# Tips and Techniques for Statistics Gathering

**Arup Nanda**

*Longtime Oracle DBA*

# Agenda

- High Level
  - Pending stats
  - Correlated Stats
  - Sampling

# Reporting

- New reporting function for auto stats collection
- Returns the report in CLOB

```
SQL> var ret clob
SQL> set long 999999
SQL> exec :ret :=
dbms_stats.report_stats_operations;

PL/SQL procedure successfully completed.

SQL> print ret
```

rep.sql

# Lowdown on Stats

- Optimizer Statistics on tables and indexes are vital for the optimizer to compute optimal execution plans
- In many cases you gather stats with estimate
- Without accurate stats, the optimizer may decide on a sub-optimal execution plan
- When stats change, the optimizer may change the plan
- Truth: stats affect the plan, but not necessarily positively

# Data: Value *vs* Pattern

| State | Customers | %age |
|---|---|---|
| CT | 1,000 | 10% |
| NY | 5,000 | 50% |
| CA | 4,000 | 40% |

After some days

| State | Customers | %age |
|---|---|---|
| CT | 2,000 | 10% |
| NY | 10,000 | 50% |
| CA | 8,000 | 40% |

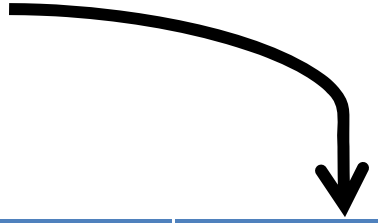
***Important***

The data itself changed; but the pattern did not. The new stats will not change the execution path, and therefore probably not needed

# Case 2

| State | Customers | %age |
|---|---|---|
| CT | 1,000 | 10% |
| NY | 5,000 | 50% |
| CA | 4,000 | 40% |

After some days



| State | Customers | %age |
|---|---|---|
| CT | 2,500 | 12.5% |
| NY | 10,500 | 52.5% |
| CA | 7,000 | 35.0% |

***Important***

The pattern is different; but still close to the original pattern. *Most* queries should perform well with the original execution plan.

# Naked Truth

- Stats can actually create performance issues
- Example
  - A query plan had nested loop as a path
  - Data changed in the underlying tables
  - But the pattern did not change much
  - So, NL was still the best path
  - Stats were collected
  - Optimizer detected the subtle change in data pattern and changed to hash joins
  - Disaster!

# The problem with new stats

- The CBO does not now what is close *enough*
  - For it, 50.0% and 52.5% are *different* values
- The internal logic of the CBO may determine a different plan due to this *subtle* change
- This new plan may be better, or worse
  - This is why many experts recommend not collecting stats when database performance is acceptable

# What's the Solution?

- If only you could predict the effect of new stats before the CBO uses them
  - and make CBO use them if there are no untoward issues
- Other Option
  - You can collect stats in a different database
  - Test in that database
  - If everything looks ok, you can export the stats from there and import into production database
- The other option is not a very good one
  - The test database may not have the same distribution
  - It may not have the same workload

# Pending Stats

- Answer: **Pending Statistics**
- In short
  - DBA collects stats as usual
  - But the CBO does not see these new stats
  - DBA examines the effects of the stats on queries of a session where these new stats are active
  - If all look well, he can "publish" these stats
  - Otherwise, he discards them

# How to Make Stats "Pending"

- It's the property of the table (or index)
- Set it by a packaged procedure
  `DBMS_STATS.SET_TABLE_PREFS`
- Example:

```
begin
   dbms_stats.set_table_prefs (
      ownname => 'ARUP',
      tabname => 'SALES',
      pname   => 'PUBLISH',
      pvalue  => 'FALSE'
   );
end;
```

prefs_false.sql
sales_stats.sql_

- After this, the stats collected will be *pending*

# Table Preferences

- The procedure is not new. Used before to set the default properties for stats collection on a table.
  - e.g. to set the default degree of stats collection on the table to 4:

```
dbms_stats.set_table_prefs (
   ownname => 'ARUP',
   tabname => 'SALES',
   pname   => 'DEGREE',
   pvalue  => 4
 );
```

# Stats after “Pending”

- When the table property of stats “PUBLISH” is set to “”FALSE”
- The stats are not visible to the Optimizer
- The stats will not be updated on USER_TABLES view either:

```
select to_char(last_analyzed,'mm/dd/yy
  hh24:mi:ss')
from user_tables
where table_name = 'SALES';

TO_CHAR(LAST_ANAL
-----------------
09/10/07 22:04:37
```

la.sql_

# Visibility of Pending Stats

- The stats will be visible on a new view
  `USER_TAB_PENDING_STATS`

```
select to_char(last_analyzed,'mm/dd/yy
  hh24:mi:ss')
from user_tab_pending_stats
where table_name = 'SALES';

TO_CHAR(LAST_ANAL
-----------------
09/21/07 11:03:35
```

pending.sql_

# Checking the Effect

- Set a special parameter in the session

```
alter session set
  optimizer_use_pending_statistics = true;
```

- After this setting, the CBO will consider the new stats *in that session only*
- You can even create and index and collect the pending stats on the presence of the index
- To check if the index would make any sense

alter_true.sql_

# Publishing Stats

- Once satisfied, you can make the stats visible to optimizer

```
begin
    dbms_stats.publish_pending_stats
        ('ARUP', 'SALES');
end;
```

- Now the USER_TABLES will show the correct stats
- Optimizer will use the newly collected stats
- Pending Stats will be deleted

publish.sql_

# New Stats make it Worse?

- Simply delete them

```
begin
   dbms_stats.delete_pending_stats
  ('ARUP',‘SALES');
end;
```

- The pending stats will be deleted
- You will not be able to publish them

# Checking for Preferences

- You can check for the preference for publishing stats on the table SALES:

```
select dbms_stats.get_prefs ('PUBLISH','ARUP','SALES') from dual;

DBMS_STATS.GET_PREFS('PUBLISH','ARUP','SALES')
-----------------------------------------------
FALSE
```

- Or, here is another way, with the change time:

```
select pname, valchar, valnum, chgtime
from optstat_user_prefs$
where obj# = (select object_id from dba_objects
where object_name = 'SALES' and owner = 'ARUP')

PNAME      VALCHAR CHGTIME
---------- ------- ---------------------------------
PUBLISH    TRUE    02-MAR-10 01.38.56.362783 PM -05:00
```

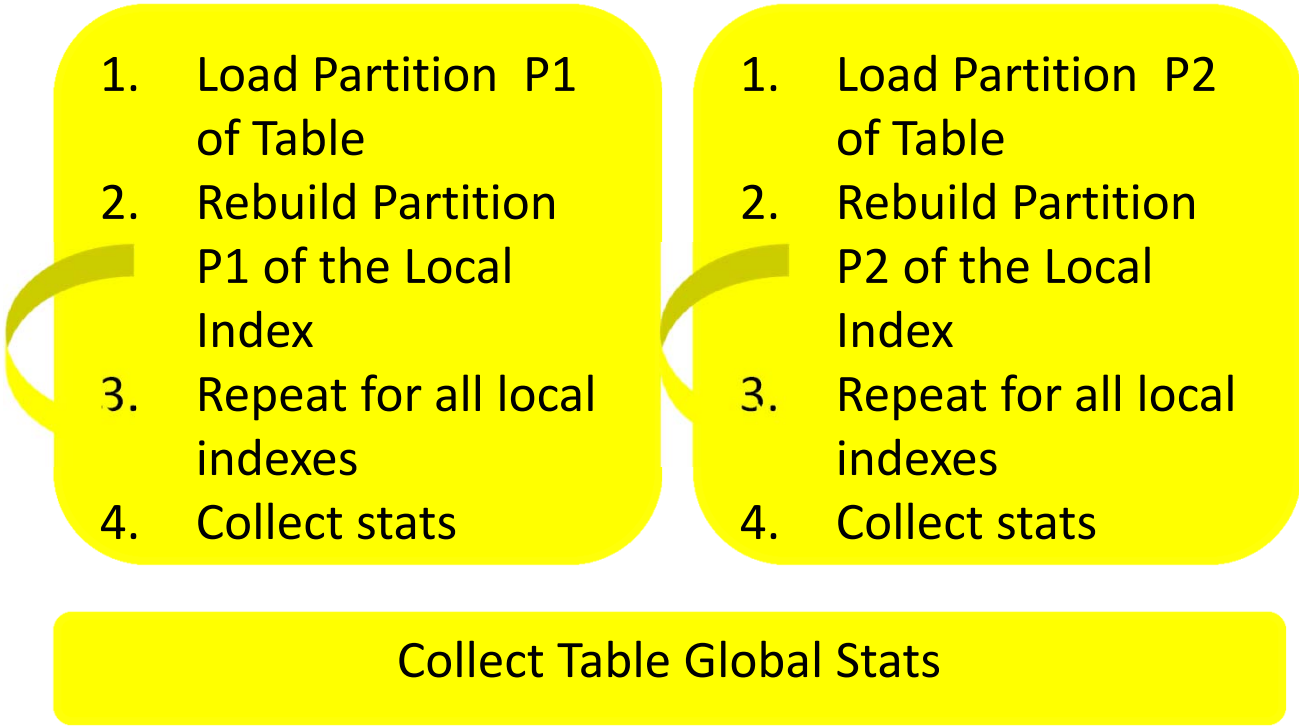
# Other Preferences

- The table property is now set to FALSE
- You can set the default stats gathering of a whole schema to pending

```
begin
    dbms_stats.set_schema_prefs (
        ownname => 'ARUP',
        pname   => 'PUBLISH',
        pvalue  => 'FALSE');
end;
```

- You can set it for the whole database as well
  - dbms_stats.set_database_prefs

# Loading of Partitioned Tables

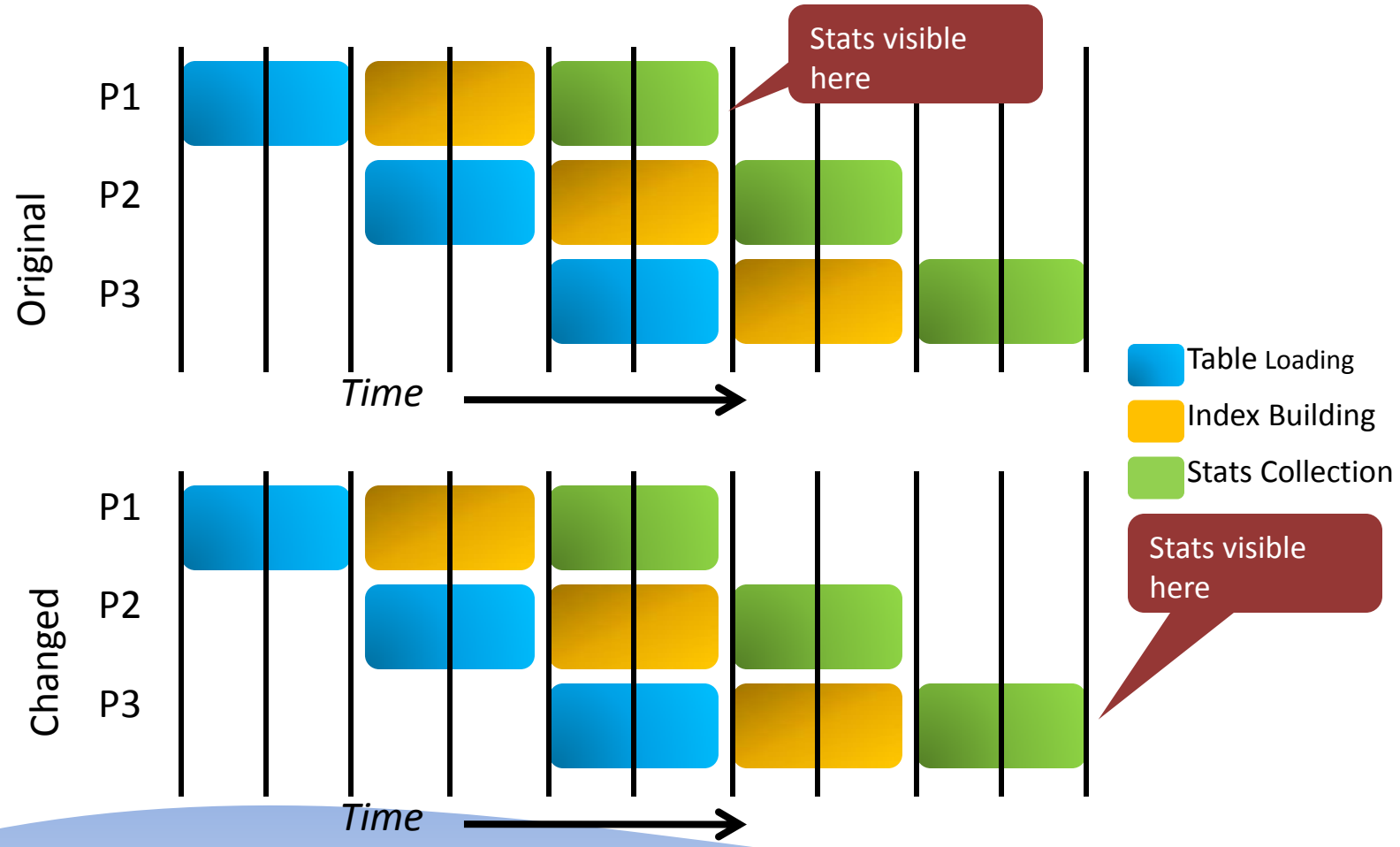



1. You may want to make sure that the final table global stats are collected after all partition stats are gathered
2. And all are visible to CBO at the same time

# Options

- You can postpone the stats collection of the partitions to the very end
- But that means you will lose the processing window that was available after the partition was loaded
- Better option: set the table's stats PUBLISH preference to FALSE
- Once the partition is loaded, collect the stat; but defer the publication to the very end

# Defer Partition Table Stats



# Stats History

- When new stats are collected, they are maintained in a history as well
- In the table `SYS.WRI$_OPTSTAT_TAB_HISTORY`
- Exposed through `*_TAB_STATS_HISTORY`

```
select to_char(stats_update_time,'mm/dd/yy hh24:mi:ss')
from user_tab_stats_history
where table_name = 'SALES';

    TO_CHAR(STATS_UPD
    -----------------
    03/01/10 21:32:57
    03/01/10 21:40:38
```

hist.sql_

# Reinstate the Stats

- Suppose things go wrong
- You wish the older stats were present rather than the newly collected ones
- You want to restore the old stats

```
begin
   dbms_stats.restore_table_stats (
       ownname         => 'ARUP',
       tabname         => 'SALES',
       as_of_timestamp => '14-SEP-13 11:59:00 AM'
   );
end;
```

reinstate.sql_

# Exporting the Pending Stats

- First create a table to hold the stats

```
begin
   dbms_stats.create_stat_table (
      ownname  => 'ARUP',
      stattab  => 'STAT_TABLE'
   );
end;
```

- This will create a table called `STAT_TABLE`
- This table will hold the pending stats

cr_stattab.sql_

# Export the stats

- Now export the pending stats to the newly created stats table

```
begin
   dbms_stats.export_pending_stats (
     tabname    => 'SALES',
     stattab    => 'STAT_TABLE'
   );
end;
```

export.sql
del_stats.sql
import.sql_

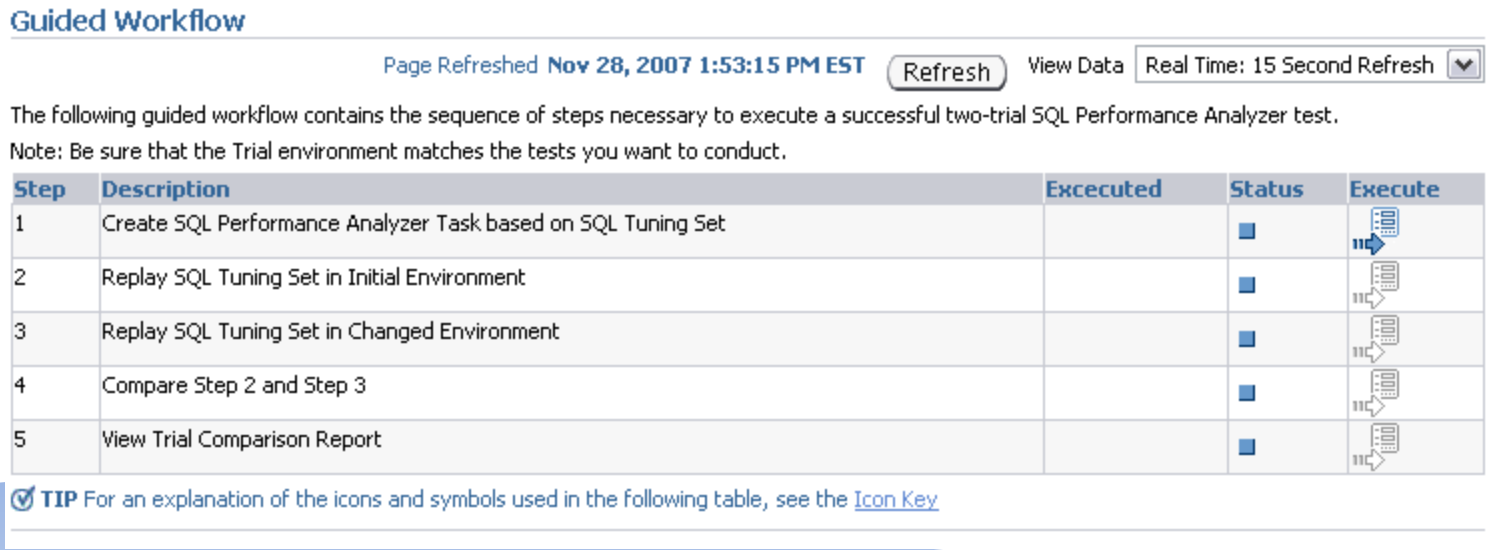
- Now you can export the table and plug in these stats in a test database
  - `dbms_stats.import_pending_stats`

# Some additional uses

- You can create a SQL Profile in your session
  - With private stats
- Then this profile can be applied to the other queries
- You can create SQL Plan Management Baselines based on these private stats
- Later you can apply these baselines to other sessions

# Real Application Testing

- You can use Database Replay and SQL Performance Analyzer to recreate the production workload
- But under the *pending* stats, to see the impact
- That way you can predict the impact of the new stats with your specific workload



Guided Workflow

Page Refreshed **Nov 28, 2007 1:53:15 PM EST** Refresh View Data Real Time: 15 Second Refresh

The following guided workflow contains the sequence of steps necessary to execute a successful two-trial SQL Performance Analyzer test.

Note: Be sure that the Trial environment matches the tests you want to conduct.

| Step | Description | Excecuted | Status | Execute |
|---|---|---|---|---|
| 1 | Create SQL Performance Analyzer Task based on SQL Tuning Set | | | |
| 2 | Replay SQL Tuning Set in Initial Environment | | | |
| 3 | Replay SQL Tuning Set in Changed Environment | | | |
| 4 | Compare Step 2 and Step 3 | | | |
| 5 | View Trial Comparison Report | | | |

**TIP** For an explanation of the icons and symbols used in the following table, see the Icon Key

# Sampling

- Estimate_Percent parameter of dbms_stats

```
begin
   dbms_stats.gather_table_stats (
      ownname => 'ARUP',
      tabname => 'SALES',
      estimate_percent => dbms_stats.auto_sample_size
);
end;
/
```

# Histograms

- Query

```
select … from customers where age = 35
```

- Should index be used?

| Age | Count |
|---|---|
| Under 30 | 10% |
| 30-40 | 80% |
| Over 40 | 10% |

| Age | Count |
|---|---|
| Under 30 | 80% |
| 30-40 | 10% |
| Over 40 | 10% |

| Age | Count |
|---|---|
| Under 30 | 10% |
| 30-35 | 10% |
| 36-40 | 70% |
| Over 40 | 10% |

```
method_opt => 'for all columns size auto'
exec :ret := dbms_stats.report_col_usage
('ARUP','SALES');
```

# Cardinality

$$\text{Number of Rows} \times \frac{1}{\text{Number of Distinct Values of Col1}} \times \frac{1}{\text{Number of Distinct Values of Col2}}$$

# Effect of Stats on Two Columns

- Optimizer Statistics on tables and indexes are vital for the optimizer to compute optimal execution plans
- If there are stats on two different columns used in the query, how does the optimizer decide?
- It takes the selectivity of each column, and multiplies that to get the selectivity for the query.

# Example

- Two columns
  - Month of Birth: selectivity = 1/12
  - Zodiac Sign: selectivity = 1/12
- What will be the selectivity of a query
  - Where zodiac sign = 'Pisces'
  - And month of birth = 'January'
- Problem:
  - According to the optimizer it will be 1/12 $\times$ 1/12 = 1/144
  - In reality, it will be 0, size the combination is not possible
- What will be the selectivity of a query
  - Where zodiac sign = 'Capricorn'
  - And month of birth = 'January'

# Multi-column Intelligence

- If the Optimizer knew about these combinations, it would have been able to choose the proper path
- How would you let the optimizer learn about these?
- In Oracle 10g, we saw a good approach – SQL Profiles
  - which allowed data to be considered for execution plans
  - but was not a complete approach
  - it still lacked a dynamism – applicability in all circumstances
- In 11g, there is an ability to provide this information to the optimizer
  - Multi-column stats

# An Example

- Table BOOKINGS
- Index on (HOTEL_ID, RATE_CODE)
- What will be plan for the following?

```
select min(book_txn)
from bookings
where hotel_id = 10
and rate_code = 23
```

| HOTEL_ID | RATE_CODE | COUNT(1) |
|---|---|---|
| 10 | 11 | 444578 |
| 10 | 12 | 50308 |
| 20 | 22 | 100635 |
| 20 | 23 | 404479 |

cr_bookings.sql
cr_indx.sql
ins_bookings.sql
stats.sql
vals.sql

# The Plan

## Here is the plan

expl1.sql

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |     1 |    10 |   769   (3)| 00:00:10 |
|   1 |  SORT AGGREGATE    |          |     1 |    10 |            |          |
|*  2 |   TABLE ACCESS FULL| BOOKINGS |   199K|  1951K|   769   (3)| 00:00:10 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
---------------------------------------------------
   2 - filter("RATE_CODE"=23 AND "HOTEL_ID"=10))
```

- It didn't choose index scan
- The estimated number of rows are 199K, or about 20%; so full table scan was favored over index scan

# Solution

- Create Extended Stats in the related columns – HOTEL_ID and RATE_CODE

```
var ret varchar2(2000)
begin
  :ret := dbms_stats.create_extended_stats(
    'ARUP', 'BOOKINGS','(HOTEL_ID, RATE_CODE)'
  );
end;
/
print ret
```

- The variable “ret” shows the name of the extended statistics

xstats.sql

# Then Collect Stats Normally

```
begin
  dbms_stats.gather_table_stats (
    ownname          => 'ARUP',
    tabname          => 'BOOKINGS',
    estimate_percent=> 100,
    method_opt       => 'FOR ALL COLUMNS SIZE SKEWONLY',
    cascade          => true
  );
end;
/
```

stats.sql

# The Plan Now

- After extended stats, the plan looks like this:

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |    10 |   325   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE              |                |     1 |    10 |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| BOOKINGS       | 23997 |   234K|   325   (1)| 00:00:04 |
|*  3 |    INDEX RANGE SCAN          | IN_BOOKINGS_01 | 23997 |       |    59   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

- Note:
  - No of Rows is now more accurate
  - As a result, the index scan was chosen

expl1.sql

# Extended Stats

- Extended stats store the correlation of data among the columns
  - The correlation helps optimizer decide on an execution path that takes into account the data
  - Execution plans are more accurate
- Under the covers,
  - extended stats create an invisible virtual column
  - Stats on the columns collects stats on this virtual column as well

# 10053 Trace

```
Single Table Cardinality Estimation for BOOKINGS[BOOKINGS]
  Column (#2):
    NewDensity:0.247422, OldDensity:0.000000 BktCnt:1000000,
PopBktCnt:1000000, PopValCnt:2, NDV:2
  Column (#3):
    NewDensity:0.025295, OldDensity:0.000000 BktCnt:1000000,
PopBktCnt:1000000, PopValCnt:4, NDV:4
  Column (#5):
    NewDensity:0.025295, OldDensity:0.000000 BktCnt:1000000,
PopBktCnt:1000000, PopValCnt:4, NDV:4
  ColGroup (#1, VC) SYS_STU4JHE7J4YQ3ZLDXSW5L1O8KX
    Col#: 2 3    CorStregth: 2.00
  ColGroup Usage:: PredCnt: 2  Matches Full:   Using density:
0.025295 of col #5 as selectivity of unpopular value pred
```

# Extended Stats

- This hidden virtual column shows up in column statistics

```
select column_name, density, num_distinct
from user_tab_col_statistics
where table_name = 'BOOKINGS‘

COLUMN_NAME                      DENSITY NUM_DISTINCT
------------------------------ ---------- ------------
BOOKING_ID                        .000001      1000000
HOTEL_ID                        .0000005            2
RATE_CODE                       .0000005            4
BOOK_TXN                       .002047465         2200
SYS_STU4JHE7J4YQ3ZLDXSW5L1O8KX  .0000005            4
```

Tabcolstats.sql

# Checking for Extended Stats

- To check the presence of extended stats, check the view dba_stat_extensions.

```
select extension_name, extension
from dba_stat_extensions
where table_name='BOOKINGS';
```

Output:

```
EXTENSION_NAME                 EXTENSION
------------------------------ ------------------------
SYS_STU4JHE7J4YQ3ZLDXSW5L1O8KX ("HOTEL_ID","RATE_CODE")
```

check.sql

# Deleting Extended Stats

- If you want, you can drop the extended stats, you can use the dbms_stats package, specifically the procedure drop_exteneded_stats

```
begin
  dbms_stats.drop_extended_stats (
    ownname => 'ARUP',
    tabname => 'BOOKINGS',
    extension => '("HOTEL_ID","RATE_CODE")'
   );
end;
```

drop.sql

# Another way

- You can collect the extended stats using the normal dbms_stats as well:

```
begin
    dbms_stats.gather_table_stats (
        ownname          => 'ARUP',
        tabname          => 'BOOKINGS',
        estimate_percent => 100,
        method_opt       =>
'FOR ALL COLUMNS SIZE SKEWONLY FOR COLUMNS
(HOTEL_ID,RATE_CODE)',
        cascade          => true
    );
end;
/
```

startx.sql

# The Case on Case Sensitivity

- A table of CUSTOMERS with 1 million rows
- LAST_NAME field has the values
  - McDonald - 20%
  - MCDONALD - 10%
  - McDONALD - 10%
  - mcdonald - 10%
- They make up 50% of the rows, with the variation of the same name.
- When you issue a query like this:

```
select * from customers where upper(last_name) = 'MCDONALD'
```

# Normal Plan

- The plan looks like this:

No of rows wrongly estimated

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           | 10000 |   498K|  2140   (2)| 00:00:26 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS | 10000 |   498K|  2140   (2)| 00:00:26 |
-------------------------------------------------------------------------------

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

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
   1 - filter(UPPER("LAST_NAME")='MCDONALD')
```

expl2.sql

# Extended Stats

- You collect the stats for the UPPER() function

```
begin
   dbms_stats.gather_table_stats (
      ownname     => 'ARUP',
      tabname     => 'CUSTOMERS',
      method_opt => 'for all columns
size skewonly for columns
(upper(last_name))'
   );
end;
```

statsx_cust.sql

# With Extended Stats

- The plan is now:

No of rows correctly estimated

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |   500K|    33M|  2140   (2)| 00:00:26 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |   500K|    33M|  2140   (2)| 00:00:26 |
-------------------------------------------------------------------------------

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

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
   1 - filter("CUSTOMERS"."SYS_STUJ6BPFDTE396EPTURAB2DBIE"='MCDONA
```

Extended stats name come here

expl2.sql

# Alternatives

- Remember, the extended stats create a virtual column – hidden from you
- You can have the same functionality as extended stats by defining virtual columns
- Advantage
  - You can have a column name of your choice
  - You can index it, if needed
  - You can partition it
  - You can create Foreign Key constraints on it

# Restrictions

- Has to be 11.0 or higher
- Not for SYS owned tables
- Not on IOT, clustered tables, GTT or external tables
- Can't be on a virtual column
- An Expression
  - can't contain a subquery
  - must have ≥1 columns
- A Column Group
  - no of columns should be ≤32 and ≥2
  - can't contain expressions
  - can't have the same column repeated

# Column Usage

```
SQL> select
dbms_stats.report_col_usage('ARUP','ACCOUNTS') from
dual;

DBMS_STATS.REPORT_COL_USAGE('ARUP','ACCOUNTS')
------------------------------------------------------------------------------
LEGEND:
.......

EQ         : Used in single table EQuality predicate
RANGE      : Used in single table RANGE predicate
LIKE       : Used in single table LIKE predicate
NULL       : Used in single table is (not) NULL predicate
EQ_JOIN    : Used in EQuality JOIN predicate
NONEQ_JOIN : Used in NON EQuality JOIN predicate
FILTER     : Used in single table FILTER predicate
JOIN       : Used in JOIN predicate
GROUP_BY   : Used in GROUP BY expression
.....................................................................

###############################################################################

COLUMN USAGE REPORT FOR ARUP.ACCOUNTS
.....................................

1. ACCNO                       : EQ
```

# *Thank You!*

My Blog: **arup.blogspot.com**
My Tweeter: **arupnanda**