SQL TUNING WITH STATISTICS

Wolfgang Breitling, Centrex Consulting Corporation

This paper looks at the DBMS_STATS package and how it can be used – beyond just the gathering of statistics – in the tuning effort, particularly where the SQL may not be changed for technical or license reasons. After introducing the DBMS_STATS package, the paper demonstrates how it can be used to alter the access path – and the performance – of a SQL without touching the SQL.

The findings presented are based on experience and tests with Oracle 8*i* (8.1.7) on Windows 2000, Linux Redhat 7.2, HP-UX 11.0, and Compaq Tru64 5.1. Comments on changes in Oracle 9*i* are based on Oracle 9.2.0 on Windows 2000 and Linux Redhat 7.2.

THE DBMS_STATS PACKAGE

Gathering Statistics

There are four procedures to gather statistics on objects in the database:

- GATHER_DATABASE_STATS, GATHER_SCHEMA_STATS
  gathers statistics for all objects in the database or in a schema. There are many options for special processing, e.g. finding all objects without, or with stale statistics.
- GATHER_TABLE_STATS
  gathers table and column statistics. It can gather statistics for individual partitions; the default is to gather global table statistics and individual statistics for all partitions. Gather_table_stats is also the procedure to gather histograms. Unlike analyze, gather_table_stats can do much of the statistics gathering in parallel, but there are limitations. Consult the manual.
  After gathering table and column stats, gather_table_stats can also gather statistics on all indexes of the table, but unlike analyze does not do so by default (cascade => { `true` | `false` })
- GATHER_INDEX_STATS
  gathers index statistics. It does not execute in parallel. As with gather_table_stats, unless a partition name is specified, global statistics and statistics for all individual partitions are gathered for a partitioned index.

All gathering procedures have the option to save the current statistics in a user statistics table before overwriting them.

A fifth procedure is new with Oracle 9*i*:

- GATHER_SYSTEM_STATS
  gathers system statistics. The current values can be retrieved with GET_SYSTEM_STATS or viewed by querying sys.stats_aux$

sreadtim	average time to read single block (random read), in milliseconds
mreadtim	average time to read an mbrc block at once (sequential read), in milliseconds
cpuspeed	average number of CPU cycles per second, in millions
mbrc	average multiblock read count for sequential read, in blocks
maxthr	maximum I/O system throughput, in bytes/sec
slavethr	average slave I/O throughput, in bytes/sec

Unlike the database object statistics gathering, gathering system statistics does not invalidate any SQL in the SGA and cause it to be reparsed. However, Oracle9*i* now has a parameter in all the object statistics gathering procedures to not invalidate SQL either (no_invalidate => { `true` | `false` })

There are corresponding DELETE_XXX_STATS procedures plus an additional DELETE_COLUMN_STATS procedure. With DBMS_STATS it is now possible to delete the statistics for individual columns.

Getting or Setting Statistics

There are matching procedures to retrieve and set statistics for individual tables, indexes, and columns, as well as retrieving and setting system statistics in Oracle 9*i*.

USES OF DBMS_STATS BEYOND ANALYZE

Transferring Statistics

The DBMS_STATS package offers the ability to copy statistics between the dictionary and a user statistics table: EXPORT_*_STATS copies the statistics from the dictionary to the user table and IMPORT_*_STATS does the reverse.

The "stattab" Table

The user statistics table must be built with the CREATE_STAT_TABLE procedure. There is also a DROP_STAT_TABLE procedure which may be useful in a stored procedure, otherwise the table may just be dropped with a "drop table …" statement.

These are the columns of the stattab table:

Name	Type	Name	Type
STATID	VARCHAR2(30)	N1	NUMBER
TYPE	CHAR(1)	N2	NUMBER
VERSION	NUMBER	N3	NUMBER
FLAGS	NUMBER	N4	NUMBER
D1	DATE	N5	NUMBER
CH1	VARCHAR2(1000)	N6	NUMBER
C1	VARCHAR2(30)	N7	NUMBER
C2	VARCHAR2(30)	N8	NUMBER
C3	VARCHAR2(30)	N9	NUMBER
C4	VARCHAR2(30)	N10	NUMBER
C5	VARCHAR2(30)	N11	NUMBER
		N12	NUMBER
		R1	RAW(32)
		R2	RAW(32)

In order to be able to use the stattab table to modify statistics, it is necessary to understand what the columns mean. The following mapping is based on observation. Oracle does not document it: "The columns and types that compose this table are not relevant as it should be accessed solely through the procedures in this package".

STATID is the only documented column and is the user settable identifier. CH1 is currently unused and VERSION is always 4, even for Oracle 9*i*. The meaning of the FLAGS bits is unknown, but bitand(flags,2) seems to be on if the object had user_stats..

Except for the new system statistics, C5 is the owner and D1 is the LAST_ANALYZED date. TYPE identifies the type of object and the meaning of the other columns depends on the TYPE:

TYPE 'T'		TYPE 'I'		TYPE 'C'		TYPE 'C' histogram	
C5	OWNER	C5	OWNER	C5	OWNER	C5	OWNER
C1	TABLE_NAME	C1	INDEX_NAME	C1	TABLE_NAME	C1	TABLE_NAME
C2	PARTITION_NAME	C2	PARTITION_NAME	C2	PARTITION_NAME	C2	PARTITION_NAME
C3	SUBPARTITION_NAME	C3	SUBPARTITION_NAME	C3	SUBPARTITION_NAME	C3	SUBPARTITION_NAME
				C4	COLUMN_NAME	C4	COLUMN_NAME
N1	NUM_ROWS	N1	NUM_ROWS				
N2	BLOCKS	N2	LEAF_BLOCKS	N1	NUM_DISTINCT	N10	ENDPOINT_NUMBER
N3	AVG_ROW_LEN	N3	DISTINCT_KEYS	N2	DENSITY	N11	ENDPOINT_VALUE
N4	SAMPLE_SIZE	N4	LEAF_BLOCKS_PER_KEY	N4	SAMPLE_SIZE		
		N5	DATA_BLOCKS_PER_KEY	N5	NUM_NULLS		
		N6	CLUSTERING_FACTOR	N6	LO_VALUE		
		N7	BLEVEL	N7	HI_VALUE		
		N8	SAMPLE_SIZE	N8	AVG_COL_LEN		

The appendix contains view definitions on the user statistics table analogous to the DBA_xxx dictionary views.

Backup and Rollback of Statistics Changes

The user statistics table and the EXPORT_* and IMPORT_* procedures make it possible to backup statistics; and restore them if new statistics have undesirable effects on the optimizer's plan choices.

The user statistics table and the EXPORT_* and IMPORT_* procedures also open new possibilities to work with statistics. Since the user statistics table is an ordinary table, it can be exported and imported into another database and then IMPORTed into that database's dictionary to make it "look" to the optimizer like the original database. SQL statement plans can thus be analyzed in a test database with less – or even no – data.

Another possibility is to actively manipulate the statistics to affect the optimizer's access plans. Of course, this can only work if the cost based optimizer is being used.

Phases of SQL Tuning

In the following we focus strictly on SQL tuning – improving the performance of a SQL statement, as opposed to application tuning which takes into account the entire process, finding a way to achieve the desired business result through different, more efficient means, possibly discovering that the particular problem statement or the entire process is unnecessary. This narrow tuning focus breaks down into three phases:

1. Identify poorly performing SQL
2. Identify alternate – i.e. better – access plan
3. Persuade CBO to use this access plan

We will next look at a way to use DBMS_STATS in the third phase, particularly in cases where the SQL can not be modified.

Means to change an Access Plan

There are a number of ways to make the cost based optimizer choose a different the access plan. Listed by increasing global impact:

- Change the statement
- Use hint
  These two have only local impact, isolated to the changed statement. Obviously, both require the ability to change the statement.
- Change statistics
  all SQL that use or reference the component whose statistics are changed may be impacted. The big question, of course, is: "How does one know which other SQL may be affected"? A possible technique is "Explain Plan Analysis"[1]
- Create or drop an index
  all SQL that use the table may be impacted. Again, how does one know what those are? The same "Explain Plan Analysis" addresses that and there are tools on the market which do that.
- Change initialization parameters
  This is the most global change since all SQL may be impacted. Unless the parameter can be changed for just a session, e.g. a batch job. Again, the "Explain Plan Analysis" aims at finding the SQL that are affected.

As the risk of changing the access paths of other SQL statements increases, so does the potential reward – rather than tuning SQL by SQL, many statements could be improved with a single change.

We will next look at an example that uses option 3 "change statistics" to improve the performance of a SQL statement without modifying the statement itself.

1 An idea by the author waiting to be realized.

USING DBMS_STATS TO CHANGE AN ACCESS PLAN

Statistics affecting the CBO

In order to effectively use statistics in SQL tuning, one needs to know which statistics affect the CBO and how. The following are the statistics used by the cost based optimizer in deciding on an access plan.

Table
num_rows, blocks, avg_row_len

Column
num_distinct, density, num_nulls, low_value, high_value
with histograms: buckets, endpoint_number, endpoint_value

Index
blevel, leaf_blocks, distinct_keys, avg_leaf_blocks_per_key, avg_data_blocks_per_key, clustering_factor

Rather than setting statistics directly, which would be possible with the dbms_stats.set_xxx_stats procedures, I recommend exporting the statistics, changing the value(s) in the stattab table and then re-importing the changed statistics back into the dictionary. Setting statistics directly sets the flag USER_STATS to 'YES'. It is unknown if or how that affects the optimizer.

Regardless of which method is used to alter statistics, ALWAYS, ALWAYS back up the current statistics into a stattab table so that they can be restored if the change has undesirable effects:

```
DBMS_STATS.EXPORT_XXX_STATS (
  ownname => 'abc',
  …,
  stattab => 'stats_table',
  statid => 'bkup');
```

PUTTING IT TO WORK

The sql

This is the SQL we'll be using to demonstrate the technique – and the effect of tuning with statistics:

```
SELECT A.ACCOUNT, SUM(A.POSTED_TOTAL_AMT)
from PS_PCR_LEDSUM_OP A
  , PSTREESELECT06 L1
  , PSTREESELECT10 L
where A.LEDGER = 'XXXXXX'
  and A.FISCAL_YEAR = 1997
  and A.ACCOUNTING_PERIOD BETWEEN 1 and 12
  and A.CURRENCY_CD IN (' ', 'CAD')
  and A.STATISTICS_CODE = ' '
  and A.PCR_TREENODE_DEPT = 'YYYYYY'
  and L1.SELECTOR_NUM = 15101
  and A.ACCOUNT = L1.RANGE_FROM_06
  and (L1.TREE_NODE_NUM BETWEEN 1968278454 and 1968301256
    OR L1.TREE_NODE_NUM BETWEEN 1968301263 and 1968301270
    OR L1.TREE_NODE_NUM BETWEEN 1968867729 and 196888696
    OR L1.TREE_NODE_NUM BETWEEN 1969156312 and 1969207615)
  and L.SELECTOR_NUM = 15109
  and A.DEPTID = L.RANGE_FROM_10
  and L.TREE_NODE_NUM BETWEEN 1692307684 and 1794871785
group by A.ACCOUNT
```

The "Before"

And this is the corresponding explain plan:

```
  cost     card operation
   130        1 SELECT STATEMENT
   130        1  SORT GROUP BY
   128        1   NESTED LOOPS
   125        1    NESTED LOOPS
    50        1     INDEX RANGE SCAN PSAPSTREESELECT06
    75    1,753     TABLE ACCESS BY LOCAL INDEX ROWID PS_PCR_LEDSUM_OP
                        PARTITION: START=5 STOP=5
     2    1,753      INDEX RANGE SCAN PS_PCR_LEDSUM_OP_ACC PARTITION: START=5 STOP=5
     3      456    INDEX RANGE SCAN PSBPSTREESELECT10
```

From the plan alone one can not tell if it is good or bad. There is just no way. Do not let anyone tell you otherwise.

The only way is to test it. Following are the execution statistics for five consecutive calls – to eliminate the effects of blocks being, or not being, in the buffer pool. A sql trace was enabled to get the row counts.

hash	call	count	cpu	elapsed	disk	query	current	rows
3280272888	Parse	1	0.15	0.17[?]	0	0	0	0
3280272888	Exec	1	0	0	0	0	0	0
3280272888	Fetch	4	2.03	19.54	15162	22909	0	34
3280272888	total	6	2.18	19.71	15162	22909	0	34
3280272888	Parse	1	0	0	0	0	0	0
3280272888	Exec	1	0	0	0	0	0	0
3280272888	Fetch	4	1.19	1.21	12310	22909	0	34
3280272888	total	6	1.19	1.21	12310	22909	0	34
3280272888	Parse	1	0	0	0	0	0	0
3280272888	Exec	1	0	0	0	0	0	0
3280272888	Fetch	4	1.14	1.19	11413	22909	0	34
3280272888	total	6	1.14	1.19	11413	22909	0	34
3280272888	Parse	1	0	0	0	0	0	0
3280272888	Exec	1	0	0	0	0	0	0
3280272888	Fetch	4	1.08	1.08	10957	22909	0	34
3280272888	total	6	1.08	1.08	10957	22909	0	34
3280272888	Parse	1	0	0	0	0	0	0
3280272888	Exec	1	0	0	0	0	0	0
3280272888	Fetch	4	1.03	1.03	10098	22909	0	34
3280272888	total	6	1.03	1.03	10098	22909	0	34

[?] The parse times in the 1st call are so high because the 10053 trace event was enabled, capturing the CBO trace.

When contrasting the cardinality estimates from the explain plan to the actual row counts, the problem with the plan becomes apparent: the row count of 1125 instead of the estimated cardinality of 1 of the join of PSTREESELECT06 and PS_PCR_LEDSUM_OP which is then used as the driving outer "table" of the next NL join.
The index range scan of PSBPSTREESELECT10 is then done 1125 times rather than the estimated once.

```
 card     Rows  Execution Plan
    1        0  SELECT STATEMENT   GOAL: CHOOSE
    1       34   SORT GROUP BY
    1      892    NESTED LOOPS
    1    1,125     NESTED LOOPS
    1      151      INDEX RANGE SCAN PSAPSTREESELECT06
1,753     1274      TABLE ACCESS BY LOCAL INDEX ROWID PS_PCR_LEDSUM_OP
1,753                  PARTITION: START=5 STOP=5
  456   31,538       INDEX RANGE SCAN PS_PCR_LEDSUM_OP_ACC
                       PARTITION: START=5 STOP=5
          892     INDEX RANGE SCAN PSBPSTREESELECT10
```

Going one line down, we realize that the join cardinality of 1 is largely the result of the estimated cardinality of 1 of table PSTREESELECT06. The two relevant lines out of the 10053 CBO trace are included:

```
Join cardinality:  1 = outer (1) * inner (1753) * sel (7.0522e-04)
TABLE: PSTREESELECT06     ORIG CDN: 154506  CMPTD CDN: 1
```

If this sounds too easy then it is because hindsight is 20/20. The actual analysis took a bit longer and involved a more extensive analysis of the 10053 trace.

The Process

In order to correct, i.e. increase, the optimizer's cardinality estimate for the PSTREESELECT06 table we need to lower the selectivity of one or more of the columns used in the predicate. A way to do that is to modify the density statistic of the column. The two predicates on table PSTREESELECT06 are SELECTOR_NUM and TREE_NODE_NUM. SELECTOR_NUM is used in an equal predicate and since there is a histogram on it, the optimizer will use the histogram data to rather accurately calculate the selectivity of the predicate. TREE_NODE_NUM on the other hand is used in "or"ed range scans and here the density comes into play for determining the predicate selectivity. Since we decided the cardinality was too low, the selectivity is too low as well (estimated cardinality = selectivity * base cardinality)[2]. We therefore elect to increase the density by a factor of 10 and then re-import the changed statistics into the dictionary:

Before we do anything, however, we make a backup of the statistics.

```
DBMS_STATS.EXPORT_TABLE_STATS (
ownname => SYS_CONTEXT ('USERENV', 'CURRENT_SCHEMA'),
tabname => 'PSTREESELECT06',
 stattab => 'STATS_TABLE',
statid => 'BKUP',
cascade => TRUE);
```

[2] There is an apparent paradox here which stems from a dual use of the word selectivity. The selectivity value – another, better, term is filter factor – is a number between 0 and 1and a small selectivity VALUE corresponds to a high SELECTIVITY of the predicate, i.e. selecting FEW rows. It is similar with the word performance: if you make something faster, did you increase or decrease the performance?

Then we make the working copy

```
DBMS_STATS.EXPORT_TABLE_STATS (
ownname => SYS_CONTEXT ('USERENV', 'CURRENT_SCHEMA'),
tabname => 'PSTREESELECT06',
 stattab => 'STATS_TABLE',
statid => 'WORK',
cascade => TRUE);
```

table	column	NDV	density	bkts
PSTREESELECT06	SELECTOR_NUM	158	5.8728E-02	5
PSTREESELECT06	TREE_NODE_NUM	2,619	5.5647E-03	5
PSTREESELECT06	RANGE_FROM_06	10,152	9.8503E-05	1
PSTREESELECT06	RANGE_TO_06	10,152	9.8503E-05	1

```
update stats_tab_columns[3]
set density = 10*density
where statid = 'WORK'
  and table_name = 'PSTREESELECT06'
  and column_name = 'TREE_NODE_NUM';
```

After changing the density value we copy the statistics back into the dictionary.

```
DBMS_STATS.IMPORT_TABLE_STATS (
ownname => SYS_CONTEXT ('USERENV', 'CURRENT_SCHEMA'),
tabname => 'PSTREESELECT06',
stattab => 'STATS_TABLE',
statid => 'WORK',
cascade => TRUE);
```

The column statistics report shows the changed statistics

table	column	NDV	density	bkts
PSTREESELECT06	SELECTOR_NUM	158	5.8728E-02	5
PSTREESELECT06	TREE_NODE_NUM	2,619	**5.5647E-02**	5
PSTREESELECT06	RANGE_FROM_06	10,152	9.8503E-05	1
PSTREESELECT06	RANGE_TO_06	10,152	9.8503E-05	1

The "After"

And this is the explain plan after the change in density of PSTREESELECT06.TREE_NODE_NUM.

cost	card	operation
1,215	64	SELECT STATEMENT
1,215	64	SORT GROUP BY
1,209	64	HASH JOIN
4	273	INDEX RANGE SCAN PSAPSTREESELECT10
1,204	359	HASH JOIN
50	290	INDEX RANGE SCAN PSAPSTREESELECT06
1,153	1,753	TABLE ACCESS BY LOCAL INDEX ROWID PS_PCR_LEDSUM_OP PARTITION: START=5 STOP=5
59	1,753	INDEX RANGE SCAN PS_PCR_LEDSUM_OP_TDEP PARTITION: START=5 STOP=5

The plan DID change as a result of the statistics change even though the base access paths are the same as before:

[3] See the appendix for the view definition

- an index range scan on the (1997) partition of PS_PCR_LEDSUM_OP – although using a different index
- an index range scan of PSTREESELECT10 – again using a different index now
- an index range scan of PSTREESELECT06, the table where we changed a column density, using the same index as before.

The big difference are the two hash joins in place of the nested loop joins, predicated by the higher join cardinalities. The estimated cardinality of PSTREESELECT06, and the resulting join cardinality estimates from the 10053 trace show that:

```
TABLE: PSTREESELECT06     ORIG CDN: 154506  CMPTD CDN: 290Join cardinality:  359
= outer (290) * inner (1753) * sel (7.0522e-04)
Join cardinality:  64 = outer (359) * inner (273) * sel (6.5703e-04)
```

But again, just from looking at the plan one can not tell if it is better. The higher cost (1215 vs. 130) ought to indicate that it is worse. The only way to find out is to execute the SQL. Again we use 5 consecutive runs to eliminate buffer pool issues:

hash	call	count	cpu	elapsed	disk	query	current	rows
3280272888	Parse	1	0.23	0.24	0	0	0	0
3280272888	Exec	1	0	0	0	0	0	0
3280272888	Fetch	4	0.35	0.51	248	655	0	34
3280272888	total	6	0.58	0.75	248	655	0	34
3280272888	Parse	1	0	0	0	0	0	0
3280272888	Exec	1	0	0	0	0	0	0
3280272888	Fetch	4	0.18	0.19	29	655	0	34
3280272888	total	6	0.18	0.19	29	655	0	34
3280272888	Parse	1	0	0	0	0	0	0
3280272888	Exec	1	0	0.01	0	0	0	0
3280272888	Fetch	4	0.19	0.19	0	655	0	34
3280272888	total	6	0.19	0.2	0	655	0	34
3280272888	Parse	1	0	0	0	0	0	0
3280272888	Exec	1	0.01	0	0	0	0	0
3280272888	Fetch	4	0.18	0.2	0	655	0	34
3280272888	total	6	0.19	0.2	0	655	0	34
3280272888	Parse	1	0	0	0	0	0	0
3280272888	Exec	1	0	0.01	0	0	0	0
3280272888	Fetch	4	0.19	0.19	0	655	0	34
3280272888	total	6	0.19	0.2	0	655	0	34

The first execution may have benefited from the 5 prior executions when we compare it to the 1st execution of the old plan. But the 2nd through 5th executions are also consistently a second faster than those using the previous plan. Note that the hash_value of the SQL did not change – we changed the statistics, not the SQL.

If we compare further we see a drop in query (consistent) reads: 655 compared to the 22909 of the prior plan and also a big drop in physical reads: the 3rd through 5th SQL did not incur any physical IO.

It should be mentioned that these tests ran in a single user mode – nothing else was running in the database at the time. But other processes were running on the server – backups for other databases for example.

For completeness, let us compare the estimated cardinalities of the new plan with the actual row counts:

```
   card     Rows  Execution Plan
-------  -------  ----------------------------------------------------------------
     64        0  SELECT STATEMENT   GOAL: CHOOSE
     64       34  SORT GROUP BY
     64      892   HASH JOIN
    273      150    INDEX RANGE SCAN  PSAPSTREESELECT06
    359     8752    HASH JOIN
    290     9237     INDEX RANGE SCAN  PSAPSTREESELECT10
  1,753    10524     TABLE ACCESS BY LOCAL INDEX ROWID PS_PCR_LEDSUM_OP
  1,753                    PARTITION: START=5 STOP=5
           12842      INDEX RANGE SCAN  PS_PCR_LEDSUM_OP_TDEP PARTITION: START=5 STOP=5
```

We can see that there are still sizeable numerical differences, but the relative differences are much smaller now than before.

What does that mean? It means that with the modified statistics, the optimizer's cardinality estimates are closer to reality. The optimizer does an excellent job in choosing a good access path – provided its estimates are accurate. Contrary to popular belief and widespread practice, constantly updated statistics do not guarantee accurate estimates[4]. Sometimes you need to give the optimizer a helping hand – be it through a hint or, as in this case, through a statistics "white lie".

APPENDIX

STATS_xxx Views

```
create or replace view STATS_TABLES (    as
                STATID                   select statid
               ,OWNER                    ,c5
               ,TABLE_NAME               ,c1
               ,NUM_ROWS                 ,n1
               ,BLOCKS                   ,n2
               ,AVG_ROW_LEN              ,n3
               ,SAMPLE_SIZE              ,n4
               ,LAST_ANALYZED)           ,d1
grant select on STATS_TABLES to public;  from stats_table where type='T' and c2 is null
                                         and c5 = SYS_CONTEXT ('USERENV','CURRENT_SCHEMA');


create or replace view STATS_INDEXES (   as
                STATID                   select statid
               ,OWNER                    ,c5
               ,INDEX_NAME               ,c1
               ,BLEVEL                   ,n7
               ,LEAF_BLOCKS              ,n2
               ,DISTINCT_KEYS            ,n3
               ,AVG_LEAF_BLOCKS_PER_KEY  ,n4
               ,AVG_DATA_BLOCKS_PER_KEY  ,n5
               ,CLUSTERING_FACTOR        ,n6
               ,NUM_ROWS                 ,n1
               ,SAMPLE_SIZE              ,n8
               ,LAST_ANALYZED)           ,d1
                                         from stats_table where type='I' and c2 is null
grant select on STATS_INDEXES to public; and c5 = SYS_CONTEXT ('USERENV','CURRENT_SCHEMA');
```

[4] See [2] for an explanation for some of the reasons why.

```
create or replace view STATS_TAB_COLUMNS (
                STATID
               ,OWNER
               ,TABLE_NAME
               ,COLUMN_NAME
               ,NUM_DISTINCT
               ,DENSITY
               ,N3
               ,LOW_VALUE
               ,HIGH_VALUE
               ,NUM_NULLS
               ,AVG_COL_LEN
               ,SAMPLE_SIZE
               ,LAST_ANALYZED)
as
select statid
      ,c5
      ,c1
      ,c4
      ,n1
      ,n2
      ,n3
      ,n6
      ,n7
      ,n5
      ,n8
      ,n4
      ,d1
from stats_table where type='C' and c2 is null
and c5 = SYS_CONTEXT ('USERENV','CURRENT_SCHEMA');

grant select on STATS_TAB_COLUMNS to public;

create or replace view STATS_TAB_PARTITIONS (
                statid
               ,OWNER
               ,TABLE_NAME
               ,PARTITION_NAME
               ,SUBPARTITION_NAME
               ,NUM_ROWS
               ,BLOCKS
               ,AVG_ROW_LEN
               ,SAMPLE_SIZE
               ,LAST_ANALYZED)
as
select statid
      ,c5
      ,c1
      ,c2
      ,c3
      ,n1
      ,n2
      ,n3
      ,n4
      ,d1
from stats_table where type='T' and c2 is not null
and c5 = SYS_CONTEXT ('USERENV','CURRENT_SCHEMA');

grant select on STATS_TAB_PARTITIONS to public;

create or replace view STATS_IND_PARTITIONS (
                STATID
               ,OWNER
               ,INDEX_NAME
               ,PARTITION_NAME
               ,SUBPARTITION_NAME
               ,BLEVEL
               ,LEAF_BLOCKS
               ,DISTINCT_KEYS
               ,AVG_LEAF_BLOCKS_PER_KEY
               ,AVG_DATA_BLOCKS_PER_KEY
               ,CLUSTERING_FACTOR
               ,NUM_ROWS
               ,SAMPLE_SIZE
               ,LAST_ANALYZED)
as
select statid
      ,c5
      ,c1
      ,c2
      ,c3
      ,n7
      ,n2
      ,n3
      ,n4
      ,n5
      ,n6
      ,n1
      ,n8
      ,d1
from stats_table where type='I' and c2 is not null;
and c5 = SYS_CONTEXT ('USERENV','CURRENT_SCHEMA')

grant select on STATS_IND_PARTITIONS to public;
```

```
create or replace view STATS_PART_COL_STATISTICS (          as
                                              STATID        select statid
                                             ,OWNER         ,c5
                                             ,TABLE_NAME    ,c1
                                             ,PARTITION_NAME ,c2
                                             ,SUBPARTITION_NAME ,c3
                                             ,COLUMN_NAME   ,c4
                                             ,NUM_DISTINCT  ,n1
                                             ,DENSITY       ,n2
                                             ,N3            ,n3
                                             ,LOW_VALUE     ,n6
                                             ,HIGH_VALUE    ,n7
                                             ,NUM_NULLS     ,n5
                                             ,AVG_COL_LEN   ,n8
                                             ,SAMPLE_SIZE   ,n4
                                             ,LAST_ANALYZED) ,d1
                                                            from stats_table where type='C' and c2 is not null;
grant select on STATS_PART_COL_STATISTICS to public;        and c5 = SYS_CONTEXT ('USERENV','CURRENT_SCHEMA')
```

REFERENCES

[1] Wolfgang Breitling. *A Look under the Hood of CBO - the 10053 Event*. in Proceedings of the 2003 Hotsos Symposium on Oracle® System Performance. Feb 9–12, 2003. Dallas, TX. (http://www.centrexcc.com)

[2] Wolfgang Breitling. *Fallacies of the Cost Based Optimizer*. in Proceedings of the 2003 Hotsos Symposium on Oracle® System Performance. Feb 9–12, 2003. Dallas, TX. (http://www.centrexcc.com)

Metalink Notes:

114671.1 Gathering Statistics for the Cost Based Optimizer
130899.1 How to Set User-Defined Statistics Instead of RDBMS Statistics
122009.1 How to Retrieve Statistics Generated by ANALYZE SQL Statement
130688.1 Report Statistics for a Table, it's columns and it's indexes with DBMS_STATS
130911.1 How to Determine if Dictionary Statistics are RDBMS-Generated or User-Defined
102334.1 How to automate ANALYZE TABLE when changes occur on tables
1074354.6 DBMS_STATS.CREATE_STAT_TABLE: What Do Table Columns Mean?
117203.1 How to Use DBMS_STATS to Move Statistics to a Different Database
149560.1 Collect and Display System Statistics (CPU and IO) for CBO usage
153761.1 Scaling the system to improve CBO optimizer