



Oracle White Paper  
February 2015

# Understanding Optimizer Statistics with Oracle Database 12c

## Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

---

Introduction .....	1
What are Optimizer Statistics? .....	2
Table and Column Statistics .....	2
Additional column statistics.....	3
Index Statistics .....	14
Gathering Statistics .....	15
GATHER_TABLE_STATS.....	15
Changing the default value for the parameters in DBMS_STATS.GATHER_*_STATS .....	17
Automatic Statistics Gathering Job .....	19
Improving the efficiency of Gathering Statistics .....	22
Concurrent Statistic gathering.....	22
Gathering Statistics on Partitioned tables .....	24
Managing statistics .....	26
Restoring Statistics.....	26
Pending Statistics .....	27
Exporting / Importing Statistics .....	28
Copying Partition Statistics .....	29
Comparing Statistics.....	30
Locking Statistics.....	31
Manually setting Statistics .....	33
Other Types of Statistics .....	34
Dynamic Statistics (previously known as dynamic sampling) .....	34
System statistics.....	36
Statistics on Dictionary Tables.....	36
Statistics on Fixed Objects .....	37
Conclusion .....	38

## Introduction

When the Oracle database was first introduced, the decision of how to execute a SQL statement was determined by a Rule Based Optimizer (RBO). The Rule Based Optimizer, as the name implies, followed a set of rules to determine the execution plan for a SQL statement.

In Oracle Database 7, the Cost Based Optimizer (CBO) was introduced to deal with the enhanced functionality being added to the Oracle Database at this time, including parallel execution and partitioning, and to take the actual data content and distribution into account. The Cost Based Optimizer examines all of the possible plans for a SQL statement and picks the one with the lowest cost, where cost represents the estimated resource usage for a given plan. The lower the cost, the more efficient an execution plan is expected to be. In order for the Cost Based Optimizer to accurately determine the cost for an execution plan, it must have information about all of the objects (tables and indexes) accessed in the SQL statement, and information about the system on which the SQL statement will be run.

This necessary information is commonly referred to as **Optimizer statistics**. Understanding and managing Optimizer statistics is key to optimal SQL execution. Knowing when and how to gather statistics in a timely manner is critical to maintaining acceptable performance. This whitepaper is the first in a two part series on Optimizer statistics, and describes in detail, with worked examples, the different concepts of Optimizer statistics including;

- What are Optimizer statistics
- Gathering statistics
- Managing statistics
- Additional types of statistics

## What are Optimizer Statistics?

Optimizer statistics are a collection of data that describe the database and the objects in the database. These statistics are used by the Optimizer to choose the best execution plan for each SQL statement. Statistics are stored in the data dictionary and can be accessed using data dictionary views such as `USER_TAB_STATISTICS`. Optimizer statistics are different from the performance statistics visible through `V$` views. The information in the `V$` views relates to the state of the system and the SQL workload executing on it.

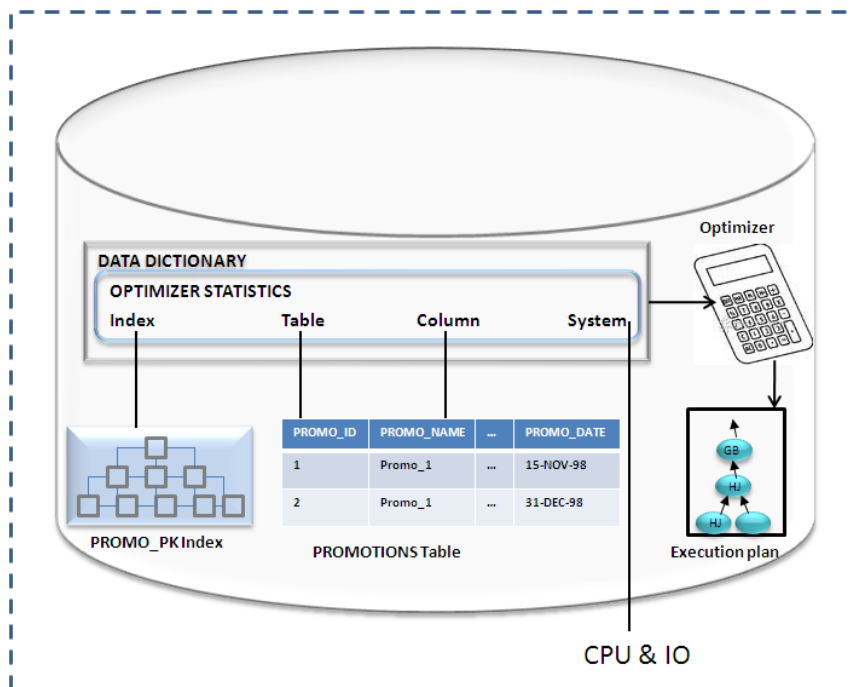


Figure 1. Optimizer Statistics stored in the data dictionary used by the Optimizer to determine execution plans

### Table and Column Statistics

Table statistics include information on the number of rows in the table, the number of data blocks used for the table, as well as the average row length in the table. The Optimizer uses this information, in conjunction with other statistics, to compute the cost of various operations in an execution plan, and to estimate the number of rows the operation will produce. For example, the cost of a table access is calculated using the number of data blocks combined with the value of the parameter `DB_FILE_MULTIBLOCK_READ_COUNT`. You can view table statistics in the dictionary view `USER_TAB_STATISTICS`.

Column statistics include information on the number of distinct values in a column (NDV) as well as the minimum and maximum value found in the column. You can view column statistics in the dictionary view `USER_TAB_COL_STATISTICS`. The Optimizer uses the column statistics information in conjunction with the table statistics (number of rows) to estimate the number of rows that will be

returned by a SQL operation. For example, if a table has 100 records, and the table access evaluates an equality predicate on a column that has 10 distinct values, then the Optimizer, assuming uniform data distribution, estimates the cardinality to be the number of rows in the table divided by the number of distinct values for the column or  $100/10 = 10$ .

```
SQL> SELECT count(*)
2 FROM tab_with_100_rows
3 where col_ndv_10 = 2;

COUNT(*)
-----
10
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	3	3 (100)	
1	SORT AGGREGATE		1	3	3 (100)	
* 2	TABLE ACCESS STORAGE FULL	TAB_WITH_100_ROWS	10	30	3 (0)	00:00:01

Predicate Information (identified by operation id):

```
2 - storage("COL_NDV_10"=2)
   filter("COL_NDV_10"=2)
```

**Cardinality estimate of 10 is calculated by dividing NUM\_ROWS (100) for the table by NUM\_DISTINCT (10) for the column**

Figure 2. Cardinality calculation using basic table and column statistics

### Additional column statistics

Basic table and column statistics tell the optimizer a great deal, but they don't provide a mechanism to tell the Optimizer about the nature of the data in the table or column. For example, these statistics can't tell the Optimizer if there is a data skew in a column, or if there is a correlation between columns in a table. Information on the nature of the data can be provided to the Optimizer by using extensions to basic statistics like, histograms, column groups, and expression statistics.

### Histograms

Histograms tell the Optimizer about the distribution of data within a column. By default (without a histogram), the Optimizer assumes a uniform distribution of rows across the distinct values in a column. As described above, the Optimizer calculates the cardinality for an equality predicate by dividing the total number of rows in the table by the number of distinct values in the column used in the equality predicate. If the data distribution in that column is not uniform (i.e., a data skew) then the cardinality estimate will be incorrect. In order to accurately reflect a non-uniform data distribution, a histogram is required on the column. The presence of a histogram changes the formula used by the Optimizer to estimate the cardinality, and allows it to generate a more accurate execution plan.

Oracle automatically determines the columns that need histograms based on the column usage information (`SYS.COL_USAGE$`), and the presence of a data skew. For example, Oracle will not automatically create a histogram on a unique column if it is only seen in equality predicates.

There are four types of histograms: frequency, top-frequency, or height-balanced and hybrid. Oracle determines the type of histogram to be created based on the number of distinct values in the column. From Oracle Database 12c onwards, height-balance histograms will be replaced by hybrid histograms<sup>1</sup>.

### Frequency Histograms

Frequency histograms are created when the number of distinct values in the column is less than 254. Oracle uses the following steps to create a frequency histogram.

1. Let's assume that Oracle is creating a frequency histogram on the `PROMO_CATEGORY_ID` column of the `PROMOTIONS` table. The first step is to select the `PROMO_CATEGORY_ID` from the `PROMOTIONS` table ordered by `PROMO_CATEGORY_ID`.
2. Each `PROMO_CATEGORY_ID` is then assigned to its own histogram bucket (Figure 3).

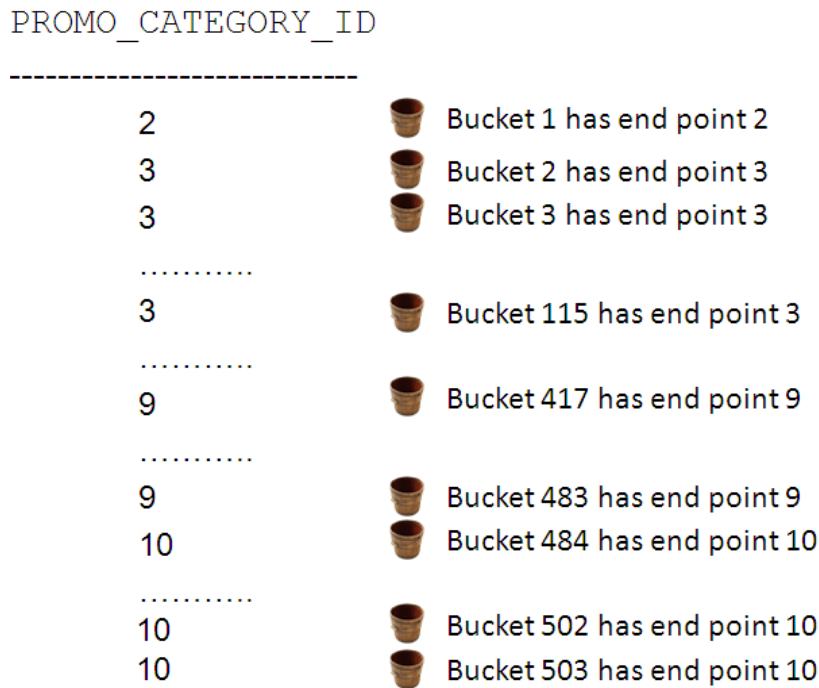


Figure 3. Step 2 in frequency histogram creation

<sup>1</sup> Assuming the parameter `ESITMATE_PERCENT` is let default in the `DBMS_STATS.GATHER_*_STATS` command used to gather the statistics

- At this stage, we could have more than 254 histogram buckets, so the buckets that hold the same value are then compressed into the highest bucket with that value. In this case, buckets 2 through 115 are compressed into bucket 115, and buckets 484 through 503 are compressed into bucket 503, and so on until the total number of buckets remaining equals the number of distinct values in the column (Figure 4). Note the above steps are for illustration purposes. The `DBMS_STATS` package has been optimized to build compressed histograms directly.

PROMO\_CATEGORY\_ID

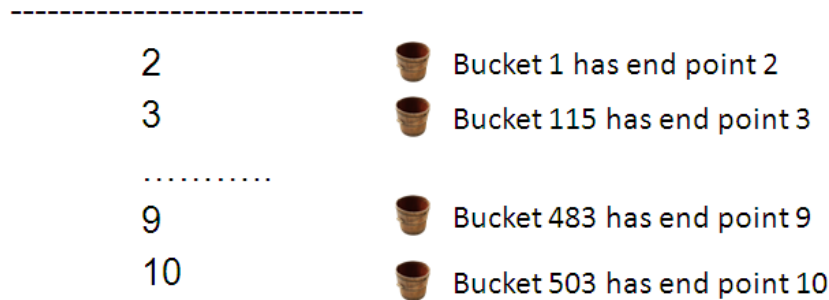


Figure 4. Step 3 in frequency histogram creation: duplicate buckets are compressed

- The Optimizer now accurately determines the cardinality for predicates on the `PROMO_CATEGORY_ID` column using the frequency histogram. For example, for the predicate `PROMO_CATEGORY_ID = 10`, the Optimizer would first need to determine how many buckets in the histogram have 10 as their end point. It does this by finding the bucket whose endpoint is 10, bucket 503, and then subtracts the previous bucket number, bucket 483,  $503 - 483 = 20$ . Then the cardinality estimate would be calculated using the following formula (number of bucket endpoints / total number of bucket) X `NUM_ROWS`,  $20/503 \times 503$ , so the number of rows in the `PROMOTOINS` table where `PROMO_CATEGORY_ID = 10` is 20.

#### Top-Frequency Histograms

Traditionally, if a column had more than 254 distinct values and the number of buckets specified is `AUTO`, a height-balanced histogram would be created. But what if 99% or more of the rows in the table had less than 254 distinct values? If a height-balance histogram is created, it runs the risk of not capturing all of the popular values in the table as the endpoint of multiple buckets. Thus some popular values will be treated as non-popular values, which could result in a sub-optimal execution plan being chosen.

In this scenario, it would be better to create a frequency histogram on the extremely popular values that make up the majority of rows in the table and to ignore the unpopular values, in order to create a better quality histogram. This is the exact approach taken with top-frequency histograms. A frequency histogram is created on the most popular values in the column, when those values appear in 99% or more of the rows in the table. This allows all of the popular values in the column to be treated as such. A top-frequency histogram is only created if the `ESTIMATE_PERCENT` parameter of the `gather statistics` command is set to `AUTO_SAMPLE_SIZE`, as all values in the column must be seen in order to determine if the necessary criteria are met (99.6% of rows have 254 or less distinct values).



Take, for example, the `PRODUCT_SALES` table, which contains sales information for a Christmas ornaments company. The table has 1.78 million rows and 632 distinct `TIME_ID`s. But the majority of the rows in `PRODUCT_SALES` have less than 254 distinct `TIME_ID`s, as the majority of Christmas ornaments are sold in December each year. A histogram is necessary on the `TIME_ID` column to make the Optimizer aware of the data skew in the column. In this case, a top-frequency histogram is created containing 254 buckets.

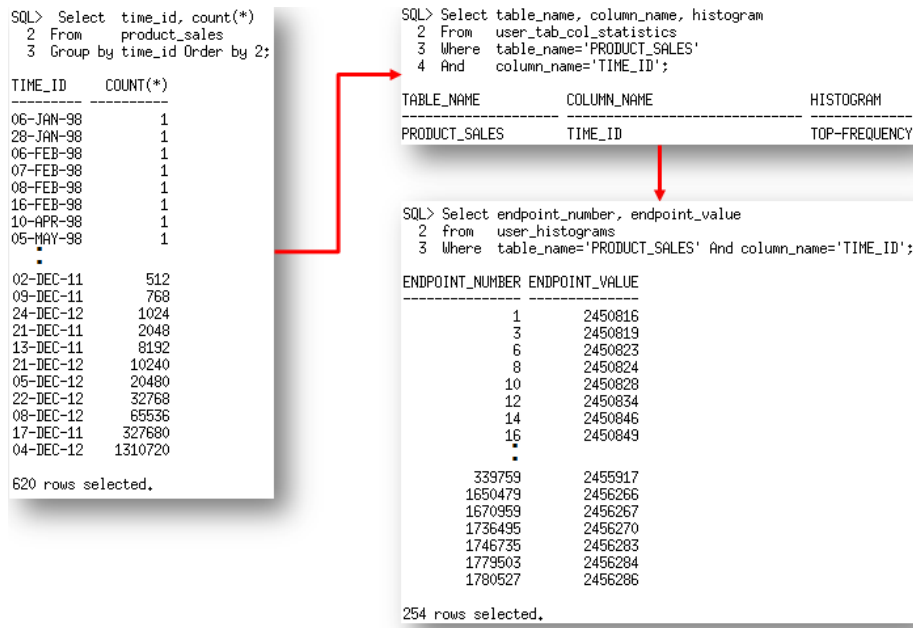


Figure 5. Data distribution of `TIME_ID` column in `PRODUCT_SALES` table & top-frequency histogram that is created on it

### Height balanced Histograms

Height-balanced histograms are created when the number of distinct values in the column is greater than 254. In a height-balanced histogram, column values are divided into buckets so that each bucket contains approximately the same number of rows. Oracle uses the following steps to create a height-balanced histogram.

1. Let's assume that Oracle is creating a height-balanced histogram on the `CUST_CITY_ID` column of the `CUSTOMERS` table because the number of distinct values in the `CUST_CITY_ID` column is greater than 254. Just like with a frequency histogram, the first step is to select the `CUST_CITY_ID` from the `CUSTOMERS` table ordered by `CUST_CITY_ID`.
2. There are 55,500 rows in the `CUSTOMERS` table and there is a maximum of 254 buckets in a histogram. In order to have an equal number of rows in each bucket, Oracle must put 219 rows in each bucket. The 219<sup>th</sup> `CUST_CITY_ID` from step one will become the endpoint for the first bucket. In this case that is 51043. The 438<sup>th</sup> `CUST_CITY_ID` from step one will become the endpoint for the second bucket, and so on until all 254 buckets are filled (Figure 6).






Row count	CUST_CITY_ID	
1	51040	
2	51040	
:	:	
219	51043	 Bucket 1 has end point 51043
:	:	
438	51044	 Bucket 2 has end point 51044
:	:	
5256	51166	 Bucket 24 has end point 51166
:	:	
5475	51166	 Bucket 25 has end point 51166
:	:	
55500	52531	 Bucket 254 has end point 51531

Figure 6. Step 2 of height-balance histogram creation: put an equal number of rows in each bucket

- Once the buckets have been created, Oracle checks to see if the endpoint of the first bucket is the minimum value for the CUST\_CITY\_ID column. If it is not, a “zero” bucket is added to the histogram that has the minimum value for the CUST\_CITY\_ID column as its end point (Figure 7).







Row count	CUST_CITY_ID	
1	51040	 Bucket 0 has end point 51040
2	51040	
:	:	
219	51043	 Bucket 1 has end point 51043
:	:	
438	51044	 Bucket 2 has end point 51044
:	:	
5256	51166	 Bucket 24 has end point 51166
:	:	
5475	51166	 Bucket 25 has end point 51166
:	:	
55500	52531	 Bucket 254 has end point 51531

Figure 7. Step 3 of height-balance histogram creation: add a zero bucket for the min value

- Just as with a frequency histogram, the final step is to compress the height-balanced histogram, and remove the buckets with duplicate end points. The value 51166 is the end

point for bucket 24 and bucket 25 in our height-balanced histogram on the CUST\_CITY\_ID column. So, bucket 24 will be compressed in bucket 25 (Figure 8).






Row count	CUST_CITY_ID	
1	51040	 Bucket 0 has end point 51040
2	51040	
:	:	
219	51043	 Bucket 1 has end point 51043
:	:	
438	51044	 Bucket 2 has end point 51044
:	:	
5475	51166	 Bucket 25 has end point 51166
:	:	
55500	52531	 Bucket 254 has end point 51531

Figure 8. Step 4 of height-balance histogram creation

- The Optimizer now computes a better cardinality estimate for predicates on the CUST\_CITY\_ID column by using the height-balanced histogram. For example, for the predicate CUST\_CITY\_ID = 51806, the Optimizer would first check to see how many buckets in the histogram have 51806 as their end point. In this case, the endpoint for bucket 136,137,138 and 139 is 51806 (info found in USER\_HISTOGRAMS). The Optimizer then uses the following formula:

(Number of bucket endpoints / total number of buckets) X number of rows in the table

In this case  $4/254 \times 55500 = 874$

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				406 (100)	
1	SORT AGGREGATE		1	5		
* 2	TABLE ACCESS FULL	CUSTOMERS	874	4370	406 (1)	00:00:01

Predicate Information (identified by operation id):

2 - filter("CUST\_CITY\_ID"=51806)

**Estimated determined using formula (num endpoints / total num buckets) X num\_rows**

Figure 9. Height balanced histogram used for popular value cardinality estimate

However, if the predicate was CUST\_CITY\_ID = 52500, which is not the endpoint for any bucket then the Optimizer uses a different formula. For values that are the endpoint for only one bucket or are not an endpoint at all, the Optimizer uses the following formula:

DENSITY X number of rows in the table

where `DENSITY` is calculated ‘on the fly’ during optimization using an internal formula based on information in the histogram. The value for `DENSITY` seen in the dictionary view `USER_TAB_COL_STATISTICS` is not the value used by the Optimizer from Oracle Database 10.2.0.4 onwards. This value is recorded for backward compatibility, as this is the value used in Oracle Database 9i and earlier releases of 10g. Furthermore, if the parameter `OPTIMIZER_FEATURES_ENABLE` is set to version release earlier than 10.2.0.4, the value for `DENSITY` in the dictionary view will be used.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				405 (100)	
1	SORT AGGREGATE		1	5		
* 2	TABLE ACCESS FULL	CUSTOMERS	68	340	405 (1)	00:00:01

Predicate Information (identified by operation id):

2 - filter("CUST\_CITY\_ID"=52500)

**Estimated determined using formula (density X num\_rows)**

Figure 10. Height balanced histogram used for non-popular value cardinality estimate

### Hybrid histograms

One prominent problem with height balanced histograms is that a value with a frequency that falls into the range of  $1/254$  of the total population and  $2/254$  of the total population may or may not appear as a popular value. Although it might span across two buckets, it may only appear as the end point value of one bucket. Such values are referred to as almost popular values. Height balanced histograms do not differentiate between almost popular values and truly unpopular values.

A hybrid histogram is similar to the traditional height-balanced histogram, as it is created when the number of distinct values in a column is greater than 254. However, that’s where the similarities end. With a hybrid histogram, no value will be the endpoint of more than one bucket, thus allowing the histogram to have more endpoint values, or effectively more buckets, than a height-balanced histogram. So, how does a hybrid histogram indicate a popular value? The frequency of each endpoint value is recorded (in a new column `endpoint_repeat_count`), thus providing an accurate indication of the popularity of each endpoint value.

Take, for example, the `CUST_CITY_ID` column in the `CUSTOMERS` table. There are 55,500 rows in the `CUSTOMERS` table and 620 distinct values in the `CUST_CITY_ID` column. Neither a frequency nor a top-frequency histogram is an option in this case. In Oracle Database 11g, a height-balanced histogram is created on this column. The height-balanced histogram has 213 buckets, but only represents 42 popular values (value is the endpoint of 2 or more buckets). The actual number of popular values in `CUST_CITY_ID` column is 54 (i.e., column values with a frequency that is larger than  $\text{num\_rows}/\text{num\_buckets} = 55500/254 = 54$ ).

In Oracle Database 12c a hybrid histogram is created. The hybrid histogram has 254 buckets and represents all 54 popular values. The hybrid histogram actually treats 63 values as popular values. This means that values that were considered as nearly popular (endpoint value of only 1 bucket) in Oracle

Database 11g are now treated as popular values and will have a more accurate cardinality estimate. Figure 11 shows an example of how a nearly popular value (52114) in Oracle Database 11g gets a much better cardinality estimate in Oracle Database 12c.

There are 227 rows with the CUST\_CITY\_ID of 52114

```
SQL> select count(*) from customers where CUST_CITY_ID=52114;
```

```

COUNT(*)
-----
227
    
```

In Oracle Database 11g a height balanced histogram is created on CUST\_CITY\_ID. 52114 is the endpoint of just one bucket in the histogram.

```
SQL> Select c.column_name, c.histogram, h.endpoint_number, h.endpoint_value
2 From user_tab_col_statistics c, user_histograms h
3 Where c.table_name='CUSTOMERS'
4 And c.column_name='CUST_CITY_ID'
5 And c.table_name=h.table_name
6 And c.column_name=h.column_name
7 And h.endpoint_value='52114';
```

COLUMN_NAME	HISTOGRAM	ENDPOINT_NUMBER	ENDPOINT_VALUE
CUST_CITY_ID	HEIGHT BALANCED	190	52114

As the endpoint of just one bucket, 52114 is considered a non-popular values and its cardinality estimates is determined using the following formula  
 $DENSITY \times NUM\_ROWS$

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT				420 (100)
1	SORT AGGREGATE			5	
* 2	TABLE ACCESS FULL	CUSTOMERS	66	330	420 (7)

In Oracle Database 12c a hybrid histogram is created on CUST\_CITY\_ID. 52114 is the end point of just one bucket of the histogram

```
SQL> Select c.column_name, c.histogram, h.endpoint_number, h.endpoint_value, h.endpoint_repeat_count
2 From user_tab_col_statistics c, user_histograms h
3 Where c.table_name='CUSTOMERS'
4 And c.column_name='CUST_CITY_ID'
5 And c.table_name=h.table_name
6 And c.column_name=h.column_name
7 And h.endpoint_value='52114';
```

COLUMN_NAME	HISTOGRAM	ENDPOINT_NUMBER	ENDPOINT_VALUE	ENDPOINT_REPEAT_COUNT
CUST_CITY_ID	HYBRID	4083	52114	24

As the endpoint of one bucket in a hybrid histogram 52114 is a popular value and its cardinality estimate determined using the following formula

Cardinality estimates  $\frac{endpoint\_repeat\_count}{max(endpoint\_number)} \times NUM\_ROWS$

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT				468 (100)
1	SORT AGGREGATE			5	
* 2	TABLE ACCESS FULL	CUSTOMERS	250	1250	468 (10)

Figure 11. Hybrid histograms achieve more accurate cardinality estimates for what was considered a nearly popular value in Oracle Database 11g

As mentioned earlier, hybrid histograms are the new default histogram type for columns with greater than 254 distinct values, as long as the statistics are gathered using the default ESTIMATE\_PERCENT setting.

## Extended Statistics

In Oracle Database 11g, extensions to column statistics were introduced. Extended statistics encompass two additional types of statistics; column groups and expression statistics.

### Column Groups

In real-world data, there is often a relationship (correlation) between the data stored in different columns of the same table. For example, in the `CUSTOMERS` table, the values in the `CUST_STATE_PROVINCE` column are influenced by the values in the `COUNTRY_ID` column, as the state of California is only going to be found in the United States. Using only basic column statistics, the Optimizer has no way of knowing about these real-world relationships, and could potentially miscalculate the cardinality if multiple columns from the same table are used in the where clause of a statement. The Optimizer can be made aware of these real-world relationships by having extended statistics on these columns as a group.

By creating statistics on a group of columns, the Optimizer can compute a better cardinality estimate when several the columns from the same table are used together in a where clause of a SQL statement. You can use the function `DBMS_STATS.CREATE_EXTENDED_STATS` to define a column group you want to have statistics gathered on as a group. Once a column group has been created, Oracle will automatically maintain the statistics on that column group when statistics are gathered on the table, just like it does for any ordinary column (Figure 12).

```
SQL> SELECT DBMS_STATS.CREATE_EXTENDED_STATS(null,'customers', '(country_id, cust_state_province)')
2 FROM dual;
DBMS_STATS.CREATE_EXTENDED_STATS(NULL,'CUSTOMERS','(COUNTRY_ID,CUST_STATE_PROVINCE)')
-----
SYS_STUJGVLRVH5USVDU$XNV4_IR#4
SQL>
SQL> Exec DBMS_STATS.GATHER_TABLE_STATS(null,'customers');
PL/SQL procedure successfully completed.
```

Figure 12. Creating a column group on the `CUSTOMERS` table

After creating the column group and re-gathering statistics, you will see an additional column, with a system-generated name, in the dictionary view `USER_TAB_COL_STATISTICS`. This new column represents the column group (Figure 13).

```
SQL> SELECT column_name, num_distinct, num_nulls, histogram
2 FROM user_tab_col_statistics
3 WHERE table_name='CUSTOMERS';
```

COLUMN_NAME	NUM_DISTINCT	NUM_NULLS	HISTOGRAM
SYS_STUJGVLRVH5USVDU\$XNV4_IR#4	145	0	NONE
CUST_ID	55500	0	NONE
CUST_FIRST_NAME	1300	0	NONE
CUST_LAST_NAME	908	0	NONE
CUST_GENDER	2	0	NONE
CUST_YEAR_OF_BIRTH	75	0	NONE
CUST_MARITAL_STATUS	11	17428	NONE
CUST_STREET_ADDRESS	49900	0	NONE
CUST_POSTAL_CODE	623	0	NONE
CUST_CITY	620	0	NONE
CUST_CITY_ID	620	0	HEIGHT BALANCED
CUST_STATE_PROVINCE	145	0	FREQUENCY
CUST_STATE_PROVINCE_ID	145	0	FREQUENCY
COUNTRY_ID	19	0	FREQUENCY
CUST_MAIN_PHONE_NUMBER	51344	0	NONE
CUST_INCOME_LEVEL	12	41	NONE
CUST_CREDIT_LIMIT	8	0	NONE
CUST_EMAIL	1699	0	NONE
CUST_TOTAL	1	0	NONE
CUST_TOTAL_ID	1	0	FREQUENCY
CUST_SRC_ID	0	55500	NONE
CUST_EFF_FROM	1	0	NONE
CUST_EFF_TO	0	55500	NONE
CUST_VALID	2	0	NONE

Figure 13. System generated column name for a column group in USER\_TAB\_COL\_STATISTICS

To map the system-generated column name to the column group and to see what other extended statistics exist for a user schema, you can query the dictionary view USER\_STAT\_EXTENSIONS (Figure 14).

```
SQL> SELECT table_name, extension_name, extension
2 FROM user_stat_extensions
3 WHERE creator = 'USER';
```

TABLE_NAME	EXTENSION_NAME	EXTENSION
CUSTOMERS	SYS_STUJGVLRVH5USVDU\$XNV4_IR#4	("COUNTRY_ID", "CUST_STATE_PROVINCE")
SALES	SYS_STU05K\$0ZAT82HFHJUUK6WDCLL	("PROD_ID", "CUST_ID")

Figure 14. Information about column groups is stored in USER\_STAT\_EXTENSIONS

The Optimizer will now use the column group statistics, rather than the individual column statistics when these columns are used together in where clause predicates. Not all of the columns in the column group need to be present in the SQL statement for the Optimizer to use extended statistics; only a subset of the columns is necessary.

### Auto Column Groups Detection

Although column group statistics are extremely useful and often necessary to achieve an optimal execution plan it can be difficult to know what column group statistics should be created for a given workload.

Auto column group detection automatically determines which column groups are required for a table based on a given workload. Please note this functionality does not create extended statistics for function wrapped columns it is only for column groups. Auto Column Group detection is a simple three-step process:

## 1. Seed column usage

Oracle must observe a representative workload in order to determine the appropriate column groups. The workload can be provided in a SQL Tuning Set or by monitoring a running system. The new procedure, `DBMS_STATS.SEED_COL_USAGE`, should be used to indicate the workload and to tell Oracle how long it should observe that workload. The following example turns on monitoring for 5 minutes or 300 seconds for the current system.

```
SQL> connect /as sysdba
Connected.
SQL>
SQL> -- Switch on seed column usage for 300 seconds
SQL> BEGIN
2  dbms_stats.seed_col_usage(null,null, 300);
3  END;
4  /

PL/SQL procedure successfully completed.
```

Figure 15. Enabling automatic column group detection

The monitoring procedure records different information from the traditional column usage information you see in `sys.col_usage$` and stores it in `sys.col_group_usage$`. Information is stored for any SQL statement that is executed or explained during the monitoring window. Once the monitoring window has finished, it is possible to review the column usage information recorded for a specific table using the new function `DBMS_STATS.REPORT_COL_USAGE`. This function generates a report, which lists what columns from the table were seen in filter predicates, join predicates and group by clauses in the workload. It is also possible to view a report for all the tables in a specific schema by running `DBMS_STATS.REPORT_COL_USAGE` and providing just the schema name and NULL for the table name.

```
SQL> Select dbms_stats.report_col_usage(user, 'Customers')
2  From dual
3  /

DBMS_STATS.REPORT_COL_USAGE(USER, 'CUSTOMERS')
-----
LEGEND:
.....
EO       : 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
EO_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 SH,CUSTOMERS
*****
```

Figure 16. Enabling automatic column group detection

## 2. Create the column groups

Calling the `DBMS_STATS.CREATE_EXTENDED_STATS` function for each table will automatically create the necessary column groups based on the usage information captured during the monitoring window. Once the extended statistics have been created, they will be automatically maintained whenever statistics are gathered on the table.

Alternatively, the column groups can be manually created by specifying the group as the third argument in the `DBMS_STATS.CREATE_EXTENDED_STATS` function.



```

SQL> Select dbms_stats.create_extended_stats(user, 'customers') from dual;
DBMS_STATS.CREATE_EXTENDED_STATS(USER, 'CUSTOMERS')
-----
*****
EXTENSIONS FOR SH.CUSTOMERS
*****
1. (CUST_CITY, CUST_STATE_PROVINCE,
   COUNTRY_ID) : SYS_STUHZC3A1HLPBRO1#SK468H_N created
2. (CUST_STATE_PROVINCE, COUNTRY_ID) : SYS_STUNS#UF25Z#QAHHE#H0FFHM_ created
*****

```

Figure 17. Create the automatically detected column groups

### 3. Regather statistics

The final step is to regather statistics on the affected tables so that the newly created column groups will have statistics created for them.

```

SQL> BEGIN
2   dbms_stats.gather_table_stats(user, 'customers');
3 END;
4 /

PL/SQL procedure successfully completed.

```

Figure 18. Column group statistics are automatically maintained every time statistics are gathered

### Expression Statistics

It is also possible to create extended statistics for an expression (including functions), to help the Optimizer to estimate the cardinality of a where clause predicate that has columns embedded inside expressions. For example, if it is common to have a where clause predicate that uses the UPPER function on a customer's last name, `UPPER(CUST_LAST_NAME) = :B1`, then it would be beneficial to create extended statistics for the expression `UPPER(CUST_LAST_NAME)` (Figure 19).

```

SQL> SELECT DBMS_STATS.CREATE_EXTENDED_STATS(NULL, 'CUSTOMERS', '(UPPER(CUST_LAST_NAME))')
2 FROM dual;

DBMS_STATS.CREATE_EXTENDED_STATS(NULL, 'CUSTOMERS', '(UPPER(CUST_LAST_NAME))')
-----
SYS_STUSKCCJE8MV8IIBWT5PA5A41V

```

Figure 19. Extended statistics can also be created on expressions

Just as with column groups, statistics need to be re-gathered on the table after the expression statistics have been defined. After the statistics have been gathered, an additional column with a system-generated name will appear in the dictionary view `USER_TAB_COL_STATISTICS` representing the expression statistics. Just like for column groups, the detailed information about expression statistics can be found in `USER_STAT_EXTENSIONS`.

### Restrictions on Extended Statistics

Extended statistics can only be used when the where clause predicates are equalities or in-lists. Extended statistics will not be used if there are histograms present on the underlying columns and there is no histogram present on the column group.

### Index Statistics

Index statistics provide information on the number of distinct values in the index (distinct keys), the depth of the index (blevel), the number of leaf blocks in the index (leaf\_blocks), and the clustering

factor<sup>2</sup>. The Optimizer uses this information in conjunction with other statistics to determine the cost of an index access. For example, the Optimizer will use `b-level`, `leaf_blocks` and the table statistics `num_rows` to determine the cost of an index range scan (when all predicates are on the leading edge of the index).

## Gathering Statistics

For database objects that are constantly changing, statistics must be regularly gathered so that they accurately describe the database object. The PL/SQL package, `DBMS_STATS`, is Oracle's preferred method for gathering statistics, and replaces the now obsolete `ANALYZE`<sup>3</sup> command for collecting statistics. The `DBMS_STATS` package contains over 50 different procedures for gathering and managing statistics. The most important procedures are the `GATHER_*_STATS` procedures, which can be used to gather table, column, and index statistics. You will need to be the owner of the object or have the `ANALYZE ANY` system privilege or the `DBA` role to run these procedures. The parameters used by these procedures are nearly identical, so this paper will focus on the `GATHER_TABLE_STATS` procedure.

### GATHER\_TABLE\_STATS

The `DBMS_STATS.GATHER_TABLE_STATS` procedure allows you to gather table, partition, index, and column statistics. Although it takes 15 different parameters, only the first two or three parameters need to be specified to run the procedure, and are sufficient for most customers;

- The name of the schema containing the table
- The name of the table
- A specific partition name if it's a partitioned table and you only want to collect statistics for a specific partition (optional)

```
SQL> BEGIN
  2  dbms_stats.gather_table_stats('SH','SALES');
  3  END;
  4  /
```

```
PL/SQL procedure successfully completed.
```

Figure 20. Using the `DBMS_STATS.GATHER_TABLE_STATS` procedure

The remaining parameters can be left at their default values in most cases. Out of the remaining 12 parameters, the following are often changed from their default and warrant some explanation here.

<sup>2</sup> Chapter 11 of the [Oracle® Database Performance Tuning Guide](#)

<sup>3</sup> `ANALYZE` command is still used to `VALIDATE` or `LIST CHAINED ROWS`.

**ESTIMATE\_PERCENT parameter**

The `ESTIMATE_PERCENT` parameter determines the percentage of rows used to calculate the statistics. The most accurate statistics are gathered when all rows in the table are processed (i.e., 100% sample), often referred to as computed statistics. Oracle Database 11g introduced a new sampling algorithm that is hash based and provides deterministic statistics. This new approach has the accuracy close to a 100% sample but with the cost of, at most, a 10% sample. The new algorithm is used when `ESTIMATE_PERCENT` is set to `AUTO_SAMPLE_SIZE` (the default) in any of the `DBMS_STATS.GATHER_*_STATS` procedures. Historically, customers have set the `ESTIMATE_PERCENT` parameter to a low value to ensure that the statistics will be gathered quickly. However, without detailed testing, it is difficult to know which sample size to use to get accurate statistics. It is highly recommended that from Oracle Database 11g onward you let `ESTIMATE_PERCENT` default (i.e., not set explicitly).

**METHOD\_OPT parameter**

The `METHOD_OPT` parameter controls the creation of histograms during statistics collection. Histograms are a special type of column statistic created when the data in a table column has a non-uniform distribution, as discussed in the previous section of this paper. With the default value of `FOR ALL COLUMNS SIZE AUTO`, Oracle automatically determines which columns require histograms and the number of buckets that will be used based on the column usage information (`DBMS_STATS.REPORT_COL_USAGE`) and the number of distinct values in the column. The column usage information reflects an analysis of all the SQL operations the database has processed for a given object. Column usage tracking is enabled by default.

A column is a candidate for a histogram if it has been seen in a where clause predicate, e.g., an equality, range, `LIKE`, etc. Oracle also verifies if the column data is skewed before creating a histogram. For example, a unique column will not have a histogram created on it if it is only seen in equality predicates. It is strongly recommended you let the `METHOD_OPT` parameter default in the `GATHER_*_STATS` procedures.

**DEGREE parameter**

The `DEGREE` parameter controls the number of parallel server processes that will be used to gather the statistics. By default, Oracle uses the same number of parallel server processes specified as an attribute of the table in the data dictionary (Degree of Parallelism). By default, all tables in an Oracle database have this attribute set to 1, so it may be useful to set this parameter if statistics are being gathered on a large table to speed up statistics collection. By setting the parameter `DEGREE` to `AUTO_DEGREE`, Oracle will automatically determine the appropriate number of parallel server processes that should be used to gather statistics, based on the size of an object. The value can be between 1 (serial execution) for small objects to `DEFAULT_DEGREE (PARALLEL_THREADS_PER_CPU X CPU_COUNT)` for larger objects.

**GRANULARITY parameter**

The `GRANULARITY` parameter dictates the levels at which statistics are gathered on a partitioned table. The possible levels are table (global), partition, or sub-partition. By default Oracle will determine which levels are necessary, based on the table's partitioning strategy. Statistics are always gathered on the first

level of partitioning regardless of the partitioning type used. Sub-partition statistics are gathered when the subpartitioning type is `LIST` or `RANGE`. This parameter is ignored if the table is not partitioned.

#### **CASCADE** parameter

The `CASCADE` parameter determines whether or not statistics are gathered for the indexes on a table. When the `CASCADE` parameter is set to the default, `AUTO_CASCADE`, Oracle will only re-gather statistics for indexes whose table statistics are stale. Cascade is often set to false when a large direct path data load is done and the indexes are disabled. After the load has been completed, the indexes are rebuilt and statistics will be automatically created for them, negating the need to gather index statistics when the table statistics are gathered.

#### **NO\_INVALIDATE** parameter

The `NO_INVALIDATE` parameter determines if dependent cursors (cursors that access the table whose statistics are being re-gathered) will be invalidated immediately after statistics are gathered or not. With the default setting of `DBMS_STATS.AUTO_INVALIDATE`, cursors (statements that have already been parsed) will not be invalidated immediately. They will continue to use the plan built using the previous statistics until Oracle decides to invalidate the dependent cursors based on internal heuristics. The invalidations will happen gradually over time to ensure there is no performance impact on the shared pool or spike in CPU usage as there could be if you have a large number of dependent cursors and all of them were hard parsed at once.

### Changing the default value for the parameters in `DBMS_STATS.GATHER_*_STATS`

You can specify a particular non-default parameter value for an individual `DBMS_STATS.GATHER_*_STATS` command, or override the default value for your database. You can override the default parameter values for `DBMS_STATS.GATHER_*_STATS` procedures using the `DBMS_STATS.SET_*_PREFS` procedures. The list of parameters that can be changed are as follows:

```
AUTOSTATS_TARGET (SET_GLOBAL_PREFS only as it relates to the auto stats job)
CONCURRENT       (SET_GLOBAL_PREFS only)
CASCADE
DEGREE
ESTIMATE_PERCENT
GLOBAL_TEMP_TABLE_STATS
GRANULARITY
INCREMENTAL
INCREMENTAL_LEVEL
INCREMENTAL_STALENESS
METHOD_OPT
NO_INVALIDATE
PUBLISH
STALE_PERCENT
OPTIONS
```

You can override the default settings for each parameter at a table, schema, database, or global level using one of the following `DBMS_STATS.SET_*_PREFS` procedures, with the exception of `AUTOSTATS_TARGET` and `CONCURRENT`, which can only be modified at the global level.

```
SET_TABLE_PREFS
```

```
SET_SCHEMA_PREFS  
SET_DATABASE_PREFS  
SET_GLOBAL_PREFS
```

The `SET_TABLE_PREFS` procedure allows you to change the default values of the parameters used by the `DBMS_STATS.GATHER_*_STATS` procedures for the specified table only.

The `SET_SCHEMA_PREFS` procedure allows you to change the default values of the parameters used by the `DBMS_STATS.GATHER_*_STATS` procedures for all of the existing tables in the specified schema. This procedure actually calls the `SET_TABLE_PREFS` procedure for each of the tables in the specified schema. Since it uses `SET_TABLE_PREFS`, calling this procedure will not affect any new objects created after it has been run. New objects will pick up the `GLOBAL` preference values for all parameters.

The `SET_DATABASE_PREFS` procedure allows you to change the default values of the parameters used by the `DBMS_STATS.GATHER_*_STATS` procedures for all of the user-defined schemas in the database. This procedure actually calls the `SET_TABLE_PREFS` procedure for each table in each user-defined schema. Since it uses `SET_TABLE_PREFS`, this procedure will not affect any new objects created after it has been run. New objects will pick up the `GLOBAL` preference values for all parameters. It is also possible to include the Oracle owned schemas (`sys`, `system`, etc) by setting the `ADD_SYS` parameter to `TRUE`.

The `SET_GLOBAL_PREFS` procedure allows you to change the default values of the parameters used by the `DBMS_STATS.GATHER_*_STATS` procedures for any object in the database that does not have an existing table preference. All parameters default to the global setting unless there is a table preference set, or the parameter is explicitly set in the `GATHER_*_STATS` command. Changes made by this procedure will affect any new objects created after it has been run. New objects will pick up the `GLOBAL_PREFS` values for all parameters.

With `SET_GLOBAL_PREFS` it is also possible to set a default value for two additional parameters, `AUTOSTAT_TARGET` and `CONCURRENT`. `AUTOSTAT_TARGET` controls what objects the automatic statistic gathering job (that runs in the nightly maintenance window) will look after. The possible values for this parameter are `ALL`, `ORACLE`, and `AUTO`. The default value is `AUTO`. A more in-depth discussion about the automatic statistics collection can be found in the statistics management section of this paper.

The `CONCURRENT` parameter controls whether or not statistics will be gathered on multiple tables in a schema (or database), and multiple (sub)partitions within a table concurrently. It is set to `OFF` by default. A more in-depth discussion about concurrent statistics gathering can be found in the *Improving the efficiency of Gathering Statistics* section of this paper.

The `DBMS_STATS.GATHER_*_STATS` procedures and the automatic statistics gathering job obeys the following hierarchy for parameter values; parameter values explicitly set in the command overrule everything else. If the parameter has not been set in the command, we check for a table level preference. If there is no table preference set, we use the `GLOBAL` preference.

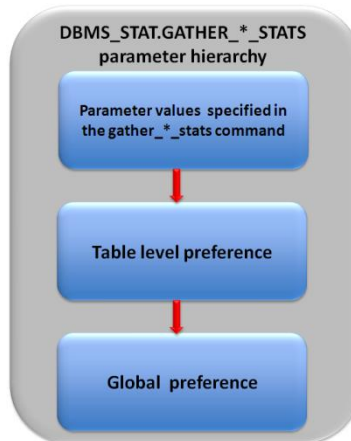


Figure 21. DBMS\_STATS.GATHER\_\*\_STATS hierarchy for parameter values

If you are unsure of what preferences have been set, you can use the `DBMS_STATS.GET_PREFS` function to check. The function takes three arguments; the name of the parameter, the schema name, and the table name. In the example below (figure 22), we first check the value of `STALE_PERCENT` on the `SH.SALES` table. Then we set a table level preference, and check that it took effect using `DBMS_STATS.GET_PREFS`.

```

SQL> SELECT dbms_stats.get_prefs('STALE_PERCENT', 'SH', 'SALES') stale_percent
2 FROM dual;

STALE_PERCENT
-----
10

SQL>
SQL> BEGIN
2 dbms_stats.set_table_prefs('SH', 'SALES', 'STALE_PERCENT', '65');
3 END;
4 /

PL/SQL procedure successfully completed.

SQL>
SQL> SELECT dbms_stats.get_prefs('STALE_PERCENT', 'SH', 'SALES') stale_percent
2 FROM dual;

STALE_PERCENT
-----
65
  
```

Figure 22. Using `DBMS_STATS.SET_PREFS` procedure to change the parameter `stale_percent` for the sales table

### Automatic Statistics Gathering Job

Oracle will automatically collect statistics for all database objects, which are missing statistics or have stale statistics, by running an Oracle AutoTask task during a predefined maintenance window (10pm to 2am weekdays and 6am to 2am at the weekends).

This AutoTask gathers Optimizer statistics by calling the internal procedure `DBMS_STATS.GATHER_DATABASE_STATS_JOB_PROC`. This procedure operates in a very similar

fashion to the `DBMS_STATS.GATHER_DATABASE_STATS` procedure using the `GATHER AUTO` option. The primary difference is that Oracle internally prioritizes the database objects that require statistics; objects which most need updated statistics are processed first. You can verify that the automatic statistics gathering job exists by querying the `DBA_AUTOTASK_CLIENT_JOB` view or through Enterprise Manager (Figure 23). You can also change the maintenance window that the job will run in through Enterprise Manager.

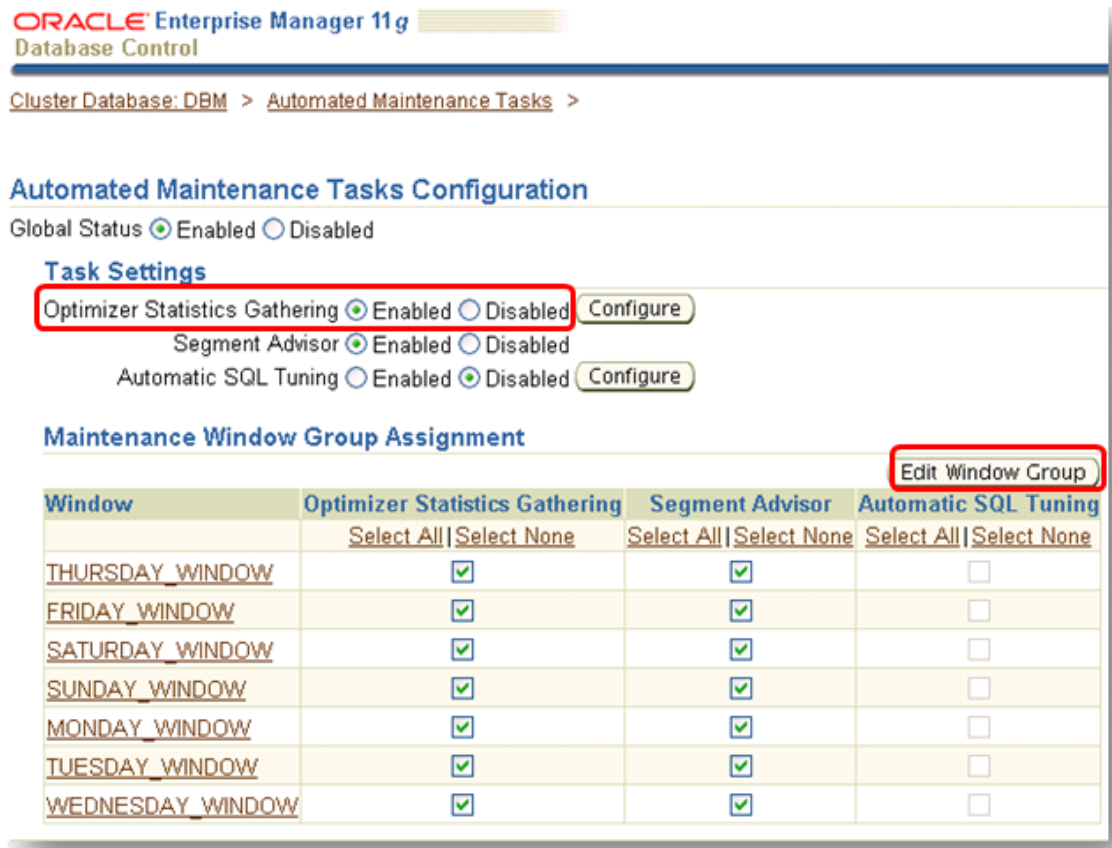


Figure 23. Checking that the automatic statistics gathering job is enabled

Statistics on a table are considered stale when more than `STALE_PERCENT` (default 10%) of the rows are changed (total number of inserts, deletes, updates) in the table. Oracle monitors the DML activity for all tables and records it in the SGA. The monitoring information is periodically flushed to disk, and is exposed in the `*_TAB_MODIFICATIONS` view.

```
SQL> SELECT table_name, inserts, updates, deletes
  2 FROM USER_TAB_MODIFICATIONS
  3 WHERE table_name='PRODUCTS2';
```

TABLE_NAME	INSERTS	UPDATES	DELETES
PRODUCTS2	766	1532	38

Figure 24. Querying USER\_TAB\_MODIFICATIONS view to check DML activity on the PRODUCTS2 table

It is possible to manually flush this data by calling the procedure `DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO` if you want to get up-to-date information at query time (internally the monitoring data is flushed before all statistics collection operations). You can then see which tables have stale statistics by querying the `STALE_STATS` column in the `USER_TAB_STATISTICS` view.

```
SQL> connect hr/hr
Connected.
SQL> SELECT table_name, stale_stats
  2 FROM user_tab_statistics;
```

TABLE_NAME	STALE_STATS
COUNTRIES	NO
DEPARTMENTS	NO
EMPLOYEES	NO
JOBS	
JOB_HISTORY	NO
LOCATIONS	NO
REGIONS	YES

Figure 25. Querying USER\_TAB\_STATISTICS to see if any tables have stale statistics

Tables where `STALE_STATS` is set to `NO`, have up to date statistics. Tables where `STALE_STATS` is set to `YES`, have stale statistics. Tables where `STALE_STATS` is not set are missing statistics altogether.

If you already have a well-established statistics gathering procedure, or if for some other reason you want to disable automatic statistics gathering for your main application schema, consider leaving it on for the dictionary tables. You can do this by changing the value of `AUTOSTATS_TARGET` to `ORACLE` instead of `AUTO` using `DBMS_STATS.SET_GLOBAL_PREFS` procedure.

```
BEGIN
  DBMS_STATS.SET_GLOBAL_PREFS('AUTOSTATS_TARGET',' ORACLE');
END;
/
```

To disable the task altogether:

```
BEGIN
  DBMS_AUTO_TASK_ADMIN.DISABLE(
    client_name => 'auto optimizer stats collection',
    operation => NULL,
    window_name => NULL);
END;
/
```



## Improving the efficiency of Gathering Statistics

Once you define the statistics you are interested in, you want to ensure to collect these statistics in a timely manner. Traditionally, people have sped up statistics gathering by using parallel execution as discussed above. However, what if all of the objects a schema were small and didn't warrant parallel execution, how could you speed up gathering statistics on that schema?

### Concurrent Statistic gathering

Concurrent statistics gathering enables statistics to be gathered on multiple tables in a schema (or database), and multiple (sub)partitions within a table concurrently. Gathering statistics on multiple tables and (sub)partitions concurrently can reduce the overall time it takes to gather statistics by allowing Oracle to fully utilize a multi-processor environment.

Concurrent statistics gathering is controlled by the global preference, `CONCURRENT`, which can be set to `MANUAL`, `AUTOMATIC`, `ALL`, `OFF`. By default it is set to `OFF`. When `CONCURRENT` is enabled, Oracle employs Oracle Job Scheduler and Advanced Queuing components to create and manage multiple statistics gathering jobs concurrently.

Calling `DBMS_STATS.GATHER_TABLE_STATS` on a partitioned table when `CONCURRENT` is set to `MANUAL` or `ALL`, causes Oracle to create a separate statistics gathering job for each (sub)partition in the table. How many of these jobs will execute concurrently, and how many will be queued is based on the number of available job queue processes (`JOB_QUEUE_PROCESSES` initialization parameter, per node on a RAC environment) and the available system resources. As the currently running jobs complete, more jobs will be dequeued and executed until all of the (sub)partitions have had their statistics gathered.

If you gather statistics using `DBMS_STATS.GATHER_DATABASE_STATS`, `DBMS_STATS.GATHER_SCHEMA_STATS`, or `DBMS_STATS.GATHER_DICTIONARY_STATS`, then Oracle will create a separate statistics gathering job for each non-partitioned table, and each (sub)partition for the partitioned tables. Each partitioned table will also have a coordinator job that manages its (sub)partition jobs. The database will then run as many concurrent jobs as possible, and queue the remaining jobs until the executing jobs complete. However, to prevent possible deadlock scenarios multiple partitioned tables cannot be processed simultaneously. Hence, if there are some jobs running for a partitioned table, other partitioned tables in a schema (or database or dictionary) will be queued until the current one completes. There is no such restriction for non-partitioned tables.

The following figure illustrates the creation of jobs at different levels, when a `DBMS_STATS.GATHER_SCHEMA_STATS` command has been issued on the `SH` schema. Oracle will create a statistics gathering job (Level 1 in Figure 26 for each of the non-partitioned tables;

```
CHANNELS,
COUNTRIES,
CUSTOMERS,
PRODUCTS,
PROMOTIONS,
TIMES
```

And, a coordinator job for each partitioned table, i.e., SALES and COSTS, which in turn creates a statistics gathering job for each of partition in SALES and COSTS tables, respectively (Level 2 in Figure 26).

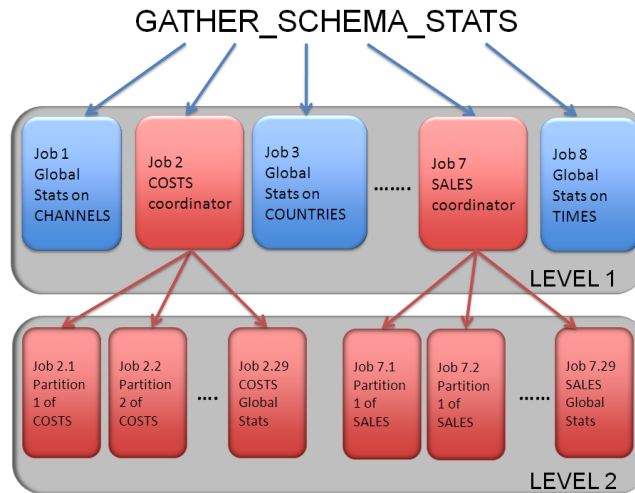


Figure 26. List of the statistics gathering job created when Concurrent Statistics Gathering occurs on the SH schema

Let's assume that the parameter `JOB_QUEUE_PROCESSES` is set to 32. The Oracle Job Scheduler would allow 32 statistics gathering jobs to start, and would queue the rest (assuming that there are sufficient system resources for 32 jobs). Suppose that the first 29 jobs (one for each partition plus the coordinator job) for the `COSTS` table get started, then three non-partitioned table statistics gathering jobs would also be started. The statistics gathering jobs for the `SALES` table will be automatically queued, because only one partitioned table is processed at any one time. As each job finishes, another job will be dequeued and started, until all 64 jobs (6 level 1 jobs and 58 level 2 jobs) have been completed. Each of the individual statistics gathering job can also take advantage of parallel execution as described above under the parameter `DEGREE`.

You should note that if a table, partition, or sub-partition is very small or empty, the database may automatically batch the object with other small objects into a single job to reduce the overhead of job maintenance.

### Configuration and Settings

The concurrency setting for statistics gathering is turned off by default. It can be turned using the `DBMS_STATS.SET_GLOBAL_PREFS` procedure. For example,

```
BEGIN
DBMS_STATS.SET_GLOBAL_PREFS ('CONCURRENT', 'ALL');
END;
/
```

You will also need some additional privileges above and beyond the regular privileges required to gather statistics. The user must have the following Job Scheduler and AQ privileges:

```
CREATE JOB
MANAGE SCHEDULER
```

```
MANAGE ANY QUEUE
```

The SYSAUX tablespace should be online, as the Job Scheduler stores its internal tables and views in SYSAUX tablespace. Finally, the `JOB_QUEUE_PROCESSES` parameter should be set to fully utilize all of the system resources available (or allocated) for the statistics gathering process. If you don't plan to use parallel execution you should set the `JOB_QUEUE_PROCESSES` to 2 X total number of CPU cores (this is a per node parameter in a RAC environment). Please make sure that you set this parameter system-wide (`ALTER SYSTEM ...` or in `init.ora` file) rather than at the session level (`ALTER SESSION`).

If you are going to use parallel execution as part of concurrent statistics gathering you should disable the `PARALLEL_ADAPTIVE_MULTI_USER` initialization parameter. That is;

```
ALTER SYSTEM SET parallel_adaptive_multi_user=false;
```

It is also recommended that you enable parallel statement queuing. This requires Resource Manager to be activated (if not already), and the creation of a temporary resource plan where the consumer group "OTHER\_GROUPS" should have queuing enabled. By default, Resource Manager is activated only during the maintenance windows. The following script illustrates one way of creating a temporary resource plan (`pqq_test`), and enabling the Resource Manager with this plan.

```
BEGIN
  dbms_resource_manager.create_pending_area();
  dbms_resource_manager.create_plan('pqq_test', 'pqq_test');
  dbms_resource_manager.create_plan_directive(
    'pqq_test',
    'OTHER_GROUPS',
    'OTHER_GROUPS directive for pqq',
    parallel_target_percentage => 90);
  dbms_resource_manager.submit_pending_area();
END;
/
ALTER SYSTEM SET RESOURCE_MANAGER_PLAN = 'pqq_test' SID='*';
```

Figure 27. Steps required to setup Resource Manager and parallel statement queuing for concurrent statistics gathering executed in parallel

If you want the automatic statistics gathering job to take advantage of concurrency, set `CONCURRENT` to either `AUTOMATIC` or `ALL`. A new `ORA$AUTOTASK` consumer group has been added to the Resource Manager plan used during the maintenance window, to ensure concurrent statistics gathering does not use too much of the system resources.

## Gathering Statistics on Partitioned tables

Gathering statistics on partitioned tables consists of gathering statistics at both the table level and partition level. Prior to Oracle Database 11g, adding a new partition or modifying data in a few

partitions required scanning the entire table to refresh table-level statistics. If you skipped gathering the global level statistics, the Optimizer would extrapolate the global level statistics based on the existing partition level statistics. This approach is accurate for simple table statistics such as number of rows – by aggregating the individual rowcount of all partitions - but other statistics cannot be determined accurately. For example, it is not possible to accurately determine the number of distinct values for a column (one of the most critical statistics used by the Optimizer) based on the individual statistics of all partitions.

Oracle Database 11g enhanced the statistics collection for partitioned tables with the introduction of incremental global statistics. If the `INCREMENTAL` preference for a partitioned table is set to `TRUE`, the `DBMS_STATS.GATHER_*_STATS` parameter `GRANULARITY` includes `GLOBAL`, and `ESTIMATE_PERCENT` is set to `AUTO_SAMPLE_SIZE`, Oracle will gather statistics on the new partition, and accurately update all global level statistics by scanning only those partitions that have been added or modified, and not the entire table.

Incremental global statistics works by storing a *synopsis* for each partition in the table. A synopsis is statistical metadata for that partition and the columns in the partition. Each synopsis is stored in the `SYSAUX` tablespace. Global statistics are then generated by aggregating the partition level statistics and the synopsis from each partition, thus eliminating the need to scan the entire table to gather table level statistics (see Figure 28). When a new partition is added to the table, you only need to gather statistics for the new partition. The global statistics will be automatically and accurately updated using the new partition synopsis and the existing partitions' synopsis.

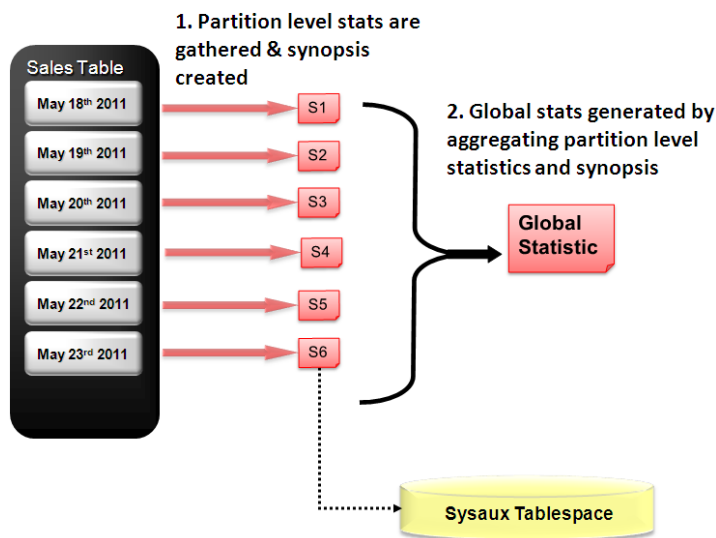


Figure 28. Incremental Statistics gathering on a range partitioned table

Below are the steps necessary to use incremental global statistics.

Begin by switching on incremental statistics at either the table or the global level.

```
BEGIN
DBMS_STATS.SET_TABLE_PREFS ('SH', 'SALES', 'INCREMENTAL', 'TRUE');
```

```
END;
/
```

Gather statistics on the object(s) as normal, letting the `ESTIMATE_PERCENT` and `GRANULARITY` parameters default.

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS ('SH', 'SALES');
END;
/
```

To check the current setting of `INCREMENTAL` for a given table, use `DBMS_STATS.GET_PREFS`.

```
SELECT DBMS_STATS.GET_PREFS ('INCREMENTAL', 'SH', 'SALES')
FROM   dual;
```

By default, when `INCREMENTAL` statistics are enabled, if a single row changes in a partition, then statistics for that partition are considered stale and have to be re-gathered before they can be used to generate global level statistics. This behavior can be changed by setting the new Oracle Database 12c preference `INCREMENTAL_STALENESS`.

By setting `INCREMENTAL_STALENESS` to `USE_STALE_PERCENT`, the partition level statistics will be used as long as the percentage of rows changed is less than the value of the preference `STALE_PERCENTAGE` (10% by default). Alternatively it can be set to `USE_LOCKED_STATS`, which means if statistics on a partition are locked, they will be used to generate global level statistics regardless of how many rows have changed in that partition since statistics were last gathered.

Note that `INCREMENTAL` statistics does not apply to the sub-partitions. Statistics will be gathered as normal on the sub-partitions and on the partitions. Only the partition statistics will be used to determine the global or table level statistics.

## Managing statistics

In addition to collecting appropriate statistics, it is equally important to provide a comprehensive framework for managing them. Oracle offers a number of methods to do this, including the ability to restore statistics to a previous version, the option to transfer statistics from one system to another, or even manually setting the statistics values yourself. These options are extremely useful in specific cases, but are not recommended to replace standard statistics gathering methods using the `DBMS_STATS` package.

### Restoring Statistics

From Oracle Database 10g onwards, when you gather statistics using `DBMS_STATS`, the original statistics are automatically kept as a backup in dictionary tables, and can be easily restored by running `DBMS_STATS.RESTORE_TABLE_STATS` if the newly gathered statistics lead to any kind of problem. The dictionary view `DBA_TAB_STATS_HISTORY` contains a list of timestamps when statistics were saved for each table.

The example below restores the statistics for the table `SALES` to what they were yesterday, and automatically invalidates all of the cursors referencing the `SALES` table in the `SHARED_POOL`. We want

to invalidate all of the cursors; because we are restoring yesterday's statistics and want them to impact any cursor instantaneously. The value of the `NO_INVALIDATE` parameter determines if the cursors referencing the table will be invalidated or not.

```
BEGIN
DBMS_STATS.RESTORE_TABLE_STATS(ownname      => 'SH',
                               tabname       => 'SALES',
                               as_of_timestamp => SYSTIMESTAMP-1
                               force         => FALSE,
                               no_invalidate => FALSE);
END;
/
```

## Pending Statistics

By default, when statistics are gathered, they are published (written) immediately to the appropriate dictionary tables and begin to be used by the Optimizer. In Oracle Database 11g, it is possible to gather Optimizer statistics but not have them published immediately; and instead store them in an unpublished, 'pending' state. Instead of going into the usual dictionary tables, the statistics are stored in pending tables so that they can be tested before they are published. These pending statistics can be enabled for individual sessions, in a controlled fashion, which allows you to validate the statistics before they are published. To activate pending statistics collection, you need to use one of the `DBMS_STATS.SET_*_PREFS` procedures to change value of the parameter `PUBLISH` from `TRUE` (default) to `FALSE` for the object(s) you wish to create pending statistics for.

```
BEGIN
DBMS_STATS.SET_TABLE_PREFS('SH', 'SALES', 'PUBLISH', 'FALSE');
END;
/
```

Gather statistics on the object(s) as normal.

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS('SH', 'SALES');
END;
/
```

The statistics gathered for these objects can be displayed using the dictionary views called `USER_*_PENDING_STATS`. You can tell the Optimizer to use pending statistics by issuing an `ALTER SESSION` command to set the initialization parameter `OPTIMIZER_USE_PENDING_STATS` to `TRUE` and running a SQL workload. For tables accessed in the workload that do not have pending statistics, the Optimizer will use the current statistics in the standard data dictionary tables. Once you have validated the pending statistics, you can publish them using the procedure

```
DBMS_STATS.PUBLISH_PENDING_STATS.

BEGIN
DBMS_STATS.PUBLISH_PENDING_STATS('SH', 'SALES');
END;
/
```

## Exporting / Importing Statistics

One of the most important aspects of rolling out a new application or a new part of an existing application is testing it at scale. Ideally, you want the test system to be identical to production in terms of hardware and data size. This is not always possible, most commonly due to the size of the production environments. By copying the Optimizer statistics from a production database to any other system running the same Oracle version, e.g., a scaled-down test database; you can emulate the Optimizer behavior of a production environment. The production statistics can be copied to the test database using the `DBMS_STATS.EXPORT_*_STATS` and `DBMS_STATS.IMPORT_*_STATS` procedures.

Before exporting statistics, you need to create a table to store the statistics using `DBMS_STATS.CREATE_STAT_TABLE`. After the table has been created, you can export statistics from the data dictionary using the `DBMS_STATS.EXPORT_*_STATS` procedures. Once the statistics have been packed into the statistics table, you can then use `datadump` to extract the statistics table from the production database, and import it into the test database. Once the statistics table is successfully imported into the test system, you can import the statistics into the data dictionary using the `DBMS_STATS.IMPORT_*_STATS` procedures. The following example creates a statistics table called `TAB1` and exports the statistics from the `SH` schema into the `MYSTATS` statistics table.

```

SQL> CREATE OR REPLACE DIRECTORY stats_dir AS '/home/oracle/maria';
Directory created.

SQL>
SQL> BEGIN
2  dbms_stats.create_stat_table('SH','MYSTATS');
3  END;
4  /

PL/SQL procedure successfully completed.

SQL>
SQL> BEGIN
2  dbms_stats.export_schema_stats(ownname=>'SH',stattab=>'MYSTATS');
3  END;
4  /

PL/SQL procedure successfully completed.

SQL>
SQL> exit
Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.2.0 - 64bit Production
With the Partitioning, Real Application Clusters, Automatic Storage Management, OLAP,
Data Mining and Real Application Testing options
[oracle@elca01db08 blog]$ expdp sh/sh tables=MYSTATS directory=STATS_DIR dumpfile=sh_schema_stats.dmp logfile=expdp_sh_stats.log
Export: Release 11.2.0.2.0 - Production on Fri Nov 4 15:14:26 2011
Copyright (c) 1982, 2009, Oracle and/or its affiliates. All rights reserved.
Connected to: Oracle Database 11g Enterprise Edition Release 11.2.0.2.0 - 64bit Production
With the Partitioning, Real Application Clusters, Automatic Storage Management, OLAP,
Data Mining and Real Application Testing options
Starting "SH"."SYS_EXPORT_TABLE_01": sh/***** tables=MYSTATS directory=STATS_DIR dumpfile=sh_schema_stats.dmp logfile=expdp_s
. . exported "SH"."MYSTATS" 1.684 MB 16396 rows
Master table "SH"."SYS_EXPORT_TABLE_01" successfully loaded/unloaded
*****
Dump file set for SH_SYS_EXPORT_TABLE_01 is:
/home/oracle/maria/sh_schema_stats.dmp
Job "SH"."SYS_EXPORT_TABLE_01" successfully completed at 15:15:03
[oracle@elca01db08 maria]$ cd blog
[oracle@elca01db08 blog]$ cd
[oracle@elca01db08 maria]$ impdp sh/sh tables=MYSTATS directory=STATS_DIR dumpfile=sh_schema_stats.dmp logfile=impdp_sh_stats.log
Import: Release 11.2.0.2.0 - Production on Fri Nov 4 17:21:35 2011
Copyright (c) 1982, 2009, Oracle and/or its affiliates. All rights reserved.
Connected to: Oracle Database 11g Enterprise Edition Release 11.2.0.2.0 - 64bit Production
With the Partitioning, Real Application Clusters, Automatic Storage Management, OLAP,
Data Mining and Real Application Testing options
Master table "SH"."SYS_IMPORT_TABLE_01" successfully loaded/unloaded
Starting "SH"."SYS_IMPORT_TABLE_01": sh/***** tables=MYSTATS directory=STATS_DIR dumpfile=sh_schema_stats.dmp logfile=impdp_sh_stats.log
Job "SH"."SYS_IMPORT_TABLE_01" successfully completed at 17:23:51
[oracle@elca01db08 maria]$ sqlplus sh/sh
SQL*Plus: Release 11.2.0.2.0 Production on Fri Nov 4 17:24:04 2011
Copyright (c) 1982, 2010, Oracle. All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.2.0 - 64bit Production
With the Partitioning, Real Application Clusters, Automatic Storage Management, OLAP,
Data Mining and Real Application Testing options
SQL> BEGIN
2  dbms_stats.import_schema_stats(ownname=>'SH',stattab=>'MYSTATS');
3  END;
4  /

PL/SQL procedure successfully completed.

```

Figure 29. Exporting the Optimizer statistics for the SH schema

## Copying Partition Statistics

When dealing with partitioned tables, the Optimizer relies on both the statistics for the entire table (global statistics) as well as the statistics for the individual partitions (partition statistics) to select a good execution plan for a SQL statement. If the query needs to access only a single partition, the Optimizer uses only the statistics of the accessed partition. If the query access more than one partition, it uses a combination of global and partition statistics.

It is very common with range partitioned tables to have a new partition added to an existing table, and rows inserted into just that partition. If end-users start to query the newly inserted data before statistics have been gathered, it is possible to get a suboptimal execution plan due to stale statistics. One of the most common cases occurs when the value supplied in a where clause predicate is outside the domain of values represented by the [minimum, maximum] column statistics. This is known as an 'out-of-range' error. In this case, the Optimizer prorates the selectivity based on the distance between the predicate value, and the maximum value (assuming the value is higher than the max), that is, the farther the value is from the maximum or minimum value, the lower the selectivity will be.



The "Out of Range" condition can be prevented by using the `DBMS_STATS.COPY_TABLE_STATS` procedure (available from Oracle Database 10.2.0.4 onwards). This procedure copies the statistics of a representative source [sub] partition to the newly created and empty destination [sub] partition. It also copies the statistics of the dependent objects: columns, local (partitioned) indexes, etc. The minimum and maximum values of the partitioning column are adjusted as follows;

- If the partitioning type is `HASH` the minimum and maximum values of the destination partition are same as that of the source partition.
- If the partitioning type is `LIST` and the destination partition is a `NOT DEFAULT` partition, then the minimum value of the destination partition is set to the minimum value of the value list that describes the destination partition. The maximum value of the destination partition is set to the maximum value of the value list that describes the destination partition
- If the partitioning type is `LIST` and the destination partition is a `DEFAULT` partition, then the minimum value of the destination partition is set to the minimum value of the source partition. The maximum value of the destination partition is set to the maximum value of the source partition
- If the partitioning type is `RANGE`, then the minimum value of the destination partition is set to the high bound of previous partition and the maximum value of the destination partition is set to the high bound of the destination partition unless the high bound of the destination partition is `MAXVALUE`, in which case the maximum value of the destination partition is set to the high bound of the previous partition

It can also scale the statistics (such as the number of blocks, or number of rows) based on the given `scale_factor`. The following command copies the statistics from `SALES_Q3_2011` range partition to the `SALES_Q4_2011` partition of the `SALES` table and scales the basic statistics by a factor of 2.

```
BEGIN
DBMS_STATS.COPY_TABLE_STATS ('SH', 'SALES', 'SALES_Q3_2011', 'SALES_Q4_2011', 2);
END;
/
```

Index statistics are only copied if the index partition names are the same as the table partition names (this is the default). Global or table level statistics are not updated by default. The only time global level statistics would be impacted by the `DBMS_STATS.COPY_TABLE_STATS` procedure would be if no statistics existed at the global level and global statistics were being generated via aggregation.

## Comparing Statistics

One of the key reasons an execution plan can differ from one system to another is because the Optimizer statistics on each system are different - for example when data on a test system is not 100% in sync with real production system. To identify differences in statistics, the `DBMS_STATS.DIFF_TABLE_STATS_*` functions can be used to compare statistics for a table from two different sources. The statistic sources can be:

- A user statistics table and the current statistics in the data dictionary
- A single user statistics table containing two sets of statistics that can be identified using `statids`

- Two different user statistics tables
- Two points in history
- Current statistics and a point in history
- Pending Statistics with the current statistics in the dictionary
- Pending Statistics with a user statistics table

The function also compares the statistics of the dependent objects (indexes, columns, partitions), and displays all the statistics for the object(s) from both sources if the difference between the statistics exceeds a specified threshold. The threshold can be specified as an argument to the function; the default value is 10%. The statistics corresponding to the first source will be used as the basis for computing the differential percentage.

In the example below, we compare the current dictionary statistics for the EMP table with the statistics for EMP in the statistics table TAB1; the SQL statement will generate a report as shown in Figure 30.

```
SELECT report, maxdiffpct
FROM table(DBMS_STATS.DIFF_TABLE_STATS_IN_STATTAB('SCOTT','EMP','TAB1' ));
```

```
DBMS_STATS.DIFF_TABLE_STATS_IN_STATTAB(NULL,'EMP','TAB1')
#####
STATISTICS DIFFERENCE REPORT FOR:
TABLE      : EMP
OWNER      : SCOTT
SOURCE A   : User statistics table TAB1
             : Statid      :
             : Owner      : SCOTT
SOURCE B   : Current Statistics in dictionary
PCTTHRESHOLD : 10
~~~~~
NO DIFFERENCE IN TABLE / (SUB)PARTITION STATISTICS
~~~~~
COLUMN STATISTICS DIFFERENCE:
COL_NAME SRC NDV DENSITY HISTOGRAM NULLS LEN MIN MAX SIZE
DEPTNO   A  3  .333333333 NO      0    3 C10B C11F 14
         B  3  .035714285 YES     0    3 C10B C11F 14
```

Figure 30. Report output after comparing the statistics for table SCOTT.EMP in the statistics table TAB1 and the current statistics in the dictionary.

## Locking Statistics

In some cases, you may want to prevent any new statistics from being gathered on a table or schema by locking the statistics. Once statistics are locked, no modifications can be made to those statistics until the statistics have been unlocked or unless the `FORCE` parameter of the `GATHER_*_STATS` procedures has been set to `TRUE`.

```

SQL> BEGIN
  2 dbms_stats.lock_table_stats('SH','SALES');
  3 END;
  4 /

PL/SQL procedure successfully completed.

SQL> BEGIN
  2 dbms_stats.gather_table_stats('SH','SALES');
  3 END;
  4 /
BEGIN
*
ERROR at line 1:
ORA-20005: object statistics are locked (stattype = ALL)
ORA-06512: at "SYS,DBMS_STATS", line 23154
ORA-06512: at "SYS,DBMS_STATS", line 23205
ORA-06512: at line 2

SQL>
SQL> BEGIN
  2 dbms_stats.gather_table_stats('SH','SALES',FORCE=>TRUE);
  3 END;
  4 /

PL/SQL procedure successfully completed.

```

Figure 31 Locking and unlocking table statistics

Statistics can be locked and unlocked at either the table or partition level.

```

BEGIN
DBMS_STATS.LOCK_PARTITION_STATS('SH','SALES', 'SALES_Q3_2000');
END;

```

You should note there is a hierarchy with locked statistics. For example, if you lock the statistic on a partitioned table, and then unlocked statistics on just one partition in order to re-gather statistics on that one partition, it will fail with an error ORA-20005. The error occurs because the table level lock will still be honored even though the partition has been unlocked. The statistics gather for the partition will only be successfully if the `FORCE` parameter is set to `TRUE`.

```

SQL> exec DBMS_STATS.COPY_TABLE_STATS('SH', 'SALES', 'SALES_Q3_2002','SALES_Q4_2002',2);
PL/SQL procedure successfully completed.

SQL>
SQL>
SQL> BEGIN
  2  dbms_stats.lock_table_stats('SH','SALES');
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL> BEGIN
  2  dbms_stats.unlock_partition_stats('SH','SALES','SALES_Q4_2002');
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL> BEGIN
  2  dbms_stats.gather_table_stats('SH','SALES','SALES_Q4_2002');
  3  END;
  4  /
BEGIN
*
ERROR at line 1:
ORA-20005: object statistics are locked (stattype = ALL)
ORA-06512: at "SYS.DBMS_STATS", line 23154
ORA-06512: at "SYS.DBMS_STATS", line 23205
ORA-06512: at line 2

SQL> BEGIN
  2  dbms_stats.gather_table_stats('SH','SALES','SALES_Q4_2002',FORCE=>TRUE);
  3  END;
  4  /

PL/SQL procedure successfully completed.

```

Figure 32. Hierarchy with locked statistics; table level lock trumps partition level unlock

## Manually setting Statistics

Under rare circumstances, it may be beneficial to manually set the Optimizer statistics in the data dictionary. One such example could be a highly volatile global temporary table (note that while manually setting statistics is discussed in this paper, it is not generally recommended, because inaccurate or inconsistent statistics can lead to poor performing execution plans). Statistics can be manually set using `DBMS_STATS.SET_*_STATS` procedures.

## Other Types of Statistics

In addition to basic table, column, and index statistics, the Optimizer uses additional information to determine the execution plan of a statement. This additional information can come in the form of dynamic sampling and system statistics.

### Dynamic Statistics (previously known as dynamic sampling)

Dynamic sampling was introduced in Oracle Database 9i Release 2 to collect additional statement-specific object statistics during the optimization of a SQL statement. The most common misconception is that dynamic sampling can be used as a substitute for Optimizer statistics. The goal of dynamic sampling is to augment the existing statistics; it is used when regular statistics are not sufficient to get good quality cardinality estimates.

In Oracle Database 12c, dynamic sampling has been enhanced to become dynamic statistics. Dynamic statistics allow the optimizer to augment existing statistics to get more accurate cardinality estimates for not only single table accesses, but also joins and group-by predicates

So, how and when will dynamic statistics be used? During the compilation of a SQL statement, the Optimizer decides whether to use dynamic statistics or not by considering whether the available statistics are sufficient to generate a good execution plan. If the available statistics are not enough, dynamic sampling will be used. It is typically used to compensate for missing or insufficient statistics that would otherwise lead to a very bad plan. For the case where one or more of the tables in the query does not have statistics, dynamic sampling is used by the Optimizer to gather basic statistics on these tables before optimizing the statement. The statistics gathered in this case are not as high a quality or as complete as the statistics gathered using the `DBMS_STATS` package. This trade off is made to limit the impact on the compile time of the statement.

The second scenario where dynamic statistics can be used is when the statement contains a complex predicate expression, and extended statistics are not available, or cannot be used. For example, if you had a query that has non-equality where clause predicates on two correlated columns, standard statistics would not be sufficient in this case, and extended statistics could not be used. In this simple query against the `SALES` table, the Optimizer assumes that each of the where clause predicates will reduce the number of rows returned by the query, and based on the standard statistics, determines the cardinality to be 20,197, when in fact, the number of rows returned is ten times higher at 210,420.

```
SELECT count (*)
FROM   sh.Sales
WHERE  cust_id < 2222
AND    prod_id > 5;
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				528 (100)	
1	SORT AGGREGATE		1	9		
2	PARTITION RANGE ALL		20197	177K	528 (2)	00:00:07
* 3	TABLE ACCESS STORAGE FULL	SALES	20197	177K	528 (2)	00:00:07

Predicate Information (identified by operation id):

```

3 - storage(("CUST_ID"<2222 AND "PROD_ID">5))
   filter(("CUST_ID"<2222 AND "PROD_ID">5))

```

Figure 33. Execution plan for complex predicates without dynamic sampling

With standard statistics, the Optimizer is not aware of the correlation between the CUST\_ID and PROD\_ID in the SALES table. By setting OPTIMIZER\_DYNAMIC\_SAMPLING to level 6, the Optimizer will use dynamic sampling to gather additional information about the complex predicate expression. The additional information provided by dynamic statistics allows the Optimizer to generate a more accurate cardinality estimate, and therefore a better performing execution plan.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				528 (100)	
1	SORT AGGREGATE		1	9		
2	PARTITION RANGE ALL		206K	1813K	528 (2)	00:00:07
* 3	TABLE ACCESS STORAGE FULL	SALES	206K	1813K	528 (2)	00:00:07

Predicate Information (identified by operation id):

```

3 - storage(("CUST_ID"<2222 AND "PROD_ID">5))
   filter(("CUST_ID"<2222 AND "PROD_ID">5))

```

Note

dynamic sampling used for this statement (level=6)

Figure 34. Execution plan for complex predicates with dynamic sampling level 6

As seen in this example, dynamic sampling is controlled by the parameter OPTIMIZER\_DYNAMIC\_SAMPLING, which can be set to different levels (0-11). These levels control two different things; when dynamic sampling kicks in, and how large a sample size will be used to gather the statistics. The greater the sample size, the bigger impact dynamic sampling has on the compilation time of a query.

When set to 11 the Optimizer will automatically decide if dynamic statistics will be useful, and what dynamic sampling level will be used for SQL statements. The optimizer bases its decision to use dynamic statistics on the complexity of the predicates used, the existing base statistics, and the total execution time expected for the SQL statement. For example, dynamic statistics will kick in for situations where the Optimizer previously would have used a guess. For example, queries with LIKE predicates and wildcards.

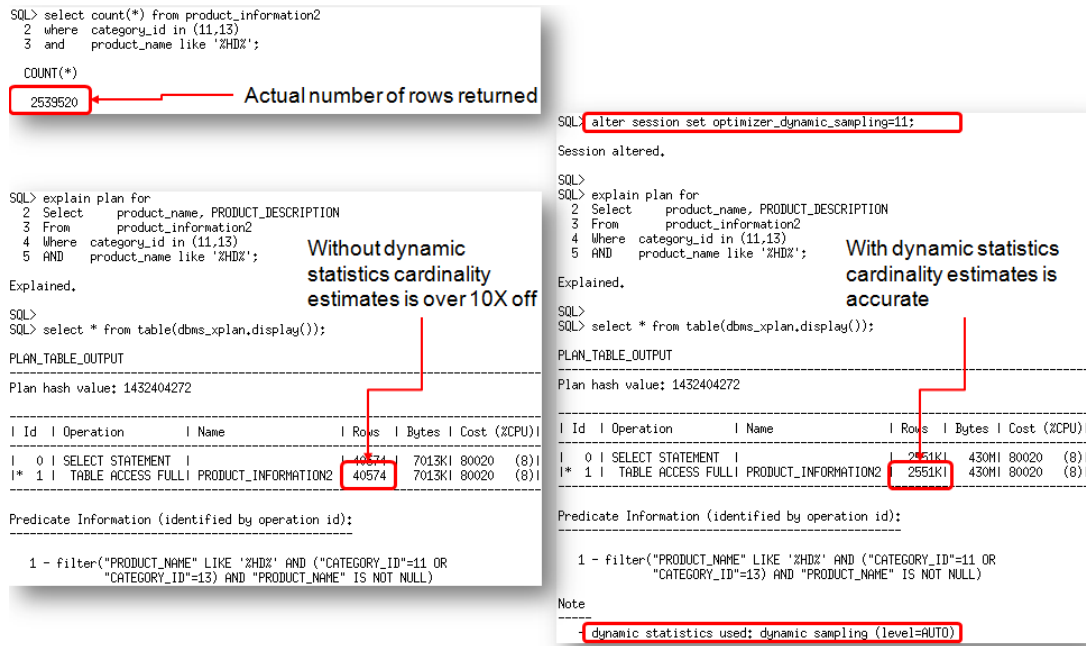


Figure 35. When OPTIMIZER\_DYNAMIC\_SAMPLING is set to level 11 dynamic sampling will be used instead of guesses

Given these criteria, it's likely that when set to level 11, dynamic sampling will kick-in more often than it did before. This will extend the parse time of a statement. In order to minimize the performance impact, the results of the dynamic sampling queries will be persisted in the cache, as dynamic statistics, allowing other SQL statements to share these statistics.

### System statistics

In Oracle Database 9i, system statistics were introduced to enable the Optimizer to more accurately cost each operation in an execution plan by using information about the actual system hardware executing the statement, such as CPU speed and IO performance.

System statistics are enabled by default, and are automatically initialized with default values; these values are representative for most system. When system statistics, are gathered they will override these initial values. To gather system statistics, you can use DBMS\_STATS.GATHER\_SYSTEM\_STATS during a representative workload time window, ideally at peak workload times.

System statistics need to be gathered only once. System statistics are not automatically collected as part of the automatic statistics gathering job. You must have GATHER\_SYSTEM\_STATISTICS or the DBA role to update system statistics.

### Statistics on Dictionary Tables

Since the Cost Based Optimizer is now the only supported optimizer, all tables in the database need to have statistics, including all of the dictionary tables (tables owned by 'SYS', SYSTEM, etc, and residing in the system and SYSAUX tablespace). Statistics on the dictionary tables are maintained via the

automatic statistics gathering job run during the nightly maintenance window. If you choose to switch off the automatic statistics gathering job for your main application schema, consider leaving it on for the dictionary tables. You can do this by changing the value of `AUTOSTATS_TARGET` to `ORACLE` instead of `AUTO` using the procedure `DBMS_STATS.SET_GLOBAL_PREFS`.

```
BEGIN
DBMS_STATS.SET_GLOBAL_PREFS ('AUTOSTATS_TARGET', 'ORACLE');
END;
/
```

Statistics can be manually gathered on the dictionary tables using the `DBMS_STATS.GATHER_DICTIONARY_STATS` procedure. You must have both the `ANALYZE ANY DICTIONARY`, and `ANALYZE ANY` system privilege, or the `DBA` role to update dictionary statistics. It is recommended that dictionary table statistics be maintained on a regular basis in a similar manner to user schemas.

### Statistics on Fixed Objects

You will also need to gather statistics on dynamic performance tables and their indexes (fixed objects). These are the `X$` tables on which the `V$` views (`V$SQL` etc.) are built. Since `V$` views can appear in SQL statements like any other user table or views, it is important to gather optimizer statistics on these tables to help the optimizer generate good execution plans. However, unlike other database tables, dynamic sampling is not automatically used for SQL statement involving `X$` tables when optimizer statistics are missing. The Optimizer uses predefined default values for the statistics if they are missing. These defaults may not be representative and could potentially lead to a suboptimal execution plan, which could cause severe performance problems in your system. It is for this reason that we strongly recommend you gather fixed objects statistics.

Fixed object statistics are not gathered or maintained by the automatic statistics gathering job. You can collect statistics on fixed objects using `DBMS_STATS.GATHER_FIXED_OBJECTS_STATS` procedure.

```
BEGIN
DBMS_STATS.GATHER_FIXED_OBJECTS_STATS;
END;
/
```

The `DBMS_STATS.GATHER_FIXED_OBJECTS_STATS` procedure gathers the same statistics as `DBMS_STATS.GATHER_TABLE_STATS` except for the number of blocks. `Blocks` is always set to 0 since the `x$` tables are in memory structures only and are not stored on disk. Because of the transient nature of the `x$` tables, it is important that you gather fixed object statistics when there is a representative workload on the system. You must have the `ANALYZE ANY DICTIONARY` system privilege or the `DBA` role to update fixed object statistics. It is recommended that you re-gather fixed object statistics if you do a major database or application upgrade.



## Conclusion

In order for the Cost Based Optimizer to accurately determine the cost for an execution plan, it must have information about all of the objects (table and indexes) accessed in the SQL statement, and information about the system on which the SQL statement will be run. This necessary information is commonly referred to as Optimizer statistics. Understanding and managing statistics is key to optimal SQL execution. Knowing when and how to gather statistics in a timely manner is critical to maintaining good performance.

By using a combination of the automatic statistics gathering job and the `DBMS_STATS` package, a DBA can maintain an accurate set of statistics for a system, ensuring the Optimizer will have the best possible source of information to determine the execution plan.



Understanding Optimizer Statistics with Oracle  
Database 12c

June 2013

Author: Maria Colgan

Oracle Corporation  
World Headquarters  
500 Oracle Parkway  
Redwood Shores, CA 94065  
U.S.A.

Worldwide Inquiries:  
Phone: +1.650.506.7000  
Fax: +1.650.506.7200

oracle.com



Oracle is committed to developing practices and products that help protect the environment

Copyright © 2012, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0612

**Hardware and Software, Engineered to Work Together**