staleness of both affect the plans of the CBO, it is fair to say that histograms should be collected at the same time as Object statistics.

Memory is allocated to histograms from the Shared Pool area of the SGA. Usage can be seen in the Dictionary Cache Statistics section of a STATSPACK report below. The highlighted section shows both its presence and usage for the given period:

```
Dictionary Cache Stats for DB: TEST Instance: TEST Snaps: 4030 -4047
->"Pct Misses" should be very low (< 2% in most cases)
->"Cache Usage" is the number of cache entries being used
->"Pct SGA" is the ratio of usage to allocated size for that cache
```

Cache	Get Requests	Pct Miss	Scan Requests	Pct Miss	Mod Req	Final Usage	Pct SGA
dc_constraints	456	33.3	0		456	10	91
dc_database_links	378	0.3	0		0	4	80
dc_files	9,704	0.2	0		0	432	97
dc_free_extents	2,866	42.8	614	0.0	2,442	619	100
dc_global_oids	62,294	0.0	0		0	44	98
dc_histogram_data	51,111	0.2	0		4,230	62	78
dc_histogram_data_valu	3,377	1.4	0		0	9	53
dc_histogram_defs	2,167,706	1.8	0		6,812	8,309	100
dc_object_ids	16,211,087	0.1	0		71	2,655	100
dc_objects	747,025	0.9	0		8,989	2,788	100
dc_outlines	0		0		0	0	0

Evidence of histogram usage can also be seen in the SQL trace file in the form of recursive queries from UserID 0 (i.e. ‘SYS’), as shown below:

```
PARSING IN CURSOR #7 len=210 dep=2 uid=0 oct=3 lid=0 tim=2464046571 hv=787810128
ad='920ec318'
select /*+ rule */ bucket_cnt, row_cnt, cache_cnt, null_cnt, timestamp#,
sample_size, minimum, maximum, distcnt, lowval, hival, density, col#, spare1, spare2,
avgcln from hist_head$ where obj#=:1 and intcol#=:2
END OF STMT
```

Metalink Note 1031826.6 has an excellent overview of histograms and the Oracle RDBMS Performance Tuning Guide provides more details.

Generating Histograms

Histograms are generated via the DBMS_STATS inbuilt PL/SQL package using appropriate commands in the method_opt parameter of the DBMS_STATS.GATHER_<Object_Level>_STATS procedures (<Object_Level> can be any of DATABASE, SCHEMA, TABLE or INDEX). Histograms are collected only when the clause FOR ALL [INDEXED | HIDDEN] COLUMNS [size_clause] is set and the [size_clause] is set to a value more than 1. The number specifies the number of buckets that the Histogram should have and the maximum that be specified is 254.

Up until Oracle 9i, a size value of more than 1 will force histograms to be collected for every column, regardless of its nature. From Oracle 9i and upward, a new clause SIZE SKEWONLY ensures that histograms are collected only for columns that have skewed sets of values. Note that histograms are not required for columns that have unique indexes or are primary keys—by their very nature, each value in these columns is distinct and there is no value spread. Histograms provide their greatest value when collected on non-unique, indexed columns that are known to have a set of distinct values. The following shows the code for collecting histograms for a LARGE_TAB table in the TESHIST schema using both the SKEWONLY clause as well as the AUTO_SAMPLE_SIZE parameter for the estimate_percent clause. Analysis of an SQL trace of the statistics gathering session showed that the latter performs a number of samples before deciding on the best estimate_percent. In fact, the AUTO_SAMPLE_SIZE parameter attaches a SAMPLE (S) clause instructs Oracle to perform the analysis using a random sample of (S) percent of rows from the table, rather than from the entire table. The SKEWONLY clause instructs the Oracle kernel to collect appropriate number of histograms only for those columns that have skewed values. The full command is shown below:

continued on page 30

```
execute dbms_stats.gather_table_stats('TESTHIST','LARGE_TAB',-
estimate_percent=>DBMS_STATS.AUTO_SAMPLE_SIZE,-
method_opt=>'FOR ALL COLUMNS SIZE SKEWONLY', cascade=>TRUE);
```

A Real Life Example

Now that we have been introduced to histograms, as well as their significance and operation, let us consider how judicious application of histograms helped in a real life situation.

The problem was this: This involved a particular parameterized and customized report in an Oracle Applications Database. The report was completing quickly (within two to three minutes) for most parameters, but was taking about an hour and a half or more for another specific parameter. The parameter that affected the runtimes was the organization ID which happened to be a non-unique indexed key column with a few distinct values. Reports for smaller orgs ran quickly as they selected a smaller number of rows, but the reports for larger orgs ran for excessively long times. The tables involved were rather large tables in the INV, PO and BOM schemas, so some performance degradation was expected, but not by such amounts. The report in question was used by many users with many different types of parameters, and could not suffer functional changes. Thus, the challenge was to tune the performance without major rewrites and SQL changes.

The analysis went thus: Tracking the session statistics using V\$ internal views such as V\$SESSTAT and V\$SESSION_EVENT showed that the process was performing a large number of physical I/Os as well as a much larger number of logical I/Os. Operating system utilities indicated that the report consumed an excessive amount of CPU. To determine the exact problem, the program was modified to generate extended SQL trace data and rerun on a recently refreshed test instance. The extended SQL trace data, also known as a “10046 level 12 trace” generates a lot of trace information. This includes the various SQL statements being processed, as well as the wait events and run time statistics recorded against each of the SQL statements. The extended trace can be switched on using the SQL statement below:

```
alter session set events = '10046 trace name context forever, level 12';
```

The “level 12” is important: other levels are 0 to set the trace off, 1 for the normal SQL trace, 4 for information on bind variables, 8 for wait event data and 12 for a combination of everything.

The SQL itself and the resulting execution plan are too large to be shown here, and in any case is not required for our understanding of this topic. Hence, a snippet of the trace file with the relevant information for this SQL statement is shown below:

```
PARSING IN CURSOR #18 len=3541 dep=0 uid=70 oct=3 lid=70 tim=2464047492 hv=1945677245
ad='83a36e88'
SELECT gcc.segment5 Account
,NVL(mc.segment11,'???') PRODECT_CODE
,SUM(moq.transaction_quantity) Trx_Qty
,SUM(ROUND(moq.transaction_quantity * NVL(cic.item_cost,0),2)) Standard_NBV
,SUM(ROUND(moq.transaction_quantity * NVL(cic1.item_cost,0),2)) * :p_conv_rate

<rest of the SQL snipped>

BINDS #18:
bind 0: dty=1 mxl=128(40) mal=00 scl=00 pre=00 oacflg=03 oacfl2=10 size=128 offset=0
bfp=01b16e70 bln=128 avl=03 flg=05
value="558"

<snipped>

WAIT #18: nam='file open' ela= 0 p1=0 p2=0 p3=0
WAIT #18: nam='db file sequential read' ela= 4 p1=432 p2=169056 p3=1
WAIT #18: nam='db file sequential read' ela= 1 p1=72 p2=195769 p3=1
WAIT #18: nam='db file sequential read' ela= 1 p1=73 p2=197743 p3=1
WAIT #18: nam='db file scattered read' ela= 2 p1=65 p2=95737 p3=8
WAIT #18: nam='db file scattered read' ela= 2 p1=65 p2=95745 p3=8
WAIT #18: nam='db file sequential read' ela= 2 p1=73 p2=197755 p3=1
WAIT #18: nam='file open' ela= 0 p1=0 p2=0 p3=0
WAIT #18: nam='db file sequential read' ela= 1 p1=62 p2=190335 p3=1
WAIT #18: nam='file open' ela= 0 p1=0 p2=0 p3=0

<snipped>

FETCH #18:c=510281,e=542530,p=37506,cr=226144503,cu=5,mis=0,r=47,dep=0,og=4,tim=
2464590823
```

While the PARSING IN CURSOR and FETCH summary sections appear in all SQL trace files, the BINDS and WAIT section appear only in the extended SQL trace. Significant additional information can be gleaned from this data. For example, the “db file sequential read” event highlighted above shows a single block numbered **169056** read from database file **432**, with the operation taking 4 centi-seconds (4/100th of a second) to complete. The next highlighted event “db file scattered read” shows that 8 blocks were read, probably as part of a full table scan, from the block 95737 in database file 65 and took 2 centi-seconds to complete. We can then use the following SQL snippet to determine which Object (table or index usually) was involved in this physical I/O:

```
select segment_type, segment_name from dba_extents
where file_id = 432
and 169056 between block_id and block_id + blocks - 1;
```

In fact we determined using this SQL that this was an indexed read of a block in the MTL_ONHAND_QUANTITIES_N5 Index. We can optionally construct similar SQLs from the lines containing ‘db file’ waits using appropriate Operating system file processing tools such as awk or perl. Using these generated SQLs or using hand constructed SQL, we can determine the objects involved, and the amount of I/O to each of these objects, and thus throw more light on what went on behind the scene.

This trace file can and was processed via the Oracle standard TKPROF utility as well. The runtimes for a particular SQL stood out – the values are shown below:

call	count	cpu	elapsed	disk	query	current	rows
Parse	2	0.04	0.06	0	0	0	0
Execute	1	0.01	0.01	0	0	0	0
Fetch	1	5102.81	5425.30	37506	226144503	5	47
total	4	5102.86	5425.37	37506	226144503	5	47

These figures show that the SQL in question took about 1.5 hours (5,425 seconds), which was a majority of the total time taken by the program. The SQL also consumed 5,100 seconds of CPU time, and requested 37,506 disk read requests and performed about 226 million logical I/Os, while fetching just 47 rows! Obviously, this SQL needed to be tuned. Further analysis from the extended trace file showed that 37,196 of these reads were indexed reads against a few tables, all of these using indexed reads. They were the CST_ITEM_COSTS (2.6 Gb), RCV_SHIPMENT_LINES (134 Mb) and the MTL_ONHAND_QUANTITIES_DETAIL (100 Mb) tables, with the majority being indexed reads against the CST_ITEM_COSTS table. You may have noticed that the Fetch line from the TKPROF output above is a processed translation from the “FETCH #18” line of the trace file above.

For further details on using the extended trace in general and within Oracle Applications, read Metalink Notes 39817.1 and 171647.1. The books *Optimizing Oracle Performance* by Cary Millsap and Jeff Holt as well as *The Oracle Wait Interface: A Practical Guide to Performance Diagnostics & Tuning* by Richmond Shee, Kirti Deshpande and K Gopalakrishnan deal with this topic in great detail and are worth reading. Web-based tools such as those available on Hotsos, Inc (www.hotsos.com) and ubTools (www.ubtools.com) can help analyze these extended trace files.

The detailed information from the extended trace file presents us with a quandary: Conventional wisdom states that large tables should be read using an index. However, it seems that this method generates a lot of I/O as well as a substantially large amount of logical I/Os, the latter contributing heavily to the CPU consumption. The core issue is this: An indexed read performs individual reads of the index blocks and using the RowID obtained therein, performs a read of the required data block. The index blocks read include the root block, the branch blocks if present and finally the leaf blocks. Thus, an indexed read performs a minimum of two reads in the case of a zero level index (just the root block, followed by the required data block), and a maximum of (n+2) I/Os where n is the B* Tree level number (DBA_INDEXES.BLEVEL). When a large number of rows are accessed via indexed reads, it is possible that the total number of I/O operations performed ends up being greater than the number of I/O operations that would have otherwise been performed when using an FTS.

In this particular case, the physical I/O wasn't all there was to it—since some of these blocks such as the index root and popular branch/leaf blocks were accessed repeatedly, they were cached, and access to the cached blocks shows up in the counters under the query column as the 226 million-plus count. This operation is a logical I/O (LIO) and costs CPU cycles as the reader process has to acquire a buffer chain latch (an internal locking mechanism) to get to the block in memory. Let us just say that such memory access is not cheap, as the Oracle myths imply, and we see ample evidence in the amount of CPU used! An excellent discourse on FTS versus indexed reads is available on the *SELECT Journal* Web site (www.selectonline.org)—See the articles “To Index or Not To Index—That is the Question” by Daniel Fink and “In Defense of Full-Table Scans” by Jeff Maresh.

The question at hand then is this: Would it be better to access these tables using an FTS depending on whether a larger percentage of rows are to be

selected? When does it make sense to use a FTS rather than indexed reads, and what is the decision point? In other words, when a smaller percentage of rows in a large table are read, it may be wise to use the index and when a larger percentage of rows are to be read, it may be advantageous to use a FTS considering the total I/O that would have otherwise occurred.

The restriction is restated here: The program (and thus the SQL) had to remain the same for the various organization IDs that could be used in the query. Thus, the same SQL should probably use an FTS for certain cases when the percentage of rows required to be read was larger, and indexed reads for the other cases when the percentage of rows in question was smaller.

There are a few options for tuning queries when the code cannot be modified. This includes using the SQL Profiles to “fix” the SQL via hints. However, this option would not have worked as the SQL changes with the types of parameters to the report. In addition, the access path is normally fixed in SQL Profiles, so that route was abandoned. The other option considering was the influencing of the access path via the use of object statistics. This is exactly where the CBO shines when presented with histograms! Simply put, histograms enables the optimizer to calculate the costs of accessing the selected rows via an index or via an FTS for the given key value, based on the relative row count for that value and the expected I/O via both access mechanisms. And this is what we will influence by judicious addition of histograms on selected index columns.

Fortunately in this case, Oracle Applications supports the use of histograms. In fact, it provides for the naming of columns on which histograms are to be collected via the use of special table named FND_HISTOGRAM_COLS in the APPLSYS schema as well as special procedures in the FND_STATS PL/SQL package that can deal with this table. Specifically, the FND_STATS.CHEK_HISTOGRAM_COLS procedure can be used to determine the viability of histograms for the specified tables and the FND_STATS.GATHER_TABLE_STATS procedure to generate object statistics. In fact, the Oracle Applications specific FND_STATS.GATHER_TABLE_STATS procedure is a wrapper that invokes DBMS_STATS.GATHER_TABLE_STATS using appropriate calls that include the generation of histograms for those table columns specified in the FND_HISTOGRAM_COLS table. Without getting into the specifics of Oracle Applications, we can easily describe the algorithm for determining histogram viability for a specific column using the formatted notes from the procedure as shown below:

For a given list of comma separated tables, this procedure checks the data in all the leading columns of all the non-unique indexes of those tables and figures out if histogram needs to be created for those columns. The algorithm is as follows:

```
select decode(floor(sum(tot)/(max(cnt)*75)),0,'YES','NO') HIST
from (select count(col) cnt , count(*) tot from tab sample (S)
where col is not null group by col);
```

The decode says whether or not a single value occupies 1/75th or more of the sample. If sum(cnt) is very small (a small non-null sample), the results may be inaccurate. A count() of at least 3000 is recommended.*

Coming back to the example, we used the following SQL snippet to first generate a list of possible columns of the identified large tables that could use a histogram (lines broken out for readability and number of tables restricted):

```
set serverout on trimspool on echo on linesize 300
spool histograms
execute dbms_output.enable (9999999);
execute fnd_stats.check_histogram_cols('PO.RCV_SHIPMENT_LINES,
INV.MTL_ONHAND_QUANTITIES_DETAIL,BOM.CST_ITEM_COSTS',
factor=>75,percent=>99,degree=>4);
```

This resulted in the following output (formatted for readability):

Table-Name	Column-Name	Histogram	Tot-Count	Max-Count
PO.RCV_SHIPMENT_LINES	DELIVER_TO_PERSON_ID	NO	451976	2718
PO.RCV_SHIPMENT_LINES	EMPLOYEE_ID	YES	451895	105168
PO.RCV_SHIPMENT_LINES	ITEM_ID	YES	451886	28905
PO.RCV_SHIPMENT_LINES	PO_HEADER_ID	NO	452050	4857
PO.RCV_SHIPMENT_LINES	PO_LINE_ID	NO	451929	3068
PO.RCV_SHIPMENT_LINES	PO_LINE_LOCATION_ID	NO	451864	2345
PO.RCV_SHIPMENT_LINES	REQUISITION_LINE_ID	NO	452012	1
INV.MTL_ONHAND_QUANTI	INVENTORY_ITEM_ID	YES	154869	6559
INV.MTL_ONHAND_QUANTI	ORGANIZATION_ID	YES	154877	44876
BOM.CST_ITEM_COSTS	ORGANIZATION_ID	YES	16078150	2304549

It was clear from the above that some of the columns in the tables in question could use histograms. As you can see, it is apparent that if the Max-Count is a large percentage of the Tot-Count it implies that there is at least one value that occurs Max-Count times and this introduces the skew in data values discussed earlier. Specifically, we see that the ORGANIZATION_ID column which we diagnosed as a possible candidate figures prominently in this list.

While we have used this inbuilt procedure to determine the viability of histograms, it is quite simple to construct SQL procedures of your own using this idea to suit your situation. For other examples, see Steve Adams' Web site (www.ixora.com.au/scripts/sql/consider_histogram.sql).

We then generated SQLs to insert selected data into the afore-mentioned FND_HISTOGRAM_COLS table directly for the specified columns as shown below so that subsequent object statistics collections would now include collection of histograms for these additional columns. Obviously, this was first tested on a test instance that was refreshed recently and verified by the business user in a user acceptance environment, prior to being implemented in the production instance!

```
insert into applsys.fnd_histogram_cols values
(702,'CST_ITEM_COSTS','ORGANIZATION_ID','',254,sysdate,1,sysdate,1,'','');
insert into applsys.fnd_histogram_cols values
(401,'MTL_ONHAND_QUANTITIES_DETAIL','INVENTORY_ITEM_ID','',254,sysdate,1,sysdate,1,'','');
insert into applsys.fnd_histogram_cols values
(401,'MTL_ONHAND_QUANTITIES_DETAIL','ORGANIZATION_ID','',254,sysdate,1,sysdate,1,'','');
insert into applsys.fnd_histogram_cols values
(201,'RCV_SHIPMENT_LINES','EMPLOYEE_ID','',254,sysdate,1,sysdate,1,'','');
insert into applsys.fnd_histogram_cols values
(201,'RCV_SHIPMENT_LINES','ITEM_ID','',254,sysdate,1,sysdate,1,'','');
```

The numeric values 201, 401 and 702 denote the application ID, while the value 254 specifies the maximum number of buckets. We could also have used the undocumented FND_STATS.LOAD_HISTOGRAM_COLS procedure to achieve the same result.

Once this was complete, we re-generated object statistics for these selected tables using the Gather Table Statistics concurrent program, choosing to perform this at table level to cater to any changes in data that may have occurred since the previous statistics were collected. This concurrent

program is an inbuilt Oracle Applications program that invokes the FND_STATS.GATHER_TABLE_STATS mentioned earlier. This in turn generates appropriate calls to the DBMS_STATS.GATHER_TABLE_STATS including calls to gather histograms for these new additional columns.

One more item had to be taken care of before we were able to proceed: The report that had this performance issue used bind variables for the organization ID columns used in the WHERE clause and executed on an Oracle 8i database, so the report had to be changed to use lexical parameters. Simply put, the Organization ID that was used as a parameter was converted into a string in the "Before Report" section—this enables the CBO to consider histograms on the said columns. This was a minimal change in the formatting of the SQL rather than a functional or tuning change that could have introduced bugs. Once the report was changed and compiled, we were set to test.

The results were astonishing! The TKPROF section for the SQL in question for the modified report is shown below:

call	count	cpu	elapsed	disk	query	current	rows
Parse	2	0.04	0.02	3	10	0	0
Execute	1	0.00	0.01	0	0	0	0
Fetch	1	30.71	101.28	42423	913573	32	47
total	4	30.75	101.31	42426	913583	32	47

This shows that the elapsed time dropped to 101 seconds, from 5,425 seconds, the CPU time drastically reduced from 5,103 seconds to just 30 seconds. The most impressive change though was in the LIO count—less than a million LIOs as compared to the 226 million LIOs before. Note that although the disk reads actually increased from 37,506 to 42,423, the program performed much faster. From the extended trace file, most of the I/O was for FTS, but as we noted before, we are only worried about elapsed time. The functionality of the report was not changed, which was one of the original objectives.

Further tests selecting smaller organizations tended to use indexed reads so that was not an issue. The original SQL was not changed and proved that we used histograms judiciously in this case!



About the Author

John Kanagaraj is a Principal Consultant with DBSoft Inc., and resides in the Bay Area in sunny California. He has been working with various flavors of UNIX since 1984 and with Oracle since 1988, mostly as a Developer/DBA/Apps DBA and System Administrator. Prior to joining DBSoft, he led small teams of DBAs and UNIX/NT SysAdmins at Shell Petroleum companies in Brunei and Oman. He started his Troubleshooting career as a member (and later became head) of the Database SWAT/Benchmarking team at Wipro Infotech, India. His specialization is UNIX/Oracle Performance management, Backup/recovery and System Availability and he has put out many fires in these areas along the way since 1984! John is also a contributing editor of *SELECT Journal* and can be reached via email at ora_apps_dba_y@yahoo.com.