

18^c ORACLE[®]
Database

Understanding Optimizer Statistics With Oracle Database 18c

ORACLE WHITE PAPER | FEBRUARY 2018





Table of Contents

Introduction	1
What are Optimizer Statistics?	2
Statistics on Partitioned Tables	12
Managing Statistics	16
Other Types of Statistics	22
Conclusion	27
References	28



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 critical for achieving optimal SQL execution. This whitepaper is the first in a two part series on optimizer statistics and describes the core concepts of what statistics are and what types are statistics are used by the Oracle Optimizer. The second paper in the series (*Best Practices for Gathering Optimizer Statistics with Oracle Database 18c*) covers how to keep optimizer statistics up-to-date so that they accurately represent the data that's stored in the database.

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`.

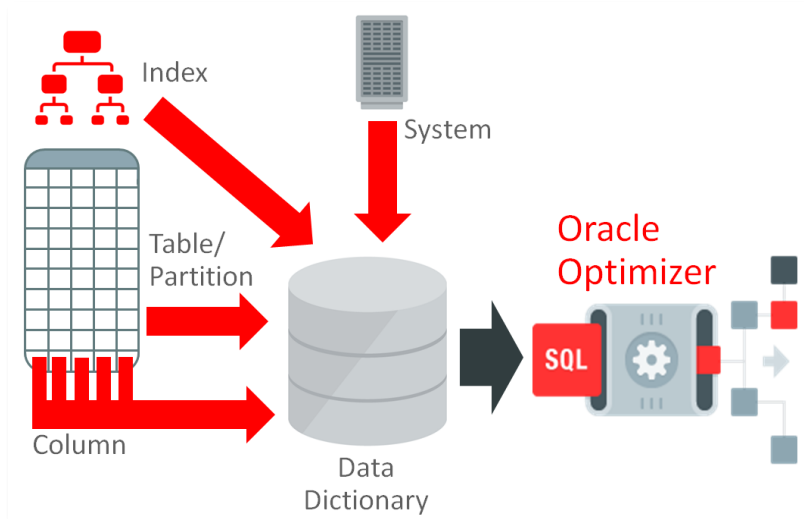


Figure 1: Optimizer Statistics stored in the data dictionary are used by the Oracle Optimizer to determine execution plans

Most types of optimizer statistics need be *gathered* or *refreshed* periodically to ensure that they accurately reflect the nature of the data that's stored in the database. If, for example, the data in the database is highly volatile (perhaps there are tables that are rapidly and continuously populated) then it will be necessary to gather statistics more frequently than if the data is relatively static. Database administrators can choose to use manual or automatic processes to gather statistics and this topic is covered in the second paper of this series¹.

Table and Column Statistics

Table statistics include information such as 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 – the number of rows

¹ Oracle White Paper: Best Practices for Gathering Optimizer Statistics with Oracle Database 12c Release 2

returned - to be the number of rows in the table divided by the number of distinct values for the column or $100/10 = 10$.

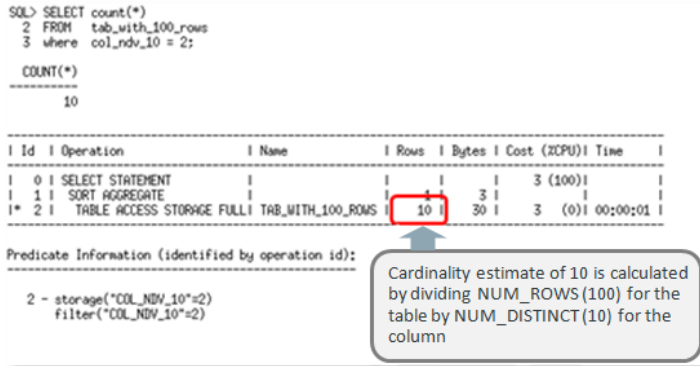


Figure 2: Cardinality calculation using basic table and column statistics

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. This type of information can be gathered by using extensions to basic statistics. These extensions are histograms, column groups, and expression statistics. Without them, the optimizer will assume uniform data value distribution and no correlations between columns values.

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 a more accurate cardinality, and allows it therefore 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. The appropriate histogram type is chosen automatically by the Oracle database. This decision is based on the number of distinct values in the column. From Oracle Database 12c onwards, height-balance histograms will be replaced by hybrid histograms². The data dictionary view `user_tab_col_statistics` has column called "histogram". It reports what type of histogram is present on any particular table column.

² Assuming the parameter `ESTIMATE_PERCENT` parameter is "AUTO_SAMPLE_SIZE" in the `DBMS_STATS.GATHER_*_STATS` command used to gather the statistics. This is the default.

Frequency Histograms

Frequency histograms are created when the number of distinct values in the column is less than the maximum number of buckets allowed. This is 254 by default, but it can be modified using DBMS_STATS procedures up to a maximum of 2048 (beginning with Oracle Database 12c).

Figure 3 demonstrates how the data distribution of the column “COL1” can be encoded and represented using frequency histograms. Chart 1 (on the left) has a column or “bucket” for each value and the height of the column represents the COUNT of that particular value. Oracle’s frequency histograms refer to the buckets as “endpoint values” and the *cumulative* frequency is stored as the “endpoint number” (see Chart 2). In this case, the “40” endpoint stores 10. Next, the “51” endpoint stores 10+1 (11) followed by “52”, which stores 11+2 (13) and so on.

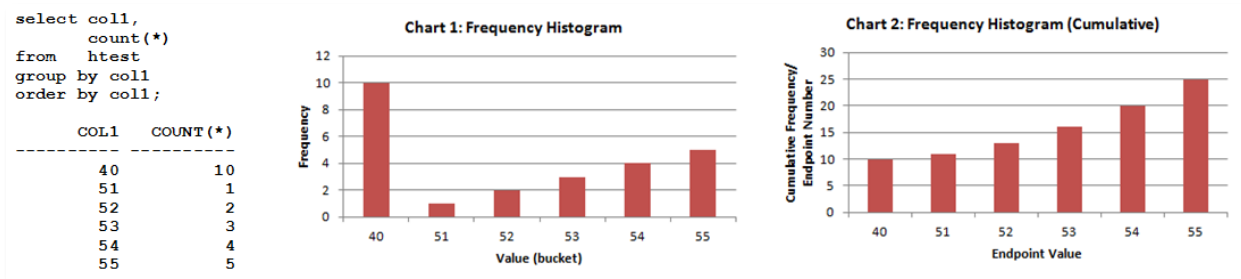


Figure 3: Representing data distributions using histograms

Figure 4 shows how this histogram can be viewed in the data dictionary. Compare the ENDPOINT_VALUE and ENDPOINT_NUMBER with Chart 2 and FREQUENCY with Chart 1.

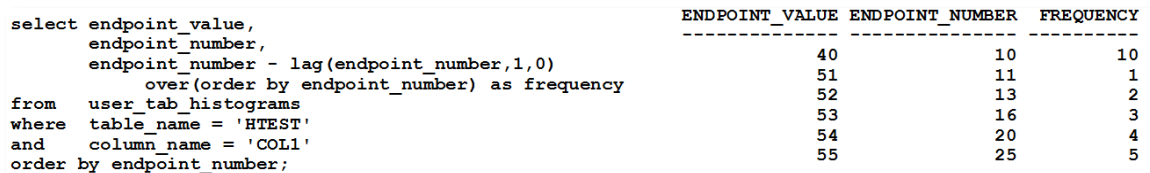


Figure 4: Viewing histograms in the data dictionary.

Note that endpoint values are numeric, so histograms on columns with non-numeric datatypes have their values encoded to a number.

Once the histogram has been created, the optimizer can use it to estimate cardinalities more accurately. For example, if there is a predicate “WHERE val = 53”, it is easy to see that the histogram can be used to establish (using endpoint value “53”) that the cardinality will be 3. Predicates for values that aren’t present in the frequency histogram (such as “WHERE val=41”) will have an estimated cardinality of 1. That is the lowest possible number chosen by the Optimizer for cost calculations, chosen for both “one row returned” and “zero rows returned”.

Height Balanced Histograms

Prior to Oracle Database 12c, height-balanced histograms are created when the number of distinct values is greater than the maximum number of buckets allowed (this is 254 by default, but it can be modified using DBMS_STATS procedures). Frequency histograms become less useful for larger distinct value counts because they store a single value per histogram bucket and the system overhead of storing, using and maintaining histograms becomes more significant as the number of histogram buckets increase. From Oracle Database 12c onwards, hybrid histograms are created instead of height-balanced histograms in most cases. Hybrid histograms will be discussed in a later section.

Consider a table with a column, “COL1” that has values ranging from 1 to 1000. Figure 5 shows how we could represent the data distribution by dividing up the range 1 to 1000 into a set of identical ranges (10 ranges, each representing a range of 100 values). There are 1,000 rows with values between 1 and 100 and a spike in row count (3,000) for values between 201 and 300:

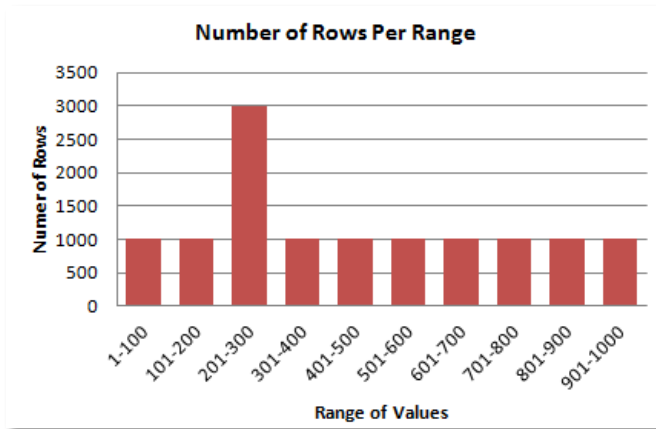


Figure 5: A frequency histogram

We can encode exactly the same information in Figure 5 using a “height-balanced” histogram, shown in Figure 6. By adjusting the size of the ranges we can contrive to make each range represent a similar number of rows. In other words, each of the vertical “bars” has a similar height. For example, the range 1-200 has 2,000 rows. The next range has to be narrower (201-267) to keep the row count at around 2,000 because there are more rows with values 201 to 300 (as can be seen more easily, above).

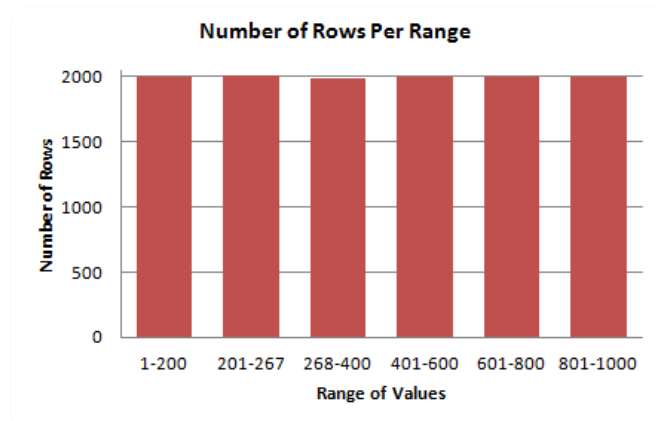


Figure 6: A height-balanced histogram created using the same data as Figure 5

The details of the histogram can be queried from the data dictionary. This example is slightly contrived because the number of buckets was chosen to be 6, but this was simply to keep the following listings short. See Figure 7 and note the presence of endpoint number “0”, indicating the minimum column value of “1”:

```

select histogram,num_buckets
from user_tab_col_statistics
where table_name = 'HTEST'
and column_name = 'COL1';

select endpoint_number,
       endpoint_value
from user_histograms
where table_name = 'HTEST'
and column_name = 'COL1'
order by endpoint_number;

```

HISTOGRAM	NUM_BUCKETS
HEIGHT BALANCED	6

ENDPOINT_NUMBER	ENDPOINT_VALUE
0	1
1	200
2	267
3	400
4	600
5	800
6	1000

Figure 7: Viewing height-balanced histogram details.

In many cases, the endpoint numbers will not be continuous. This is because the optimizer may choose to reduce the total number of buckets by “compressing” multiple buckets into a single bucket. In the following example, endpoint number 3 is not present (values 500 and 501 are very common):

ENDPOINT_NUMBER	ENDPOINT_VALUE
0	1
1	434
2	500
4	501
5	567
6	1000

Figure 8: Compressing buckets.

If a column value in the table is very common, it may span multiple histogram buckets. The reason for this is that, for this type of histogram, all buckets must contain approximately the same number of rows but a common or “popular” column value may be so common that its frequency exceeds the maximum number of rows per bucket. Column values like this are, in fact, called *popular values* and they are handled differently by the optimizer to make allowances for their popularity. However, in some cases, a value may span *nearly* two buckets, and these are called *almost popular values*. The costing for nearly popular values can be inaccurate, which is the reason why Oracle Database 12c introduced hybrid histograms (covered later) to address this limitation.

Top-Frequency Histograms

Traditionally, if a column has more distinct values than the number of buckets available (254 by default), a height-balanced or hybrid histogram would be created. However, there are situations where most of the rows in the table have a small number of distinct values, and remaining rows (with a large number of distinct values) make up a very small proportion of the total. In these circumstances, it can be appropriate to create a frequency histogram for the majority of the rows in the table and ignore the statistically insignificant set of rows (the ones with low cardinality and a high number of distinct values). To choose a frequency histogram, the database must decide if “n” histogram buckets is enough to calculate cardinality accurately even though the number of distinct values in the column exceeds “n”. It does this by counting how many distinct values there in the top 99.6% of rows in the table for the column in question (99.6% is the default, but this value is adjusted to take into account the number of histogram

buckets available). If there are enough histogram buckets available to accommodate the top-n distinct values, then a frequency histogram is created for these popular values.

A top-frequency histogram is only created if the `ESTIMATE_PERCENT` parameter of the gather statistics command is set to `AUTO_SAMPLE_SIZE` (the default), because 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 fewer distinct values).

Take, for example, a `PRODUCT_SALES` table, which contains sales information for a Christmas ornaments company. The table has 1.78 million rows and 632 distinct `TIME_IDs`. But the majority of the rows in `PRODUCT_SALES` have less than 254 distinct `TIME_IDs`, 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 9 illustrates the idea behind a top-frequency histogram. You can see in principle that it is possible to identify the most significant data (within the dotted lines) and use it to construct a frequency histogram with those values.

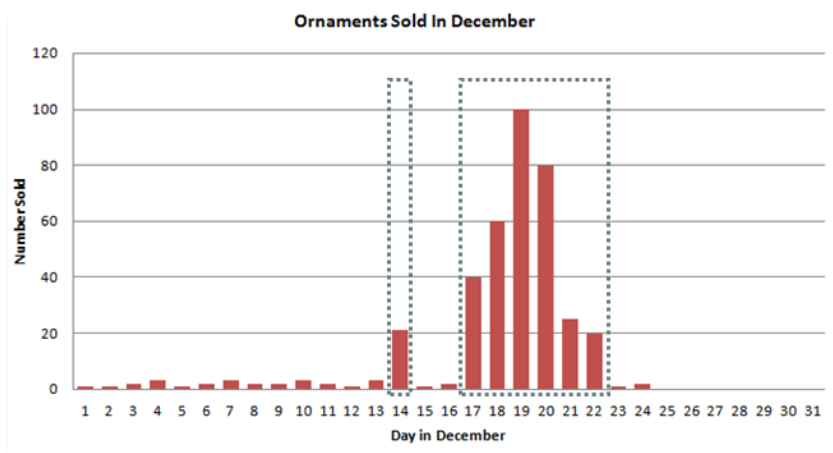


Figure 9: A Top-Frequency Histogram

Hybrid Histograms

One prominent problem with height balanced histograms is that a popular value might span two buckets, but it may only appear as the end point value of one bucket. This is the “almost popular value” scenario described above in the “height-balanced histograms” section.

To resolve this limitation, Oracle Database 12c introduced a hybrid histogram that 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.

Figure 10 shows how the hybrid histogram has characteristics of both frequency and height-balanced histograms; a bucket can contain multiple values and the endpoint number stores the cumulative frequency.

```

select histogram,num_buckets
from user_tab_col_statistics
where table_name = 'HTEST'
and column_name = 'COL1';

HISTOGRAM      NUM_BUCKETS
-----
HYBRID                6

select endpoint_value,
       endpoint_number,
       endpoint_repeat_count,
       endpoint_number - lag(endpoint_number,1,0)
       over(order by endpoint_number) as frequency
from user_tab_histograms
where table_name = 'HTEST'
and column_name = 'COL1'
order by endpoint_number;

ENDPOINT_NUMBER  ENDPOINT_VALUE  ENDPOINT_REPEAT_COUNT  FREQUENCY
-----
                10                1                10                10
                2420               214                30               2410
                4820               294                30               2400
                7220               522                10               2400
                9620               762                10               2400
               12000              1000                10               2380

```

Figure 10: Hybrid histogram.

The endpoint repeat count indicates how many times the endpoint value is repeated. For example, compare the following results with endpoint values in Figure 10:

```

SQL> select count(*) from htest where col1 = 214;

COUNT(*)
-----
        30

SQL> select count(*) from htest where col1 = 522;

COUNT(*)
-----
        10

```

Figure 11: Understanding the endpoint repeat count.

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 (`ESTIMATE_PERCENT => DBMS_STATS.AUTO_SAMPLE_SIZE`).

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:

```
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 for predicates like `country_id='US'` and `cust_state_province='CA'` when they 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 beneficial 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:

Step1: 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 procedure, `DBMS_STATS.SEED_COL_USAGE`, should be used it 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.

```
begin
  dbms_stats.seed_col_usage(null,null,300);
end;
/
```

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:

```
select dbms_stats.report_col_usage(user,'CUSTOMERS') from dual;
```

```
DBMS_STATS.REPORT_COL_USAGE(USER, 'CUSTOMERS')
-----
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 SH.CUSTOMERS
.....
1. (COUNTRY ID, CUST STATE PROVINCE) : FILTER
#####
```

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.

Step 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.

```
select dbms_stats.create_extended_stats(user,'CUSTOMERS') from dual;
```

```
DBMS_STATS.CREATE_EXTENDED_STATS(USER,'CUSTOMERS')
-----
#####
EXTENSIONS FOR SH.CUSTOMERS
.....
1. (COUNTRY_ID, CUST_STATE_PROVINCE) : SYS STUM4KJU$CCIC9C1UJ6UWC4YP created
#####
```

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

Step 3: Re-gather Statistics

The final step is to re-gather statistics on the affected tables so that the newly created column groups will have statistics created for them:

```
exec dbms_stats.gather_table_stats(null,'CUSTOMERS')
```

SQL Plan Directives and Column Groups

SQL plan directives³ are not only used to optimizer SQL execution plans, they are used by Oracle to determine if column groups would be useful to resolve cardinality misestimates. If a SQL plan directive is created, and the optimizer decides that a cardinality misestimate would be resolved with a column group, then Oracle Database Release 1 will automatically create the column group the next time statistics are gathered on the appropriate table.

This behavior has changed from Oracle Database 12c Release 2 onwards. Automatic column group creation is OFF by default and is controlled by a `DBMS_STATS` preference called `AUTO_STAT_EXTENSIONS`. This is how the preference is used:

SQL plan directives will not be used to create column groups automatically (this is the default):

```
exec dbms_stats.set_global_prefs ('AUTO_STAT_EXTENSIONS', 'OFF');
```

SQL plan directives will be used to create column groups automatically:

```
exec dbms_stats.set_global_prefs ('AUTO_STAT_EXTENSIONS', 'ON');
```

³ See Oracle white paper: *Optimizer with Oracle Database 12c Release 2*

Expression Statistics

It is 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)`:

```
select dbms_stats.create_extended_stats(NULL, 'CUSTOMERS', 'UPPER(CUST_LAST_NAME)')
from dual;
```

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

Figure 15: 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. 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).

Statistics on Partitioned Tables

Statistics on partitioned tables must be calculated 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 synopses from each partition, thus eliminating the need to scan the entire table to gather table level statistics (see Figure 16). 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' synopses.

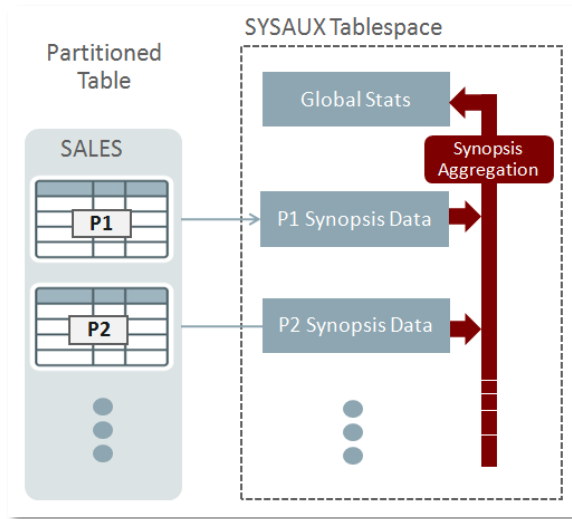


Figure 16: Incremental statistics gathering on a range partitioned table.

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. Below are the steps necessary to use incremental global statistics.

Begin by switching on incremental statistics:

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

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

```
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;
```

Incremental Statistics and Staleness

In Oracle Database 11g, if incremental statistics were enabled on a table and a single row changed in one of the partitions, then statistics for that partition were considered stale and had to be re-gathered before they could be used to generate global level statistics.

In Oracle Database 18c a preference called `INCREMENTAL_STALENESS` allows you to control when partition statistics will be considered stale and not good enough to generate global level statistics. By default, `INCREMENTAL_STALENESS` is set to `NULL`, which means partition level statistics are considered stale as soon as a single row changes (same as 11g).

Alternatively, it can be set to `USE_STALE_PERCENT` or `USE_LOCKED_STATS`. `USE_STALE_PERCENT` means 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). `USE_LOCKED_STATS` 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.

Incremental Statistics and Partition Exchange Loads

One of the benefits of partitioning is the ability to load data quickly and easily, with minimal impact on the business users, by using the exchange partition command. The exchange partition command allows the data in a non-partitioned table to be swapped into a specified partition in the partitioned table. The command does not physically move data; instead it updates the data dictionary to exchange a pointer from the partition to the table and vice versa.

In previous releases, it was not possible to generate the necessary statistics on the non-partitioned table to support incremental statistics during the partition exchange operation. Instead statistics had to be gathered on the partition after the exchange had taken place, in order to ensure the global statistics could be maintained incrementally.

In Oracle Database 18c, the necessary statistics (synopsis) can be created on the non-partitioned table prior to the exchange, so that statistics exchanged during a partition exchange load can be used to maintain incrementally global statistics automatically. The new `DBMS_STATS` table preference `INCREMENTAL_LEVEL` can be used to identify a non-partitioned table that will be used in partition exchange load. By setting the `INCREMENTAL_LEVEL` to `TABLE` (default is `PARTITION`), Oracle will automatically create a synopsis for the table when statistics are gathered. This table level synopsis will then become the partition level synopsis after the load the exchange.

For example:

```
exec dbms_stats.set_table_prefs ('SH','EXCHANGETAB','INCREMENTAL','TRUE')
exec dbms_stats.set_table_prefs ('SH','EXCHANGETAB','INCREMENTAL_LEVEL','TABLE')
```

Synopsis Enhancements

Incremental maintenance is a huge time saver for data warehouse applications with large partitioned tables. However, the performance gains of incremental can come with the price of a measurable increase in disk storage: the synopsis information is stored in the `SYSAUX` tablespace. More storage is required for synopses for tables with a high number of partitions and a large number of columns, particularly where the number of distinct values (NDV) is high. As well as consuming storage space, you might incur a performance overhead for maintaining such very large synopsis information.

From Oracle Database 12.2, `DBMS_STATS` provides new algorithm of gathering NDV information which results in much smaller synopses with a level of accuracy that is similar to the previous algorithm.

A new `DBMS_STATS` preference called `APPROXIMATE_NDV_ALGORITHM` is used to control what type of synopses is created. The default value is `REPEAT OR HYPERLOGLOG`, which means that the existing adaptive sampling algorithm will be preserved for existing synopses but new synopses will be created using the new `HYPERLOGLOG` algorithm. It is possible to use a mixture of old and new-style synopses in the same table.

When upgrading a pre-Oracle Database 12.2 system (that is using incremental statistics), there are three options:

Option 1 – Continue to use the pre-12.2 format

Change the DBMS_STATS preference to “adaptive sampling”. For example:

```
EXEC dbms_stats.set_database_prefs ('approximate_ndv_algorithm', 'adaptive sampling')
```

Option 2 – Immediately replace the old format with the new

Statistics will be re-gathered on all partitions.

```
EXEC dbms_stats.set_database_prefs ('approximate_ndv_algorithm', 'hyperloglog')
EXEC dbms_stats.set_database_prefs ('incremental_staleness', NULL)
```

Option 3 - Gradually replace the old format with the new

Old synopses are not immediately deleted and new partitions will have synopses in new format. Mixed formats will potentially yield less accurate statistics but taking this option will mean that there is no need to re-gather all statistics in the foreground because the statistics auto job will re-gather statistics on partitions with old synopses so that they will use the new format. Eventually, all synopses will be in the new format and statistics will be accurate.

```
EXEC dbms_stats.set_database_prefs ('approximate_ndv_algorithm', 'hyperloglog')
EXEC dbms_stats.set_database_prefs ('incremental_staleness', 'allow_mixed_format')
```

For new implementations in Oracle Database 18c, the recommendation is to use the default preference value - REPEAT OR HYPERLOGLOG so that new format synopses will be used.

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

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 instantaneously used by the optimizer. Beginning 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 and Importing Statistics

Statistics can be copied from one database to another. For example, in pre-production it is sometimes useful to test performance on one environment using statistics copied from another. Statistics can be copied from one database to another 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.

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 accesses more than one partition it will make use of global 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. 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`. Statistics such as average row length and number of distinct values are not adjusted and are assumed to be the same in the destination partition.

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_2002', 'SALES_Q4_2002', 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 optimizer statistics are different. For example, statistics may be different in a test environment when compared to production if the data is not in sync. To identify differences in statistics, the `DBMS_STATS.DIFF_TABLE_STATS_*` functions can be used to compare statistics for two different sources (denoted source "A" and source "B"). For example, a table in schema1 can be compared with a table in schema2. It is also possible to compare the statistics of an individual table at two different points in time or current statistics with pending statistics. For example, comparing current statistics with yesterday:

```
select report,
       maxdiffpct
from   dbms_stats.diff_table_stats_in_history(user,
                                             'CUSTOMERS',
                                             SYSDATE-1,
                                             SYSDATE,
                                             2);
```

```
REPORT
-----
#####                                MAXDIFFPCT
#####                                119.047619
STATISTICS DIFFERENCE REPORT FOR:
.....
TABLE      : CUSTOMERS
OWNER      : ADHOC
SOURCE A   : Statistics as of 14-FEB-17 05:59:06.000000 AM -08:00
SOURCE B   : Statistics as of 15-FEB-17 05:59:06.000000 AM -08:00
PCTTHRESHOLD : 2
.....
TABLE / (SUB)PARTITION STATISTICS DIFFERENCE:
.....
OBJECTNAME      TYP SRC ROWS      BLOCKS      ROWLEN      SAMPSIZE
-----
CUSTOMERS       T  A  9999          21          10          9999
                B  19998         46          10          19998
.....
COLUMN STATISTICS DIFFERENCE:
.....
COLUMN_NAME     SRC NDV      DENSITY     HIST NULLS  LEN  MIN  MAX  SAMPSIZ
-----
CUSTOMER_NUMBER A  9999      .000100010  NO  0      6  5831  58393  9999
                B 14999      .000066671  NO  0      7  5831  58393  19998
ID              A  9999      .000100010  NO  0      4  C102  C2646  9999
                B 19998      .000050005  NO  0      5  C102  C3026  19998
.....
NO DIFFERENCE IN INDEX / (SUB)PARTITION STATISTICS
#####
```

Figure 17: Comparing statistics in CUSTOMERS from one day to the next

To compare statistics in `APP1.CUSTOMERS` on one database to `APP1.CUSTOMERS` on another:

On System 1:

```
exec dbms_stats.create_stat_table('APP1','APP1STAT')
exec dbms_stats.export_table_stats('APP1','CUSTOMERS',stattab=>'APP1STAT',statid=>'mystats1')
```

Export APP1STAT table from System 1, import into System 2 and then on System 2:

```
select report
from   dbms_stats.diff_table_stats_in_stattab('APP1',
                                             'CUSTOMERS',
                                             'APP1STAT',
                                             NULL,
                                             1,
                                             'mystats1');
```

The “DIFF” functions also compare 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.

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 18: 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 19: 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 collects 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.

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 statistics will be used in addition to the existing statistics information. 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 statistics are 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.

There are circumstances where you may want to ask the optimizer to make to be more “aggressive” in its use of dynamic statistics. For example, you might want to use dynamic statistics for statements containing complex predicate expressions and extended statistics are not available or cannot be used. For example, consider a query that has non-equality in where clause predicates on two correlated columns. Standard statistics would not be sufficient and extended statistics cannot be used (because it is not an equality predicate). In the following 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. Based on the standard statistics, it 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 20: 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 statistics 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      |      |      |      |      |      |
| 1  | SORT AGGREGATE        |      |      |      |      |      |
| 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 21: Execution plan for complex predicates with dynamic sampling level 6

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 statistics 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 how much data should be sampled. 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.

```

SQL> select count(*) from product_information2
 2 where category_id in (11,13)
 3 and product_name like 'ZH%';

COUNT(*)
-----
2539520

```

Actual number of rows returned

```

SQL> explain plan for
 2 Select product_name, PRODUCT_DESCRIPTION
 3 From product_information2
 4 where category_id in (11,13)
 5 AND product_name like 'ZH%';

Explained.
SQL>
SQL> select * from table(dbms_xplan.display());

PLAN_TABLE_OUTPUT
-----
Plan hash value: 1432404272

| Id | Operation              | Name | Rows  | Bytes | Cost (%CPU)|
|* 1  | TABLE ACCESS FULL    | PRODUCT_INFORMATION2 | 40574 | 7013K | 80020 (8) |
-----
Predicate Information (identified by operation id):
-----
 1 - filter("PRODUCT_NAME" LIKE 'ZH%' AND ("CATEGORY_ID"=11 OR
"CATEGORY_ID"=13) AND "PRODUCT_NAME" IS NOT NULL)

```

Without dynamic statistics cardinality estimates is over 10X off

```

SQL> alter session set optimizer_dynamic_sampling=11;

Session altered.

SQL>
SQL> explain plan for
 2 Select product_name, PRODUCT_DESCRIPTION
 3 From product_information2
 4 where category_id in (11,13)
 5 AND product_name like 'ZH%';

Explained.
SQL>
SQL> select * from table(dbms_xplan.display());

PLAN_TABLE_OUTPUT
-----
Plan hash value: 1432404272

| Id | Operation              | Name | Rows  | Bytes | Cost (%CPU)|
| 0  | SELECT STATEMENT      |      | 254K  | 430M  | 80020 (8) |
|* 1  | TABLE ACCESS FULL    | PRODUCT_INFORMATION2 | 255K  | 430M  | 80020 (8) |
-----
Predicate Information (identified by operation id):
-----
 1 - filter("PRODUCT_NAME" LIKE 'ZH%' AND ("CATEGORY_ID"=11 OR
"CATEGORY_ID"=13) AND "PRODUCT_NAME" IS NOT NULL)

Note
-----
  dynamic statistics used: dynamic sampling (level=11)

```

With dynamic statistics cardinality estimates is accurate

Figure 22: 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 are cached in the Server Result Cache in Oracle Database 12c Release 1 and (instead)

in the SQL plan directives repository from Oracle Database 12c Release 2 onwards. This allows other SQL statements to share the statistics gathered by dynamic sampling queries. The existence of persisted dynamic sampling results can be seen in the database view, `DBA_SQL_PLAN_DIRECTIVES`, where the `TYPE` column value is `DYNAMIC_SAMPLING_RESULT` (from Oracle Database 12c Release 2 onwards).

Adaptive dynamic sampling (i.e. level-11-style dynamic sampling) can be initiated at parse time even if `OPTIMIZER_DYNAMIC_SAMPLING` is not set to 11. This can happen for parallel queries on large tables and for serial queries that have relevant `DYNAMIC_SAMPLING` SQL plan directives. This behavior is enabled if the database parameter `OPTIMIZER_DYNAMIC_STATISTICS` is set to `TRUE` (the default is `FALSE`).

System statistics

System statistics 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 defaults work well for most systems (including Oracle Exadata) so, for this reason, it is not usually necessary to gather them manually. Note that they are not automatically collected as part of the automatic statistics gathering job.

If you wish to gather system statistics manually, the default values will be overridden and this will affect the cost calculations made by the Oracle Optimizer. This is likely to change SQL execution plans, so it is important to evaluate the benefit of the change before implementing it on a production system.

To gather system statistics, `DBMS_STATS.GATHER_SYSTEM_STATS` can be used during a representative workload time window; ideally at peak workload times. Alternatively, Oracle Databases on Exadata systems have an Exadata-specific option:

```
EXEC DBMS_STATS.GATHER_SYSTEM_STATS('EXADATA')
```

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 use 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 prior to Oracle Database 12c. You can collect statistics manually on fixed objects using the `DBMS_STATS.GATHER_FIXED_OBJECTS_STATS` procedure.

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

Beginning with Oracle Database 12c, fixed table statistics are gathered by the automatic statistics gathering job if they are missing. However, it is still advisable to gather them manually once a representative workload is running.

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 import 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 recommend that you re-gather fixed object statistics if you do a major database or application upgrade.

Some fixed tables will not have statistics even if `GATHER_FIXED_OBJECTS_STATS` has been executed. This is expected behavior because some of the tables are explicitly skipped for performance reasons.

Expression Statistics

Beginning with Oracle Database 12c Release 2, optimizer expression tracking captures statistics for expressions that appear in the database workload and persists this information in the data dictionary. The data is maintained by the Oracle Optimizer and is used by the In-Memory Expressions (IME) feature of Oracle Database In-Memory.

Figure 23 shows how the statistics can be viewed. Notice how column usage information as well as expression usage is tracked.


```
BEGIN
  dbms_stats.flush_database_monitoring_info;
END;
/

select snapshot,          SNAPSHOT  TABLE_NAME  EVALUATION_COUNT  FIXED_COST  EXPRESSION_TEXT
       table_name,        CUMULATIVE  CARS         120000           .000000833  "CAR_ID"
       evaluation_count,  CUMULATIVE  CARS         33000          .000000833  "COLOR_ID"
       fixed_cost,        ...
       expression_text    LATEST     CARS         3000           .000000833  "MODEL_ID"
FROM user_expression_statistics
ORDER BY snapshot,
       table_name,
       evaluation_count desc
/
```

SNAPSHOT	TABLE_NAME	EVALUATION_COUNT	FIXED_COST	EXPRESSION_TEXT
CUMULATIVE	CARS	120000	.000000833	"CAR_ID"
CUMULATIVE	CARS	33000	.000000833	"COLOR_ID"
...				
LATEST	CARS	3000	.000000833	"MODEL_ID"
LATEST	CARS	3000	.000000833	"MAKE_ID"
LATEST	CARS	3000	.000000833	"HIGH_CARD_COL"
LATEST	COLORS	12	.000000833	"COLOR_NAME"
LATEST	COLORS	12	.000000833	"COLOR_ID"
LATEST	COLORS	12	.000083333	UPPER("COLOR_NAME")

Figure 23: Viewing expression statistics.

The statistics are updated automatically every 15 minutes, but the data can be flushed manually (as demonstrated above). The evaluation count is an estimate and, similarly, the fixed cost is an *estimate* of the cost of executing the



expression. Note that aggregate expressions (such as SUM or MAX) are not tracked and expressions that include columns from more than one table are also not tracked.

The LATEST snapshot presents the latest set of statistics captured and the CUMULATIVE presents the long-term cumulative values.



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.

Now that you have been introduced to what type of statistics are maintained by the Oracle Database, you should consider reading *Best Practices for Gathering Optimizer Statistics with Oracle Database 18c*. This white paper covers how to maintain all types of statistics effectively with minimal management overhead.



References





1. Oracle white paper: *Best Practices for Gathering Optimizer Statistics with Oracle Database 18c*
2. Oracle white paper: *Optimizer with Oracle Database 18c*



Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

CONNECT WITH US

-  blogs.oracle.com/oracle
-  facebook.com/oracle
-  twitter.com/oracle
-  oracle.com

Hardware and Software, Engineered to Work Together

Copyright © 2014, 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. 0218

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