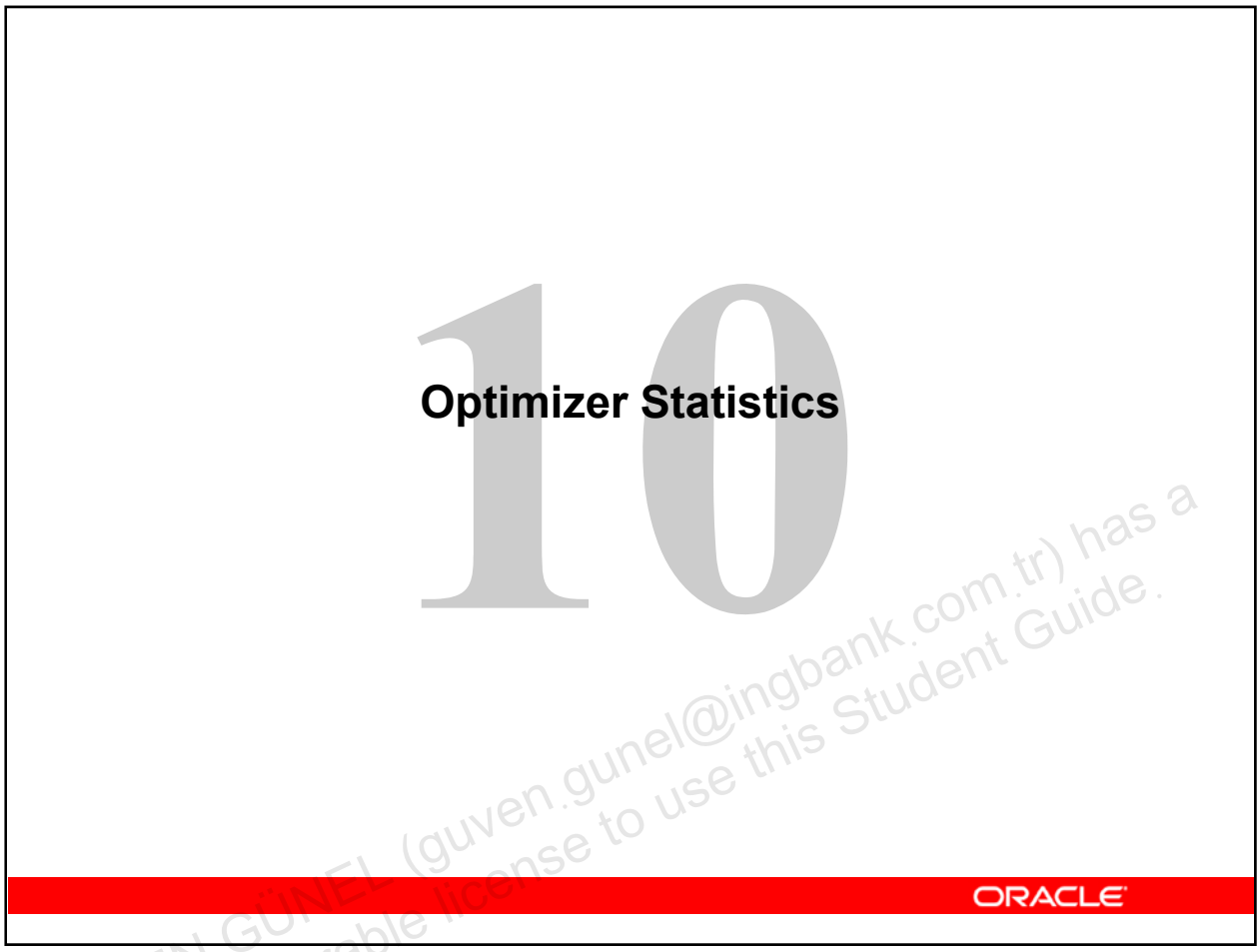
Optimizer Statistics

Chapter 10

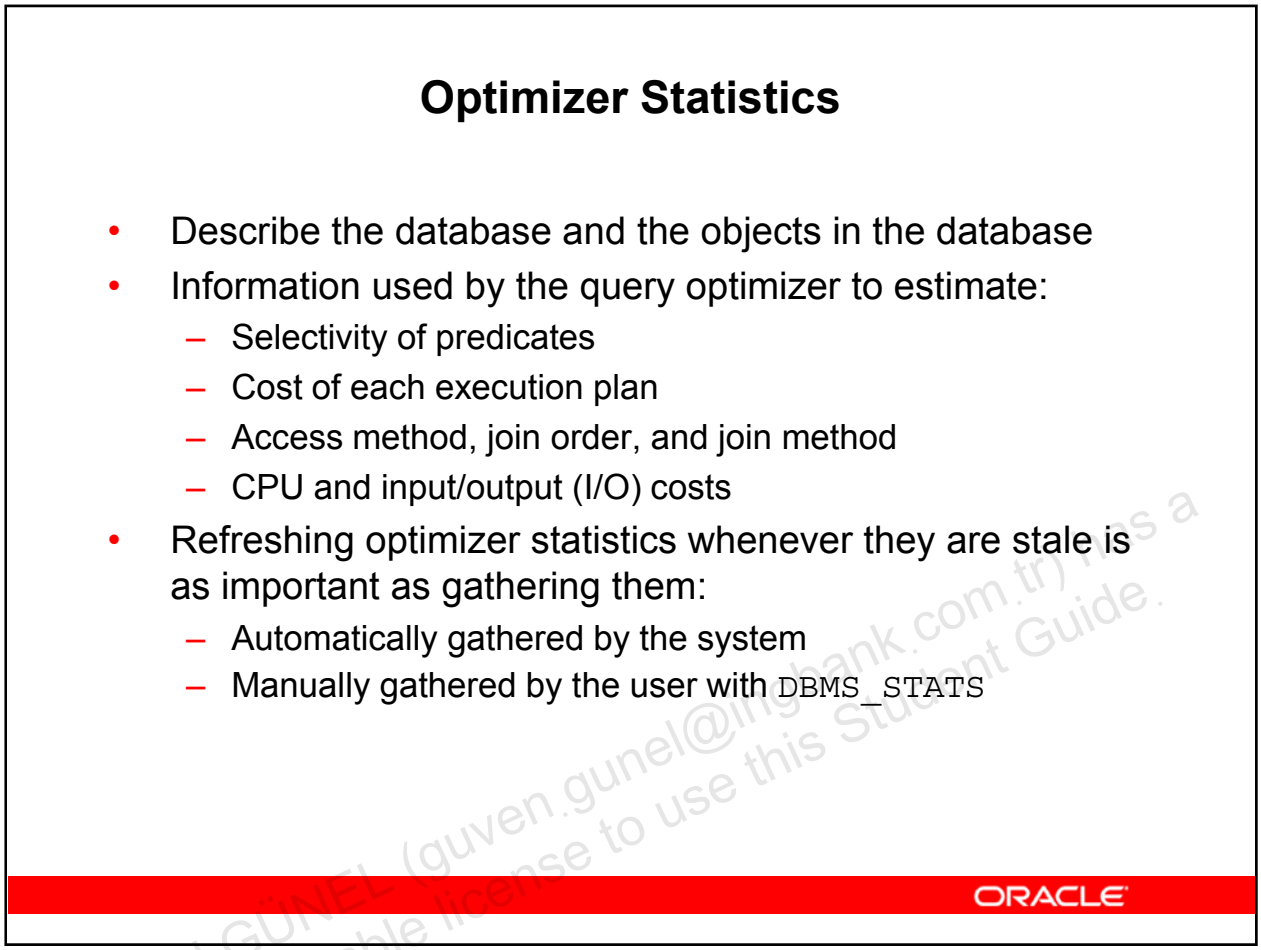


Objectives

After completing this lesson, you should be able to do the following:

- Gather optimizer statistics
- Gather system statistics
- Set statistic preferences
- Use dynamic sampling
- Manipulate optimizer statistics

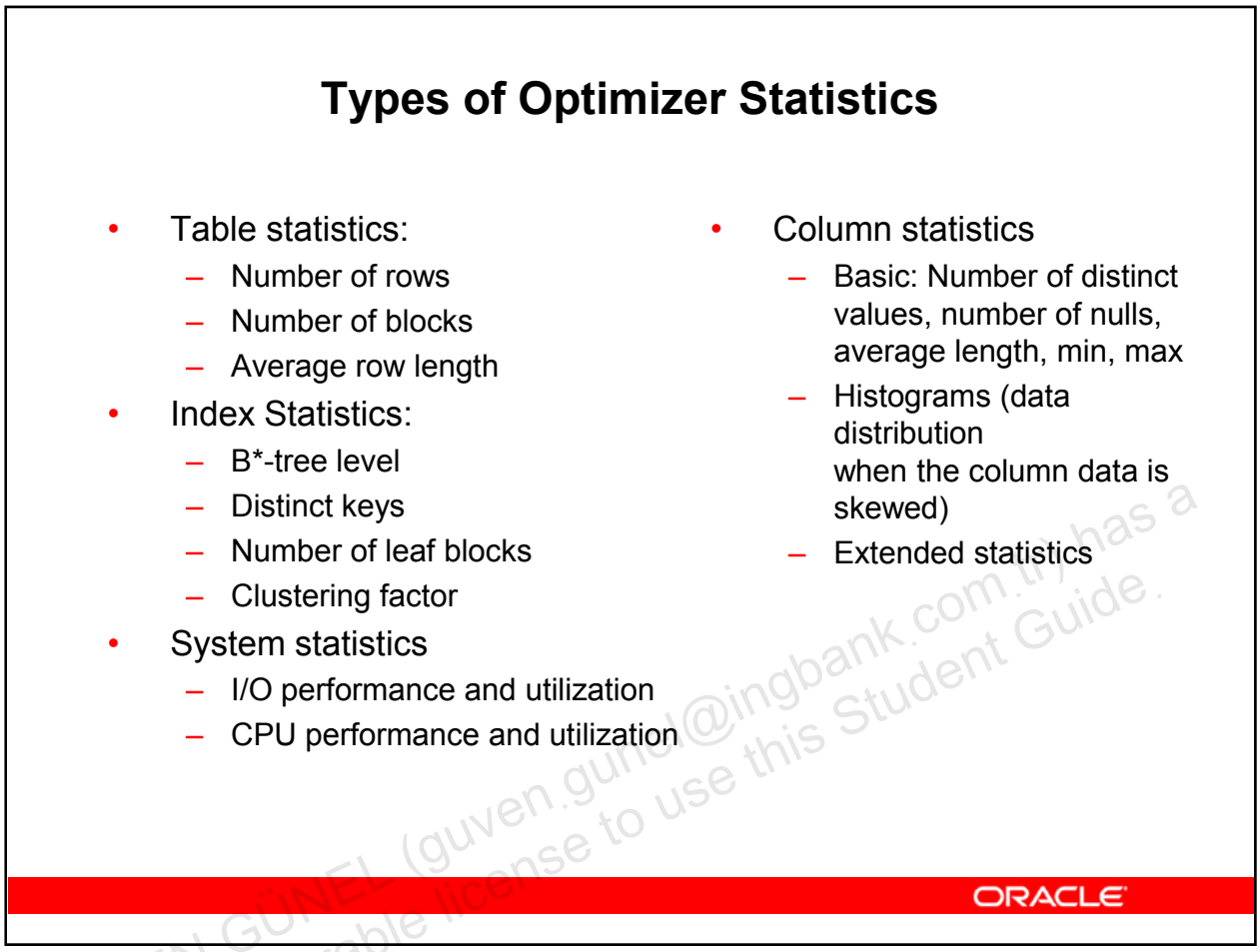
ORACLE



Optimizer Statistics

Optimizer statistics describe details about the database and the objects in the database. These statistics are used by the query optimizer to select the best execution plan for each SQL statement.

Because the objects in a database change constantly, statistics must be regularly updated so that they accurately describe these database objects. Statistics are maintained automatically by Oracle Database, or you can maintain the optimizer statistics manually using the `DBMS_STATS` package.



Types of Optimizer Statistics

Most of the optimizer statistics are listed in the slide.

Starting with Oracle Database 10*g*, index statistics are automatically gathered when the index is created or rebuilt.

Note: The statistics mentioned in this slide are optimizer statistics, which are created for query optimization and are stored in the data dictionary. These statistics should not be confused with performance statistics visible through `V$` views.

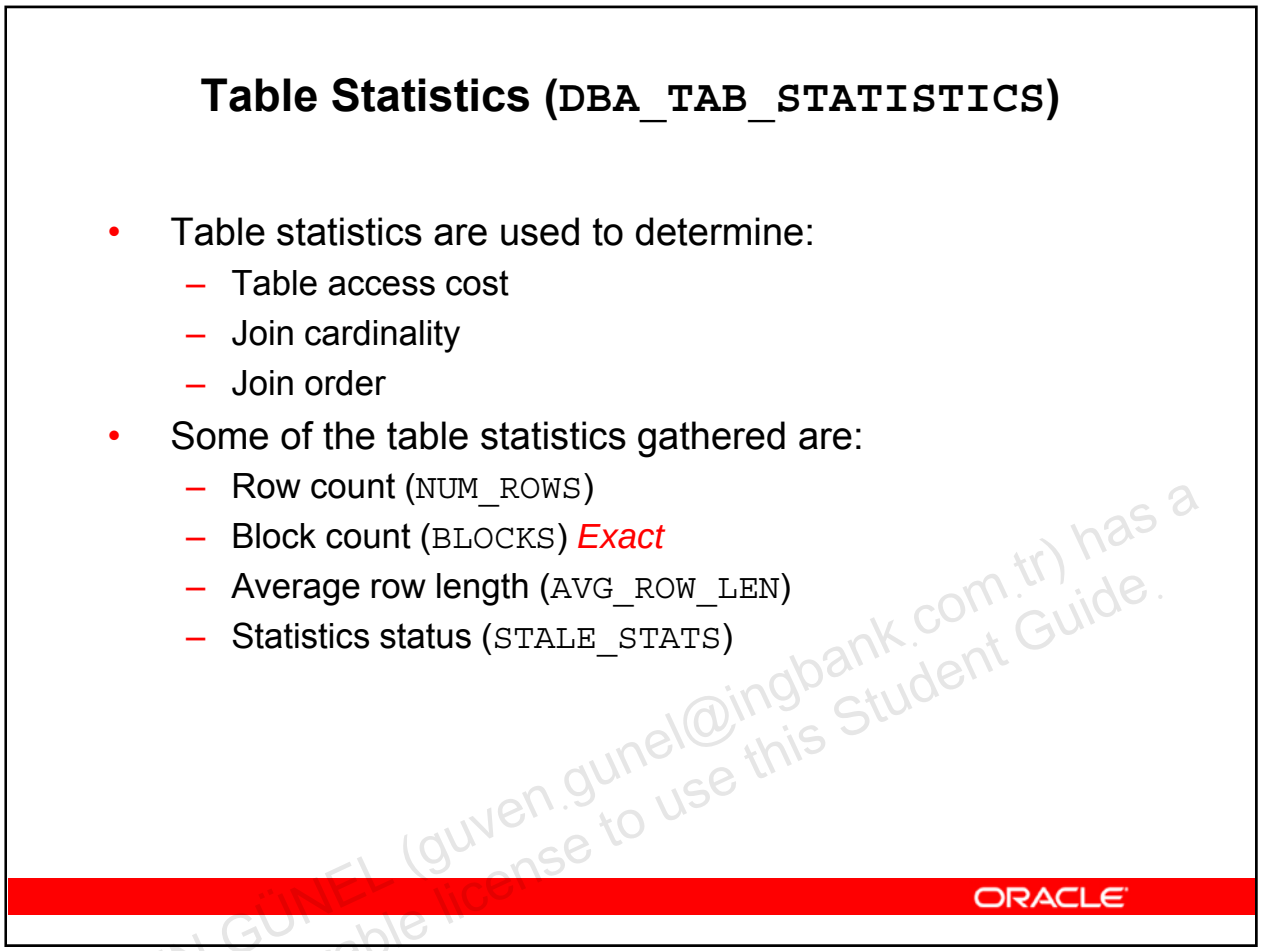


Table Statistics (DBA_TAB_STATISTICS)

NUM_ROWS
This is the basis for cardinality computations. Row count is especially important if the table is the driving table of a nested loops join, as it defines how many times the inner table is probed.

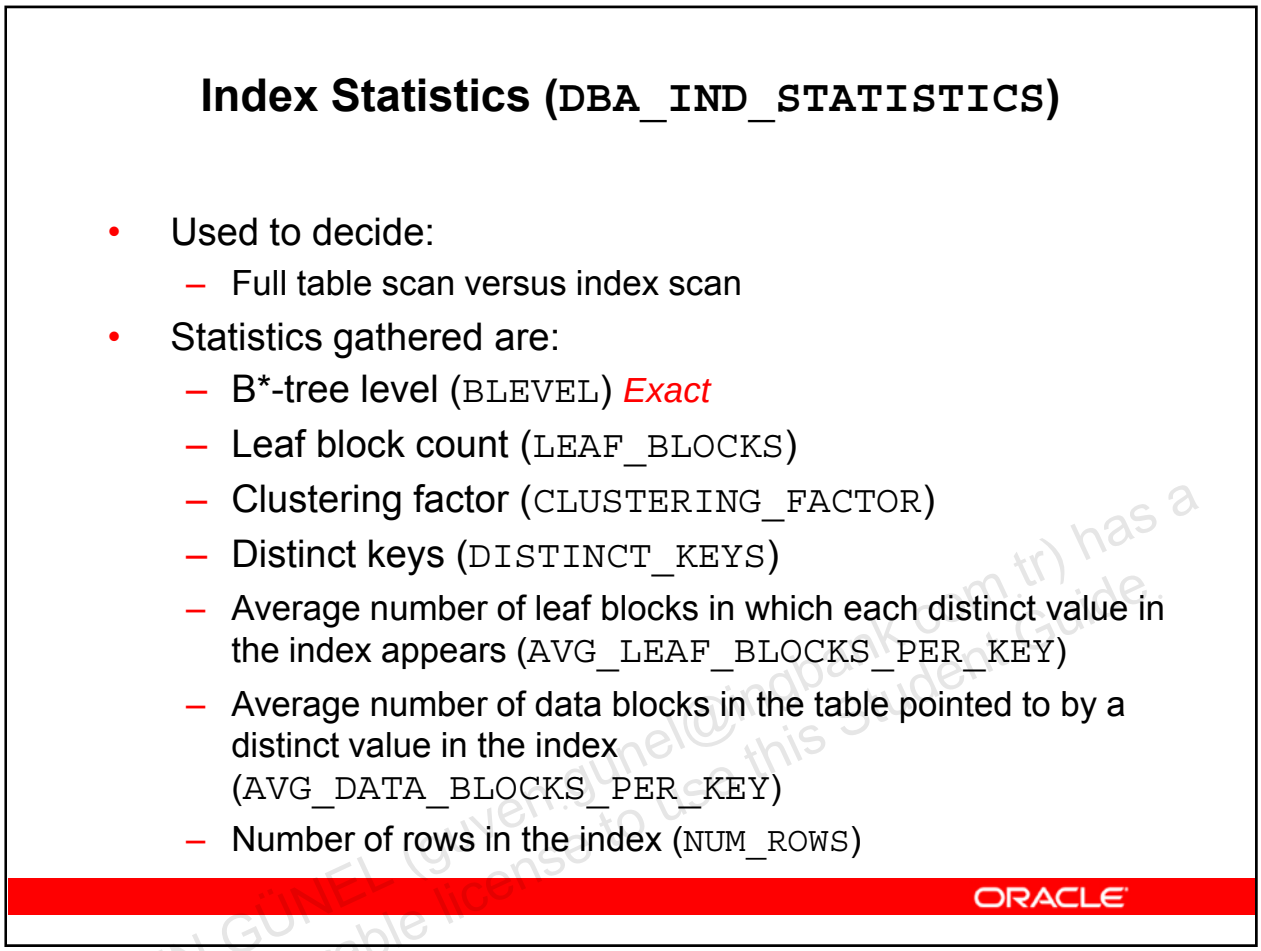
BLOCKS
This is the number of used data blocks. Block count in combination with DB_FILE_MULTIBLOCK_READ_COUNT gives the base table access cost.

AVG_ROW_LEN
This is the average length of a row in the table in bytes.

STALE_STATS

This tells you if statistics are valid on the corresponding table.

Note: There are three other statistics: EMPTY_BLOCKS, AVE_ROW_LEN, and CHAIN_CNT that are not used by the optimizer, and not gathered by the DBMS_STATS procedures. If these are required the ANALYZE command must be used.



Index Statistics (DBA_IND_STATISTICS)

In general, to select an index access, the optimizer requires a predicate on the prefix of the index columns. However, in case there is no predicate and all the columns referenced in the query are present in an index, the optimizer considers using a full index scan versus a full table scan.

BLEVEL
This is used to calculate the cost of leaf block lookups. It indicates the depth of the index from its root block to its leaf blocks. A depth of "0" indicates that the root block and leaf block are the same.

LEAF_BLOCKS
This is used to calculate the cost of a full index scan.

CLUSTERING_FACTOR
This measures the order of the rows in the table based on the values of the index. If the value is near the number of blocks, the table is very well ordered. In this case, the index entries in a single leaf block tend to point to the rows in the same data blocks. If the value is near the number of rows, the table is very randomly ordered. In this case, it is unlikely that the index entries in the same leaf block point to rows in the same data blocks.

STALE_STATS

This tells you if the statistics are valid in the corresponding index.

DISTINCT_KEYS

This is the number of distinct indexed values. For indexes that enforce `UNIQUE` and `PRIMARY KEY` constraints, this value is the same as the number of rows in the table.

AVG_LEAF_BLOCKS_PER_KEY

This is the average number of leaf blocks in which each distinct value in the index appears, rounded to the nearest integer. For indexes that enforce `UNIQUE` and `PRIMARY KEY` constraints, this value is always 1 (one).

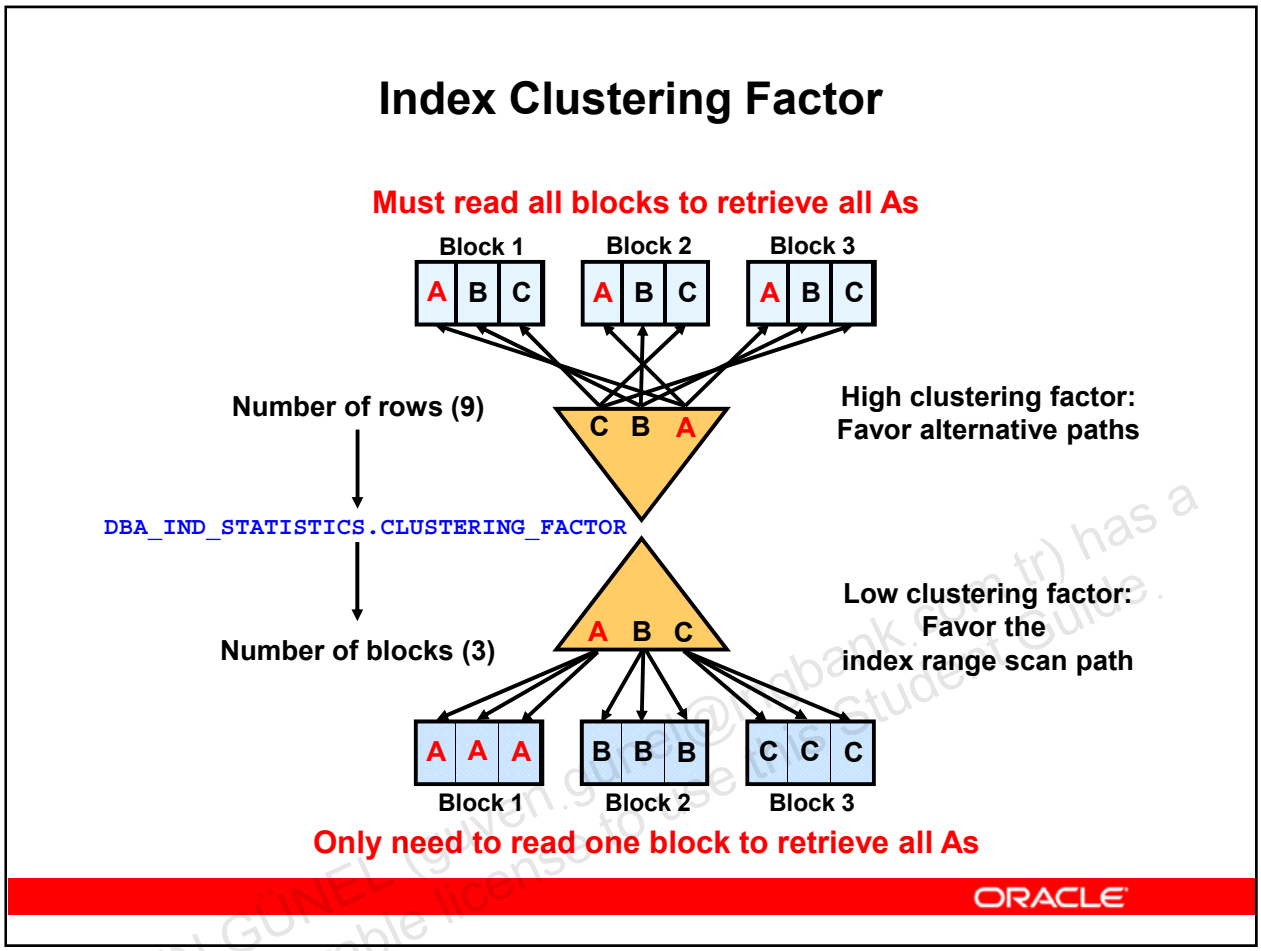
AVG_DATA_BLOCKS_PER_KEY

This is the average number of data blocks in the table that are pointed to by a distinct value in the index rounded to the nearest integer. This statistic is the average number of data blocks that contain rows that contain a given value for the indexed columns.

NUM_ROWS

It is the number of rows in the index.

Index Clustering Factor



Index Clustering Factor

The system performs input/output (I/O) by blocks. Therefore, the optimizer's decision to use full table scans is influenced by the percentage of blocks accessed, not rows. When an index range scan is used, each index entry selected points to a block in the table. If each entry points to a different block, the rows accessed and blocks accessed are the same. Consequently, the desired number of rows could be clustered together in a few blocks, or they could be spread out over a larger number of blocks. This is called the index clustering factor.

The cost formula of an index range scan uses the level of the B*-tree, the number of leaf blocks, the index selectivity, and the clustering factor. A clustering factor indicates that the individual rows are concentrated within fewer blocks in the table. A high clustering factor indicates that the individual rows are scattered more randomly across the blocks in the table. Therefore, a high clustering factor means that it costs more to use an index range scan to fetch rows by `ROWID` because more blocks in the table need to be visited to return the data. In real-life scenarios, it appears that the clustering factor plays an important role in determining the cost of an index range scan simply because the number of leaf blocks and the height of the B*-tree are relatively small compared to the clustering factor and table's selectivity.

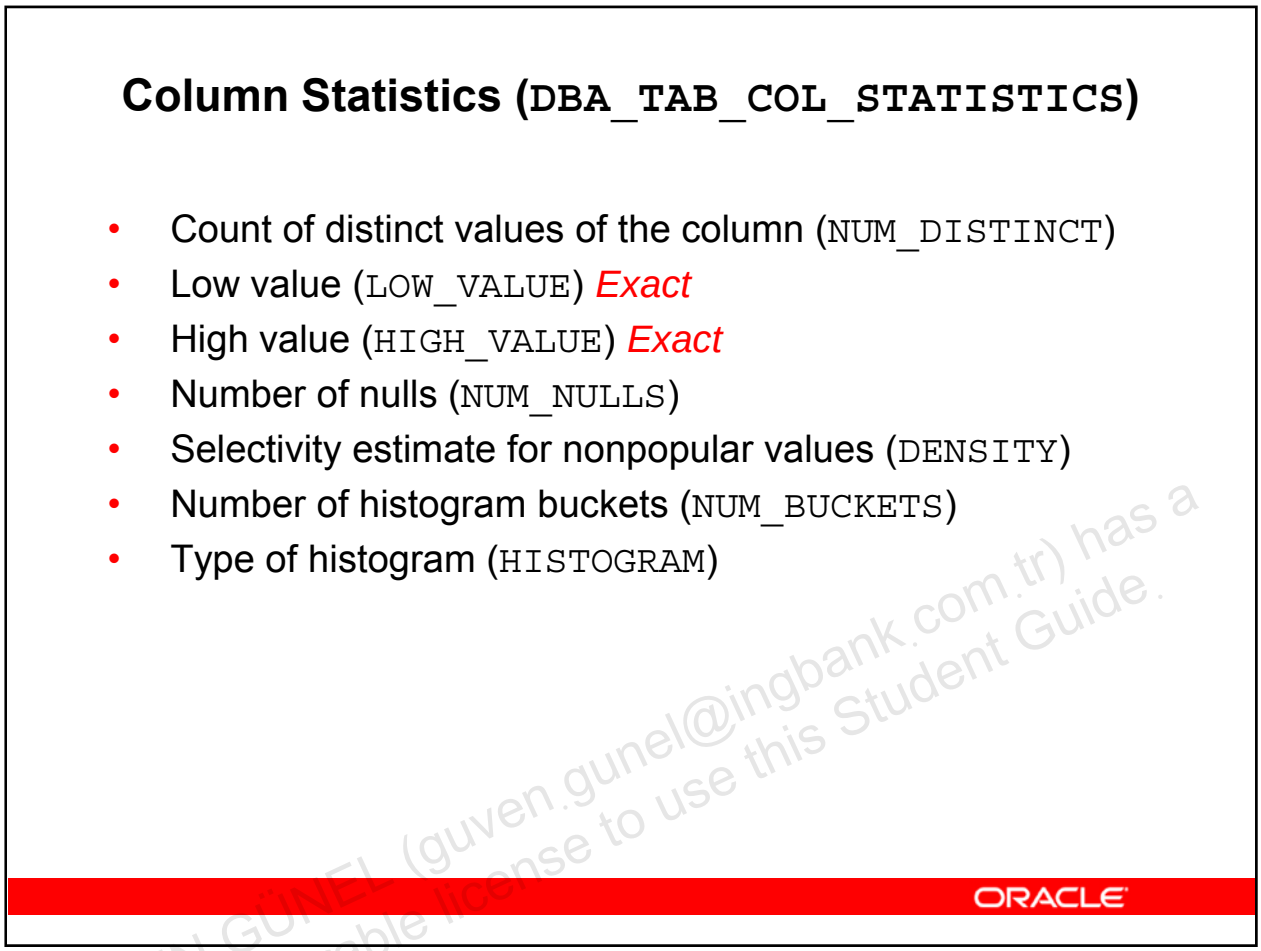
Note: If you have more than one index on a table, the clustering factor for one index might be small while at the same time the clustering factor for another index might be large. An attempt to reorganize the table to improve the clustering factor for one index can cause degradation of the clustering factor of the other index.

The clustering factor is computed and stored in the CLUSTERING_FACTOR column of the DBA_INDEXES view when you gather statistics on the index. The way it is computed is relatively easy. You read the index from left to right, and for each indexed entry, you add one to the clustering factor if the corresponding row is located in a different block than the one from the previous row. Based on this algorithm, the smallest possible value for the clustering factor is the number of blocks, and the highest possible value is the number of rows.

The example in the slide shows how the clustering factor can affect cost. Assume the following situation: There is a table with 9 rows, there is a nonunique index on col1 for table, the c1 column currently stores the values A, B, and C, and the table only has three data blocks.

- **Case 1:** If the same rows in the table are arranged so that the index values are scattered across the table blocks (rather than collocated), the index clustering factor is high.
- **Case 2:** The index clustering factor is low for the rows as they are collocated in the same block for the same value.

Note: For bitmap indexes, the clustering factor is not applicable and is not used.



Column Statistics (DBA_TAB_COL_STATISTICS)

NUM_DISTINCT is used in selectivity calculations, for example, 1/Number of Distinct Values

LOW_VALUE and **HIGH_VALUE:** The cost-based optimizer (CBO) assumes uniform distribution of values between low and high values for all data types. These values are used to determine range selectivity.

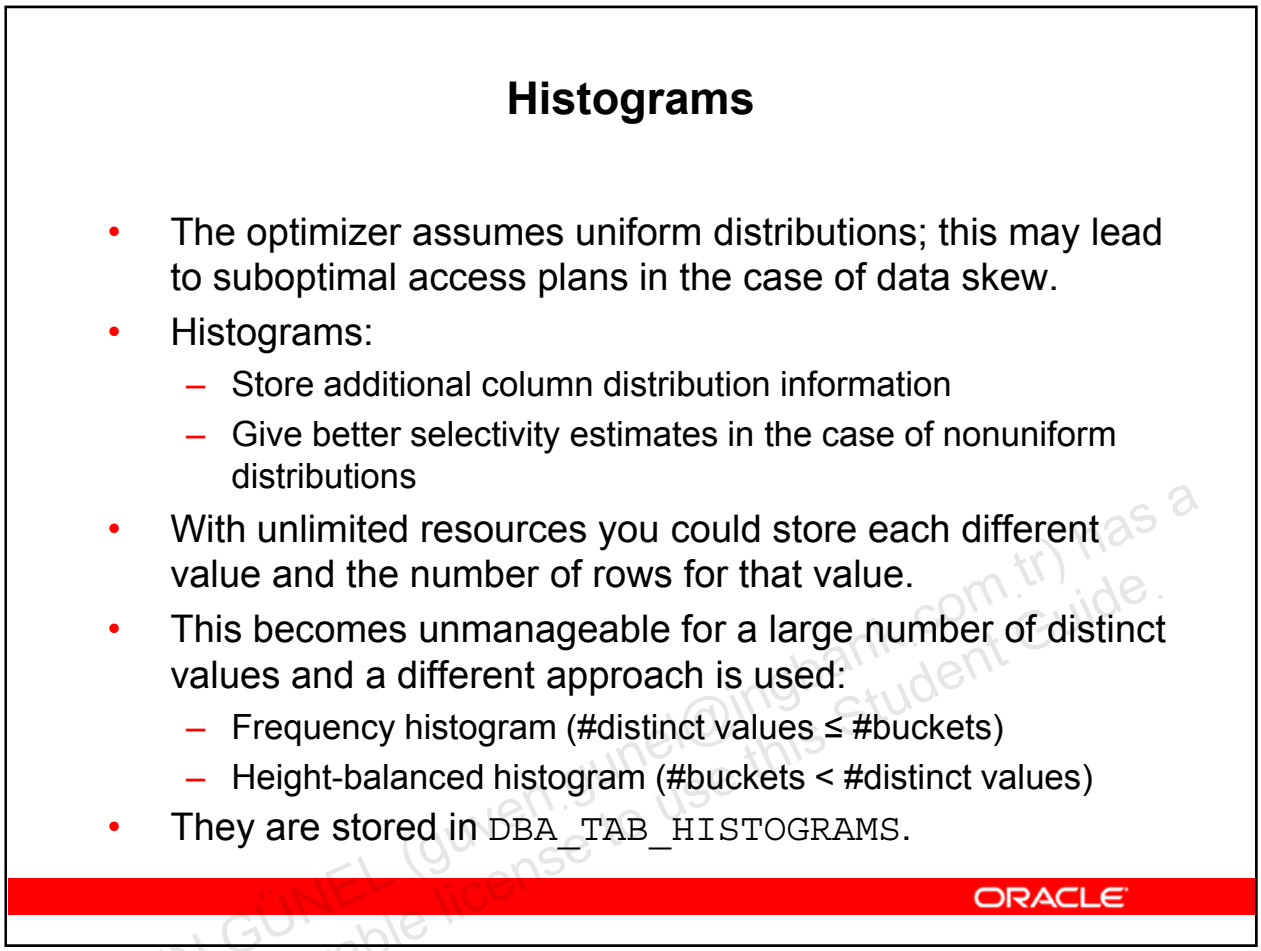
NUM_NULLS helps with selectivity of nullable columns and the `IS NULL` and `IS NOT NULL` predicates.

DENSITY is only relevant for histograms. It is used as the selectivity estimate for nonpopular values. It can be thought of as the probability of finding one particular value in this column. The calculation depends on the histogram type.

NUM_BUCKETS is the number of buckets in histogram for the column.

HISTOGRAM indicates the existence or type of the histogram: `NONE`, `FREQUENCY`, `HEIGHT BALANCED`

Histograms



Histograms

A histogram captures the distribution of different values in a column, so it yields better selectivity estimates. Having histograms on columns that contain skewed data or values with large variations in the number of duplicates help the query optimizer generate good selectivity estimates and make better decisions regarding index usage, join orders, and join methods.

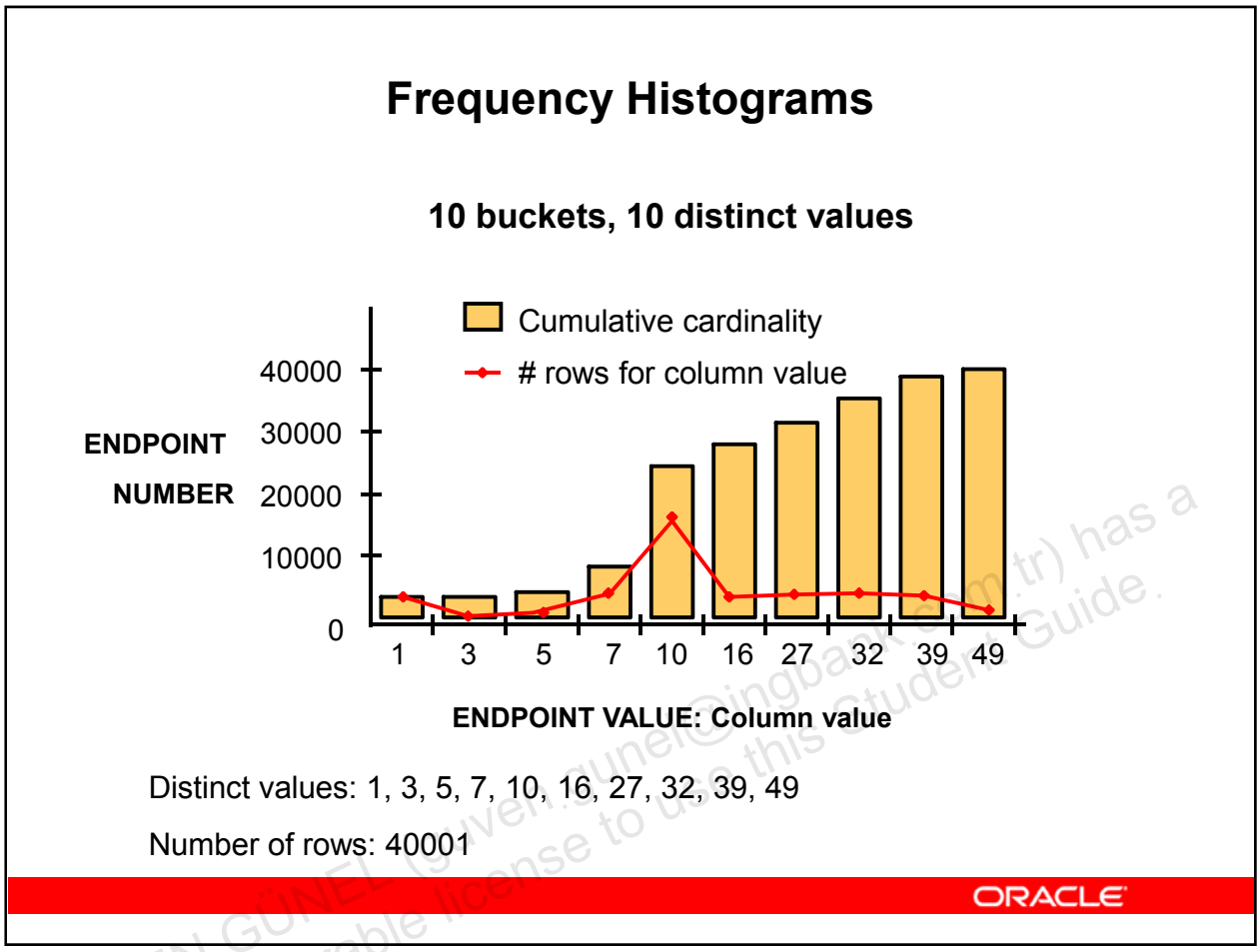
Without histograms, a uniform distribution is assumed. If a histogram is available on a column, the estimator uses it instead of the number of distinct values.

When creating histograms, Oracle Database uses two different types of histogram representations depending on the number of distinct values found in the corresponding column. When you have a data set with less than 254 distinct values, and the number of histogram buckets is not specified, the system creates a frequency histogram. If the number of distinct values is greater than the required number of histogram buckets, the system creates a height-balanced histogram.

You can find information about histograms in these dictionary views: `DBA_TAB_HISTOGRAMS`, `DBA_PART_HISTOGRAMS`, and `DBA_SUBPART_HISTOGRAMS`

Note: Gathering histogram statistics is the most resource-consuming operation in gathering statistics.

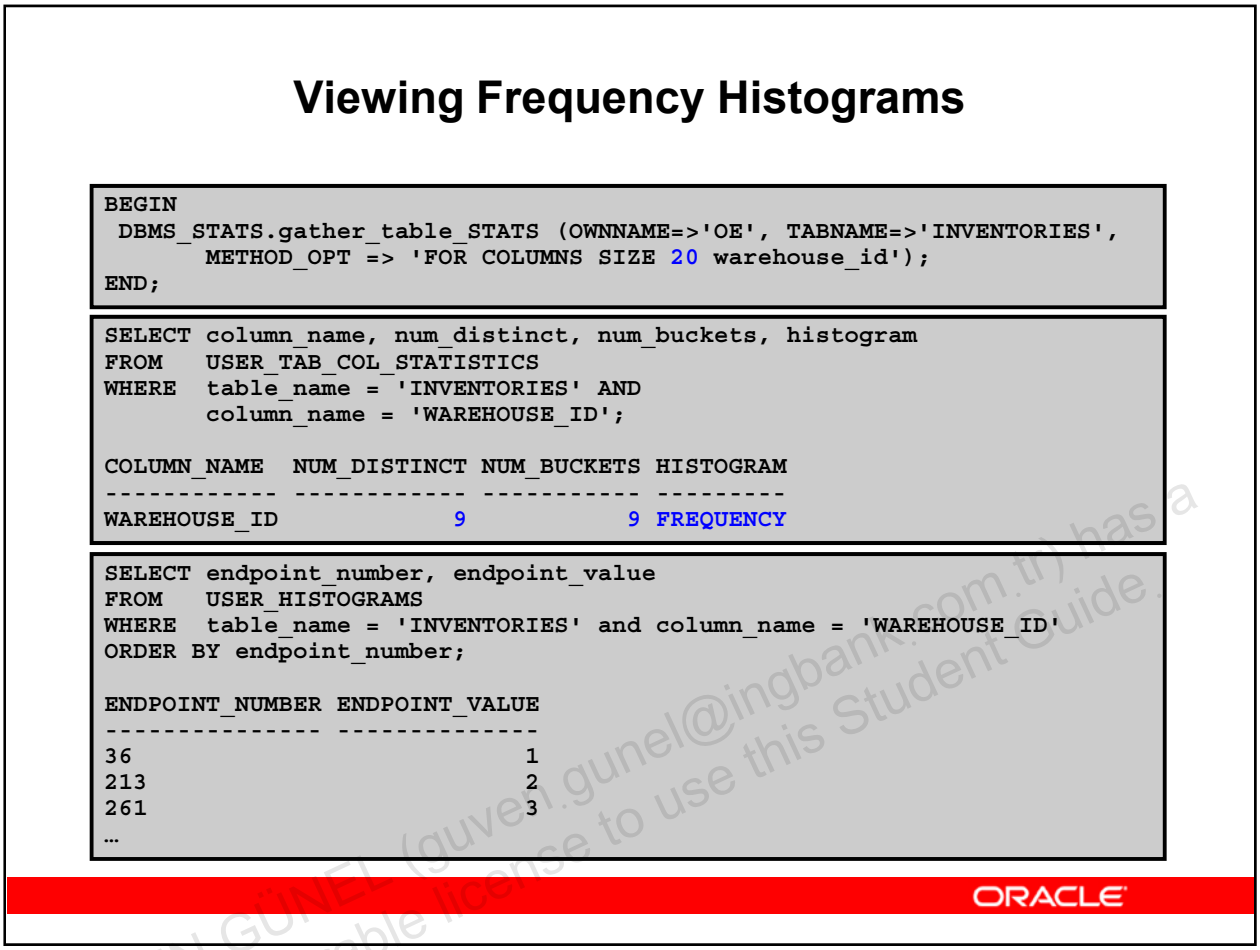
Frequency Histograms



Frequency Histograms

For the example in the slide, assume that you have a column that is populated with 40,001 numbers. You only have ten distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, and 49. Value 10 is the most popular value with 16,293 occurrences.

When the requested number of buckets equals (or is greater than) the number of distinct values, you can store each different value and record exact cardinality statistics. In this case, in `DBA_TAB_HISTOGRAMS`, the `ENDPOINT_VALUE` column stores the column value and the `ENDPOINT_NUMBER` column stores the cumulative row count including that column value, because this can avoid some calculation for range scans. The actual row counts are derived from the endpoint values if needed. The actual number of row counts is shown by the curve in the slide for clarity; only the `ENDPOINT_VALUE` and `ENDPOINT_NUMBER` columns are stored in the data dictionary.



Viewing Frequency Histograms

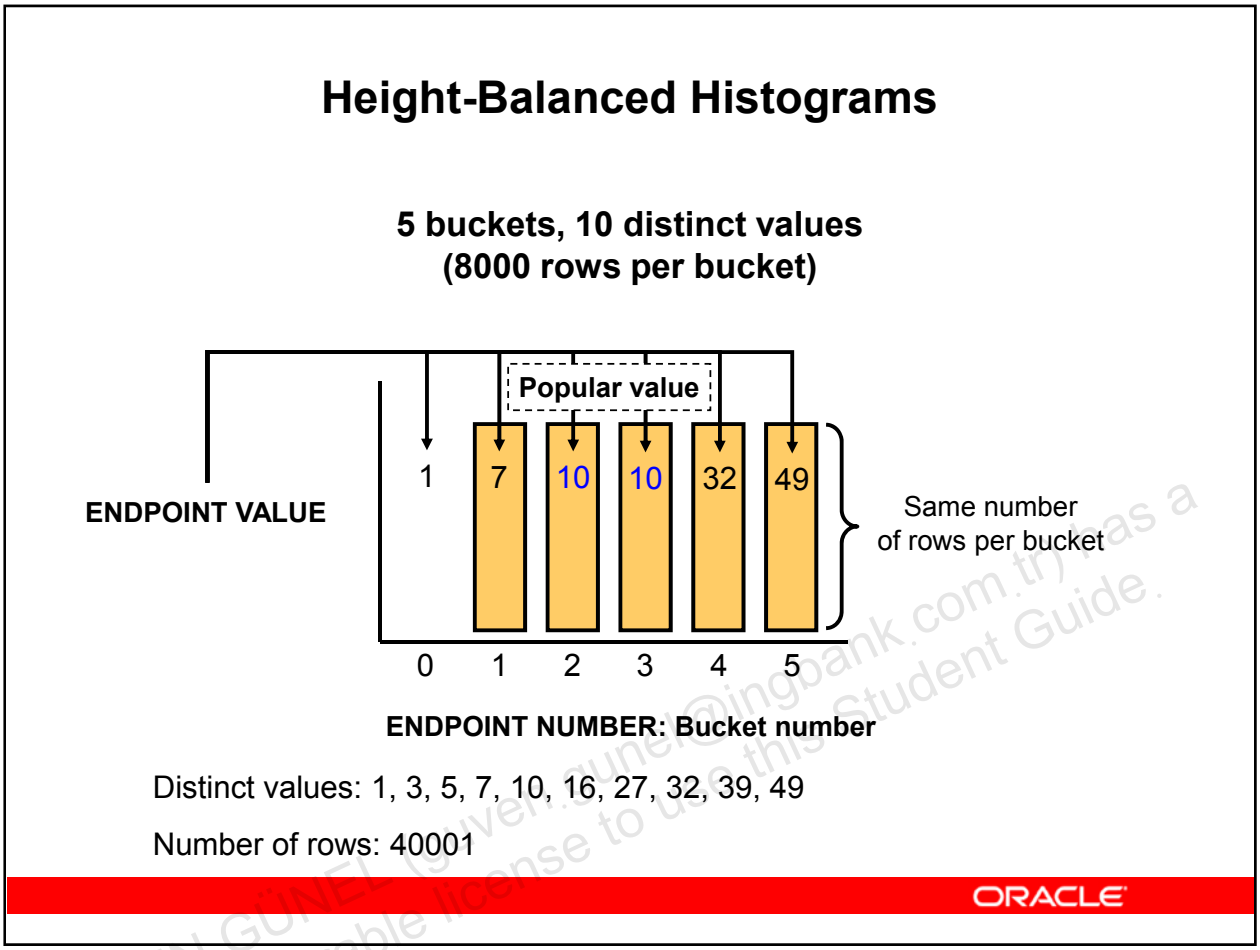
The example in the slide shows you how to view a frequency histogram. Because the number of distinct values in the `WAREHOUSE_ID` column of the `INVENTORIES` table is 9, and the number of requested buckets is 20, the system automatically creates a frequency histogram with 9 buckets. You can view this information in the `USER_TAB_COL_STATISTICS` view.

To view the histogram itself, you can query the `USER_HISTOGRAMS` view. You can see both `ENDPOINT_NUMBER` that corresponds to the cumulative frequency of the corresponding `ENDPOINT_VALUE`, which represents, in this case, the actual value of the column data.

In this case, `the warehouse_id` is 1 and there are 36 rows with `warehouse_id` = 1. There are 177 rows with `warehouse_id` = 2 so the sum of rows so far (36+177) is the cumulative frequency of 213.

Note: The `DBMS_STATS` package is dealt with later in the lesson.

Height-Balanced Histograms

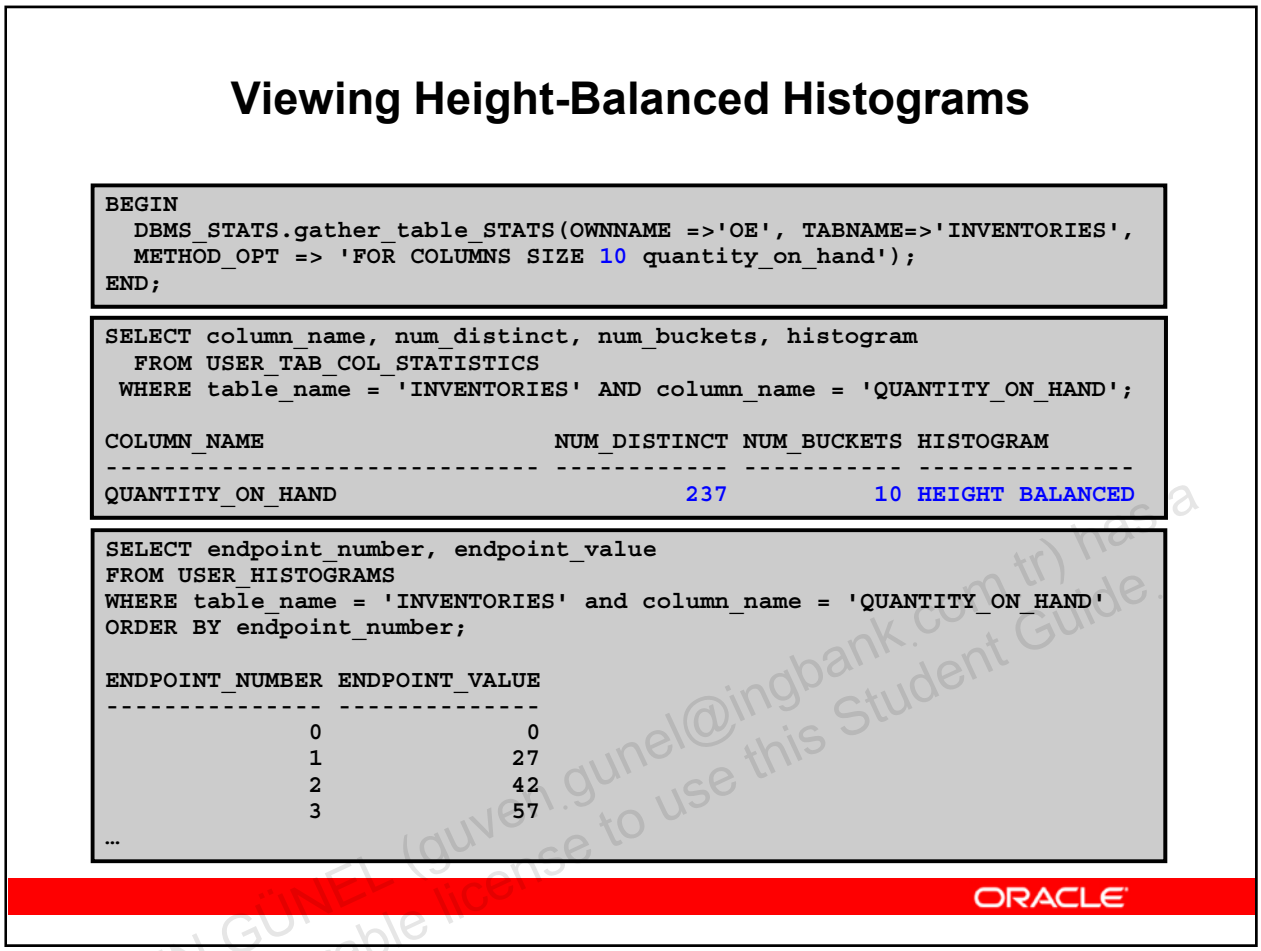


Height-Balanced Histograms

In a height-balanced histogram, the ordered column values are divided into bands so that each band contains approximately the same number of rows. The histogram tells you values of the endpoints of each band. In the example in the slide, assume that you have a column that is populated with 40,001 numbers. There will be 8,000 values in each band. You only have ten distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, and 49. Value 10 is the most popular value with 16,293 occurrences. When the number of buckets is less than the number of distinct values, `ENDPOINT_NUMBER` records the bucket number and `ENDPOINT_VALUE` records the column value that corresponds to this endpoint. In the example, the number of rows per bucket is one-fifth of the total number of rows, that is 8000. Based on this assumption, value 10 appears between 8000 and 24000 times. So you are sure that value 10 is a popular value.

This type of histogram is good for equality predicates on popular value, and range predicates.

The number of rows per bucket is not recorded because this can be derived from the total number of values and the fact that all the buckets contain an equal number of values. In this example, value 10 is a popular value because it spans multiple endpoint values. To save space, the histogram does not actually store duplicated buckets. In the example in the slide, bucket 2 (with endpoint value 10) would not be recorded in `DBA_TAB_HISTOGRAMS` for that reason.

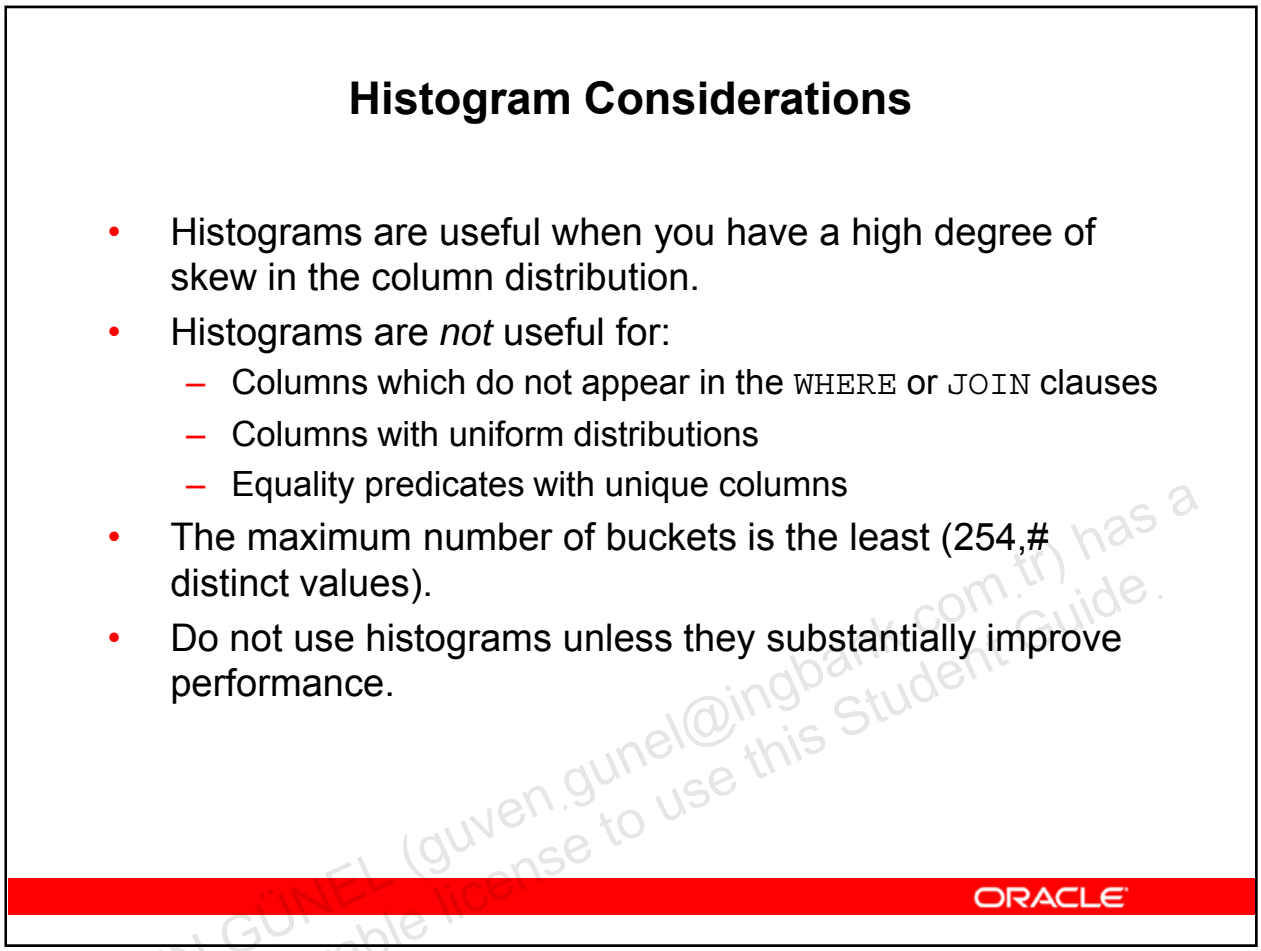


Viewing Height-Balanced Histograms

The example in the slide shows you how to view a height-balanced histogram. Because the number of distinct values in the QUANTITY_ON_HAND column of the INVENTORIES table is 237, and the number of requested buckets is 10, the system automatically creates a height-balanced histogram with 10 buckets. You can view this information in the USER_TAB_COL_STATISTICS view.

To view the histogram itself, you can query the USER_HISTOGRAMS view. You can see that the ENDPOINT_NUMBER corresponds to the bucket number, and ENDPOINT_VALUE corresponds to values of the endpoints end.

Note: The DBMS_STATS package is dealt with later in the lesson.



Histogram Considerations

Histograms are useful only when they reflect the current data distribution of a given column. The data in the column can change as long as the distribution remains constant. If the data distribution of a column changes frequently, you must recompute its histogram frequently.

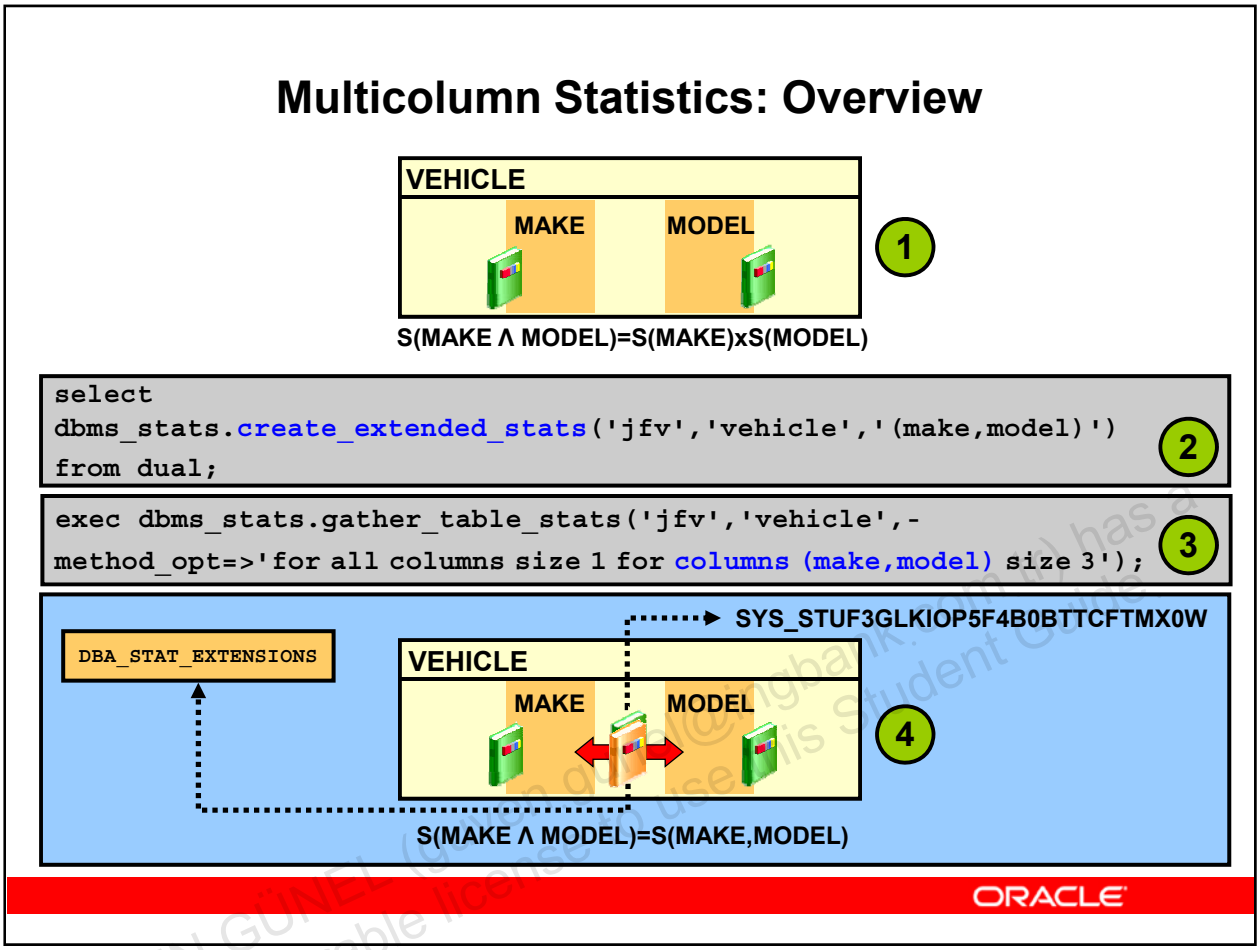
Histograms are useful when you have a high degree of data skew in the columns for which you want to create histograms.

However, there is no need to create histograms for columns which do not appear in a `WHERE` clause of a SQL statement. Similarly, there is no need to create histograms for columns with uniform distribution.

In addition, for columns declared as `UNIQUE`, histograms are useless because the selectivity is obvious. Also, the maximum number of buckets is 254, which can be lower depending on the actual number of distinct column values. Histograms can affect performance and should be used only when they substantially improve query plans. For uniformly distributed data, the optimizer can make fairly accurate guesses about the cost of executing a particular statement without the use of histograms.

Note: Character columns have some exceptional behavior as histogram data is stored only for the first 32 bytes of any string.

Multicolumn Statistics: Overview



Multicolumn Statistics: Overview

With Oracle Database 10*g*, the query optimizer takes into account the correlation between columns when computing the selectivity of multiple predicates in the following limited cases:

- If all the columns of a conjunctive predicate match all the columns of a concatenated index key, and the predicates are equalities used in equijoins, then the optimizer uses the number of distinct keys (NDK) in the index for estimating selectivity, as 1/NDK.
- When `DYNAMIC_SAMPLING` is set to level `4`, the query optimizer uses dynamic sampling to estimate the selectivity of complex predicates involving several columns from the same table. However, the sample size is very small and increases parsing time. As a result, the sample is likely to be statistically inaccurate and may cause more harm than good.

In all other cases, the optimizer assumes that the values of columns used in a complex predicate are independent of each other. It estimates the selectivity of a conjunctive predicate by multiplying the selectivity of individual predicates. This approach results in underestimation of the selectivity if there is a correlation between the columns. To circumvent this issue, Oracle Database 11*g* allows you to collect, store, and use the following statistics to capture functional dependency between two or more columns (also called groups of columns): number of distinct values, number of nulls, frequency histograms, and density.

For example, consider a VEHICLE table in which you store information about cars. The MAKE and MODEL columns are highly correlated, in that MODEL determines MAKE. This is a strong dependency, and both columns should be considered by the optimizer as highly correlated. You can signal that correlation to the optimizer by using the CREATE_EXTENDED_STATS function as shown in the example in the slide, and then compute the statistics for all columns (including the ones for the correlated groups that you created).

The optimizer only uses multicolumn statistics with equality predicates.

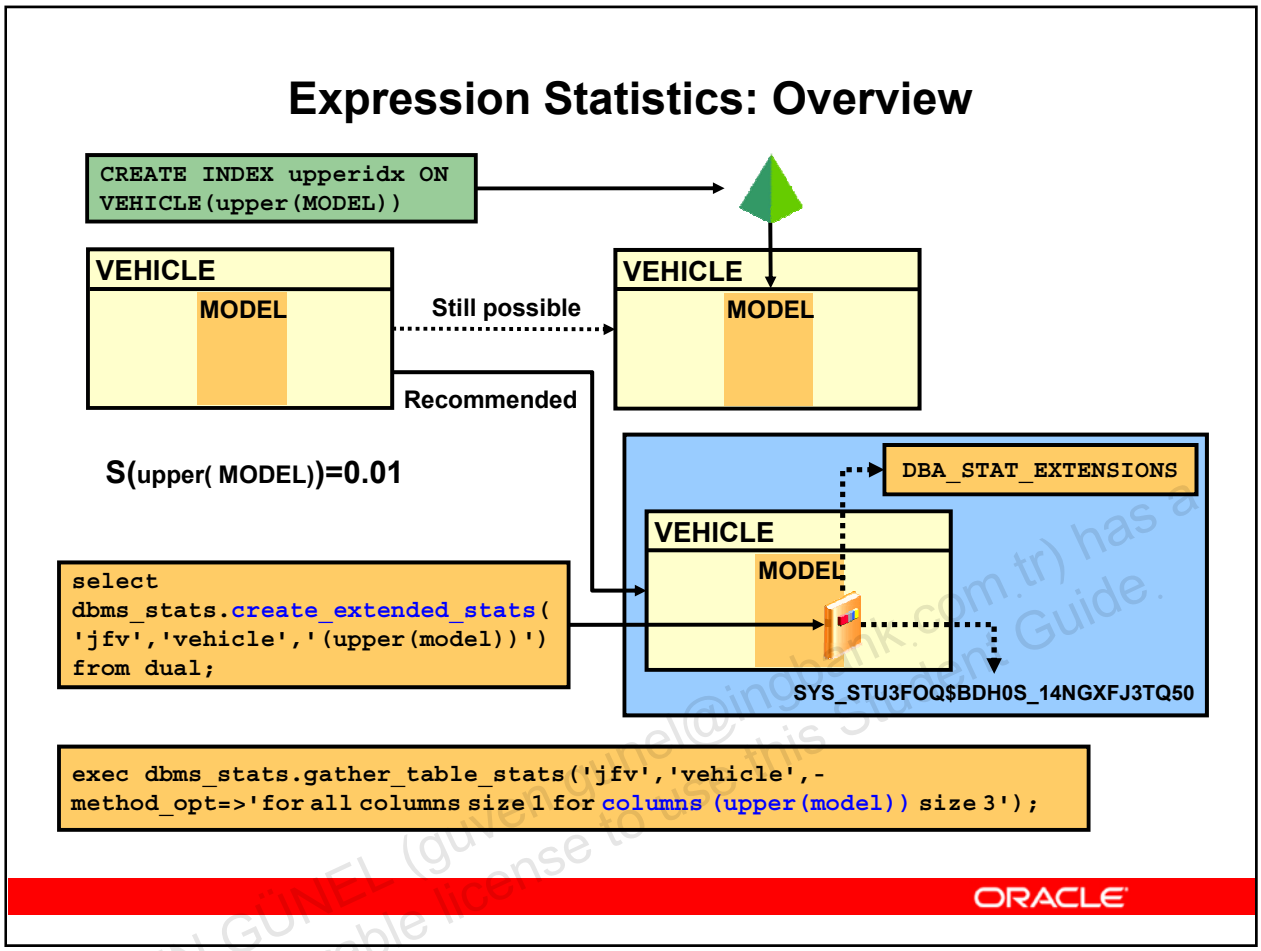
Note

- The CREATE_EXTENDED_STATS function returns a virtual hidden column name, such as SYS_STUW_5RHLX443AN1ZCLPE_GLE4.
- Based on the example in the slide, the name can be determined by using the following SQL:

```
select
dbms_stats.show_extended_stats_name('jfv','vehicle','(make,model)
') from dual
```

- After you create the statistics extensions, you can retrieve them by using the ALL|DBA|USER_STAT_EXTENSIONS views.

Expression Statistics: Overview



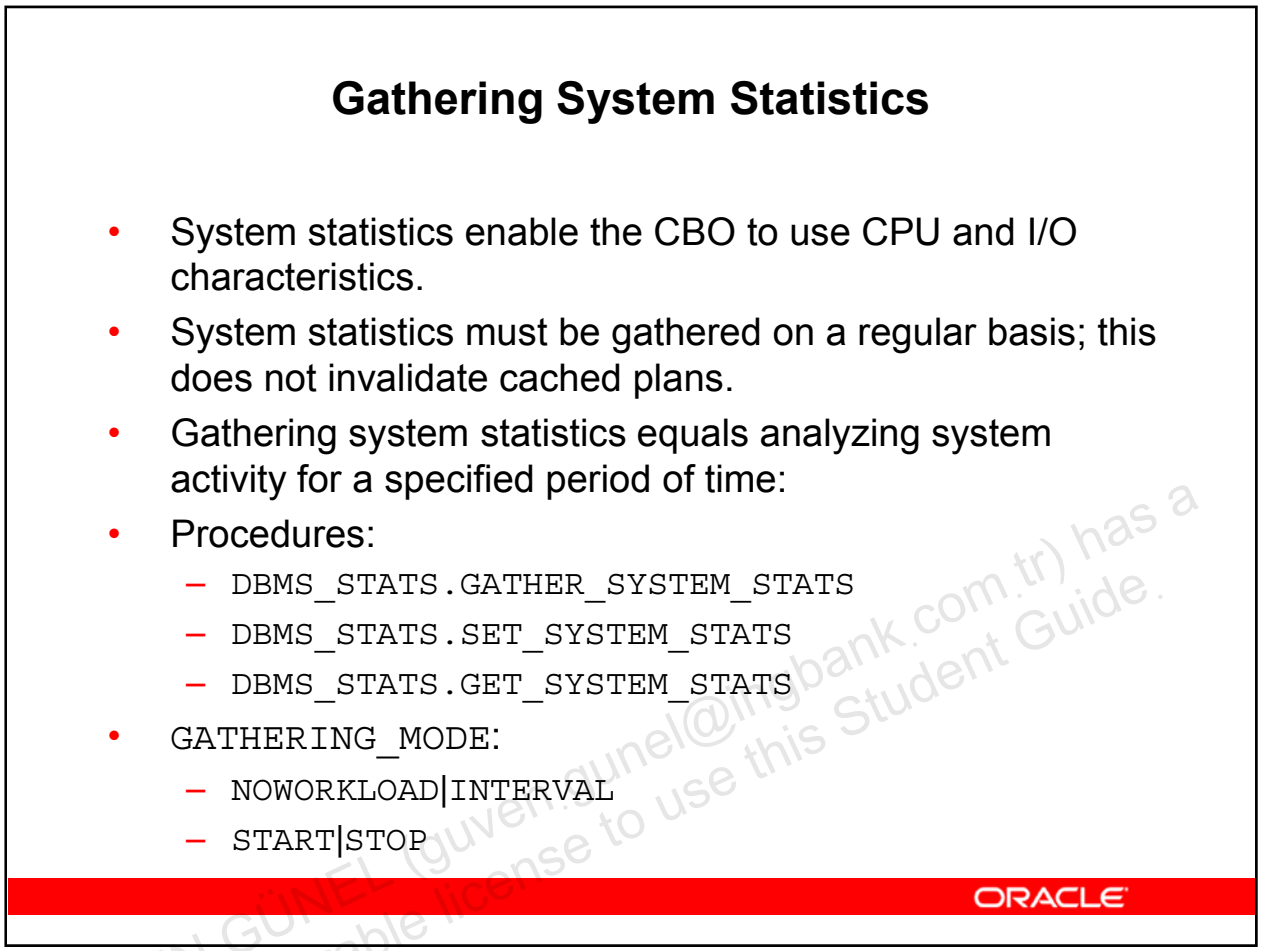
Expression Statistics: Overview

Predicates involving expressions on columns are a significant issue for the query optimizer. When computing selectivity on predicates of the form *function(Column) = constant*, the optimizer assumes a static selectivity value of 1 percent. This approach almost never has the correct selectivity and it may cause the optimizer to produce suboptimal plans.

The query optimizer has been extended to better handle such predicates in limited cases where functions preserve the data distribution characteristics of the column and thus allow the optimizer to use the columns statistics. An example of such a function is `TO_NUMBER`.

Further enhancements have been made to evaluate built-in functions during query optimization to derive better selectivity using dynamic sampling. Finally, the optimizer collects statistics on virtual columns created to support function-based indexes.

However, these solutions are either limited to a certain class of functions or work only for expressions used to create function-based indexes. By using expression statistics in Oracle Database 11*g*, you can use a more general solution that includes arbitrary user-defined functions and does not depend on the presence of function-based indexes. As shown in the example in the slide, this feature relies on the virtual column infrastructure to create statistics on expressions of columns.



Gathering System Statistics

System statistics allow the optimizer to consider a system's I/O and CPU performance, and utilization. For each candidate plan, the optimizer computes estimates for I/O and CPU costs. It is important to know the system characteristics to select the most efficient plan with optimal proportion between I/O and CPU cost. System CPU and I/O characteristics depend on many factors and do not stay constant all the time. Using system statistics management routines, you can capture statistics in the interval of time when the system has the most common workload. For example, database applications can process online transaction processing (OLTP) transactions during the day and run OLAP reports at night. You can gather statistics for both states and activate appropriate OLTP or OLAP statistics when needed. This allows the optimizer to generate relevant costs with respect to the available system resource plans. When the system generates system statistics, it analyzes system activity in a specified period of time. Unlike the table, index, or column statistics, the system does not invalidate already parsed SQL statements when system statistics get updated. All new SQL statements are parsed using new statistics.

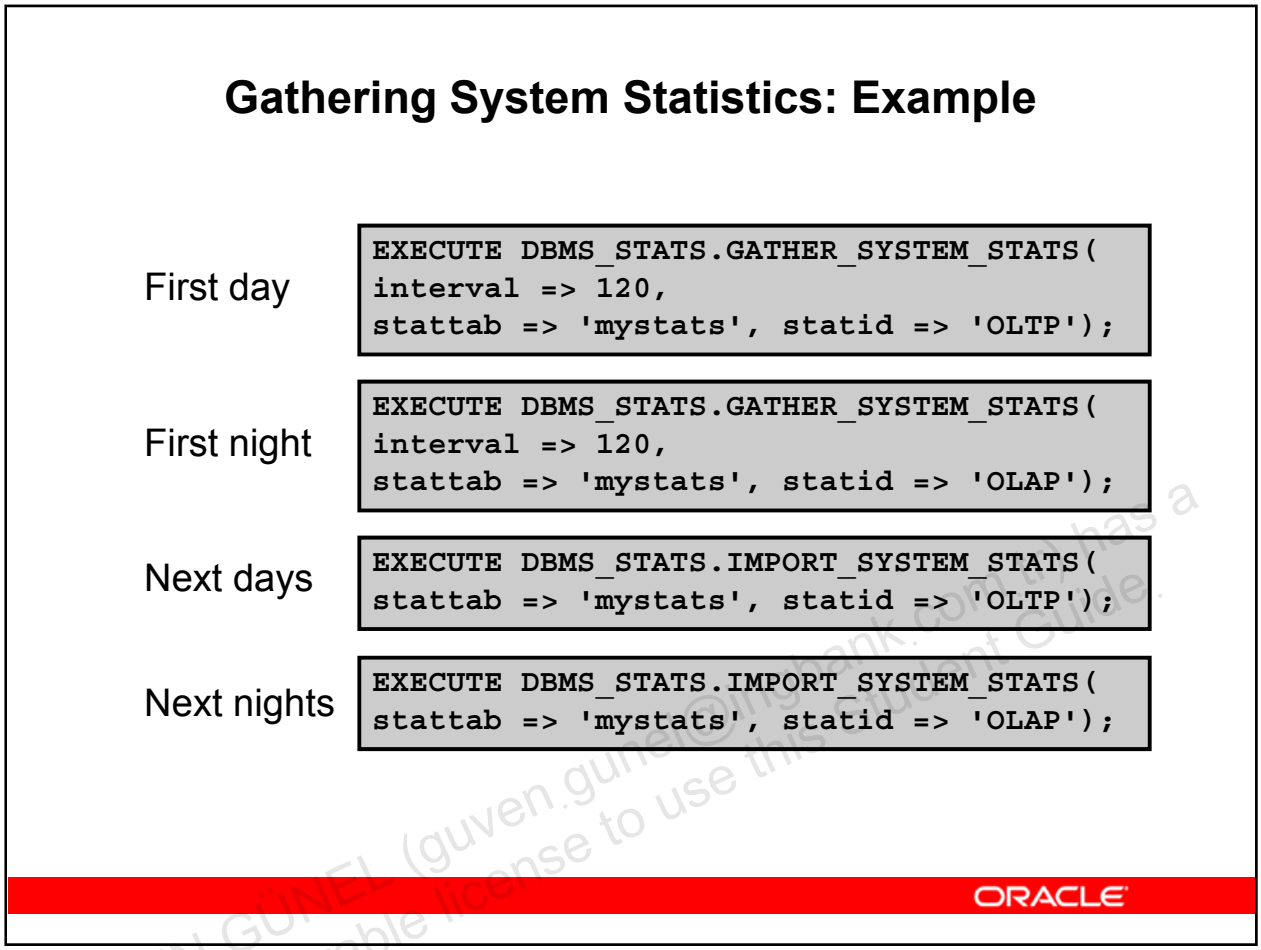
It is highly recommended that you gather system statistics. System statistics are gathered in a user-defined time frame with the `DBMS_STATS.GATHER_SYSTEM_STATS` routine. You can also set system statistics values explicitly using `DBMS_STATS.SET_SYSTEM_STATS`. Use `DBMS_STATS.GET_SYSTEM_STATS` to verify system statistics.

When you use the GATHER_SYSTEM_STATS procedure, you should specify the GATHERING_MODE parameter:

- NOWORKLOAD: This is the default. This mode captures characteristics of the I/O system. Gathering may take a few minutes and depends on the size of the database. During this period the system estimates the average read seek time and transfer speed for the I/O system. This mode is suitable for all workloads. It is recommended that you run GATHER_SYSTEM_STATS ('noworkload') after you create the database and tablespaces.
- INTERVAL: Captures system activity during a specified interval. This works in combination with the interval parameter that specifies the amount of time for the capture. You should provide an interval value in minutes, after which system statistics are created or updated in the dictionary or a staging table. You can use GATHER_SYSTEM_STATS (gathering_mode=>'STOP') to stop gathering earlier than scheduled.
- START | STOP: Captures system activity during specified start and stop times, and refreshes the dictionary or a staging table with statistics for the elapsed period.

Note: Since Oracle Database 10*g,* Release 2, the system automatically gathers essential parts of system statistics at startup.

Gathering System Statistics: Example



Gathering System Statistics: Example

The example in the slide shows database applications processing OLTP transactions during the day and running reports at night.

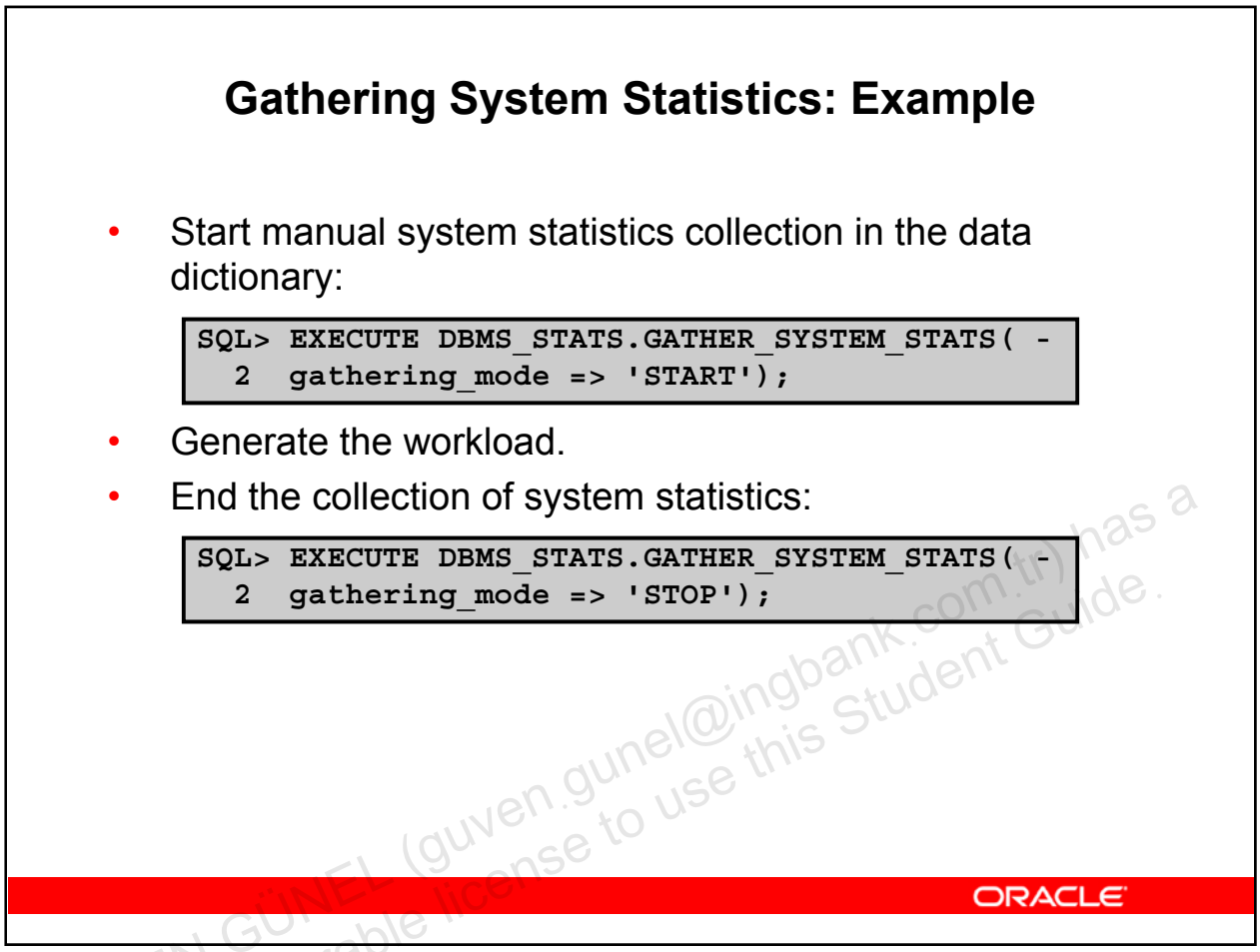
First, system statistics must be collected during the day. In this example, gathering ends after 120 minutes and is stored in the `mystats` table.

Then, system statistics are collected during the night. Gathering ends after 120 minutes and is stored in the `mystats` table.

Generally, the syntax in the slide is used to gather system statistics. Before invoking the `GATHER_SYSTEM_STATS` procedure with the `INTERVAL` parameter specified, you must activate job processes using a command, such as `SQL> alter system set job_queue_processes = 1;`.

Note: In Oracle Database 11*g* Release 2, the default value of `job_queue_processes` is 1000. You can also invoke the same procedure with different arguments to enable manual gathering instead of using jobs.

If appropriate, you can switch between the statistics gathered. Note that it is possible to automate this process by submitting a job to update the dictionary with appropriate statistics. During the day, a job may import the OLTP statistics for the daytime run, and during the night, another job imports the online analytical processing (OLAP) statistics for the nighttime run.

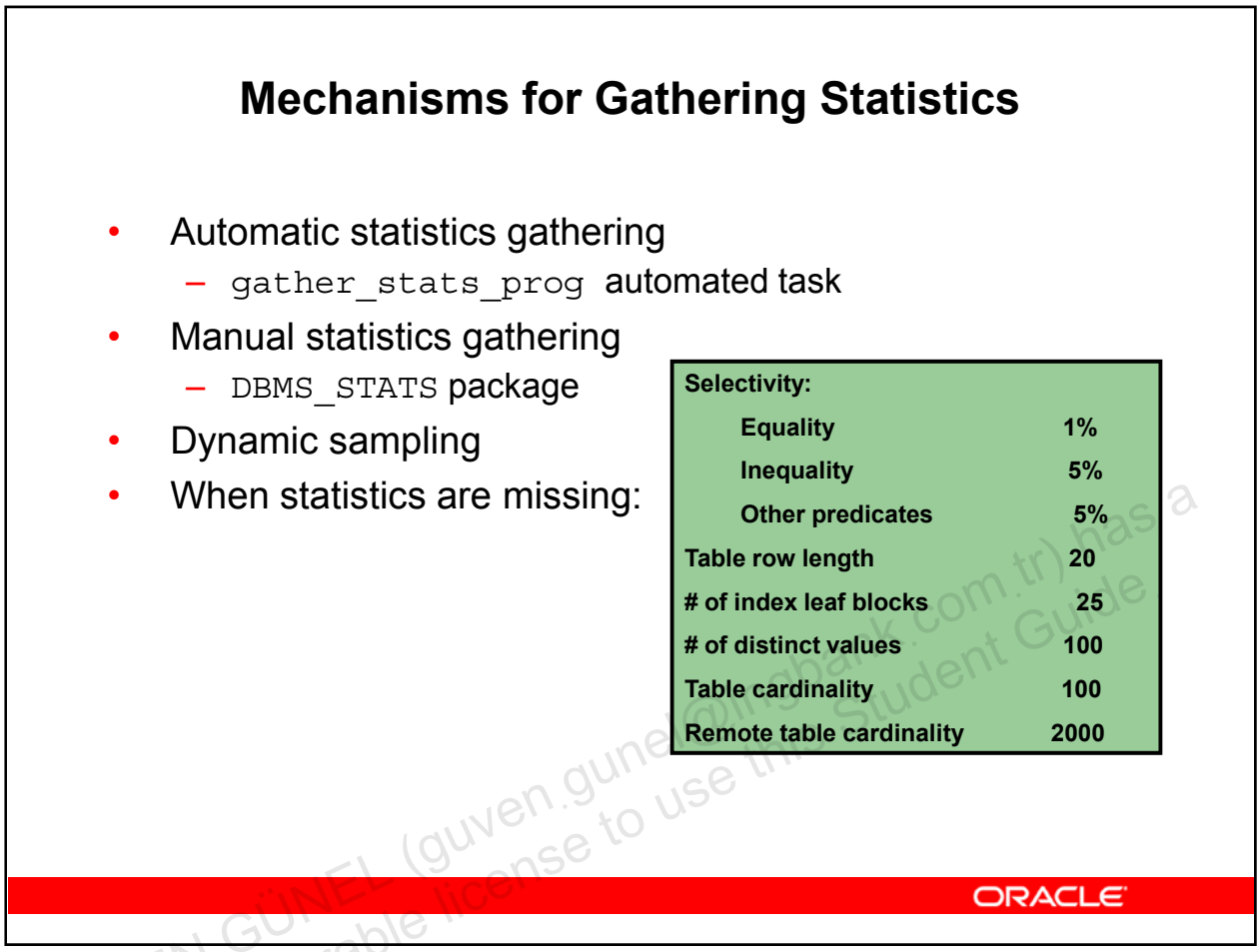


Gathering System Statistics: Example (continued)

The example in the previous slide shows how to collect system statistics with jobs by using the internal parameter of the `DBMS_STATS.GATHER_SYSTEM_STATS` procedure. To collect system statistics manually, another parameter of this procedure can be used as shown in the slide.

First, you must start the system statistics collection, and then you can end the collection process at any time after you are certain that a representative workload has been generated on the instance.

The example collects system statistics and stores them directly in the data dictionary.

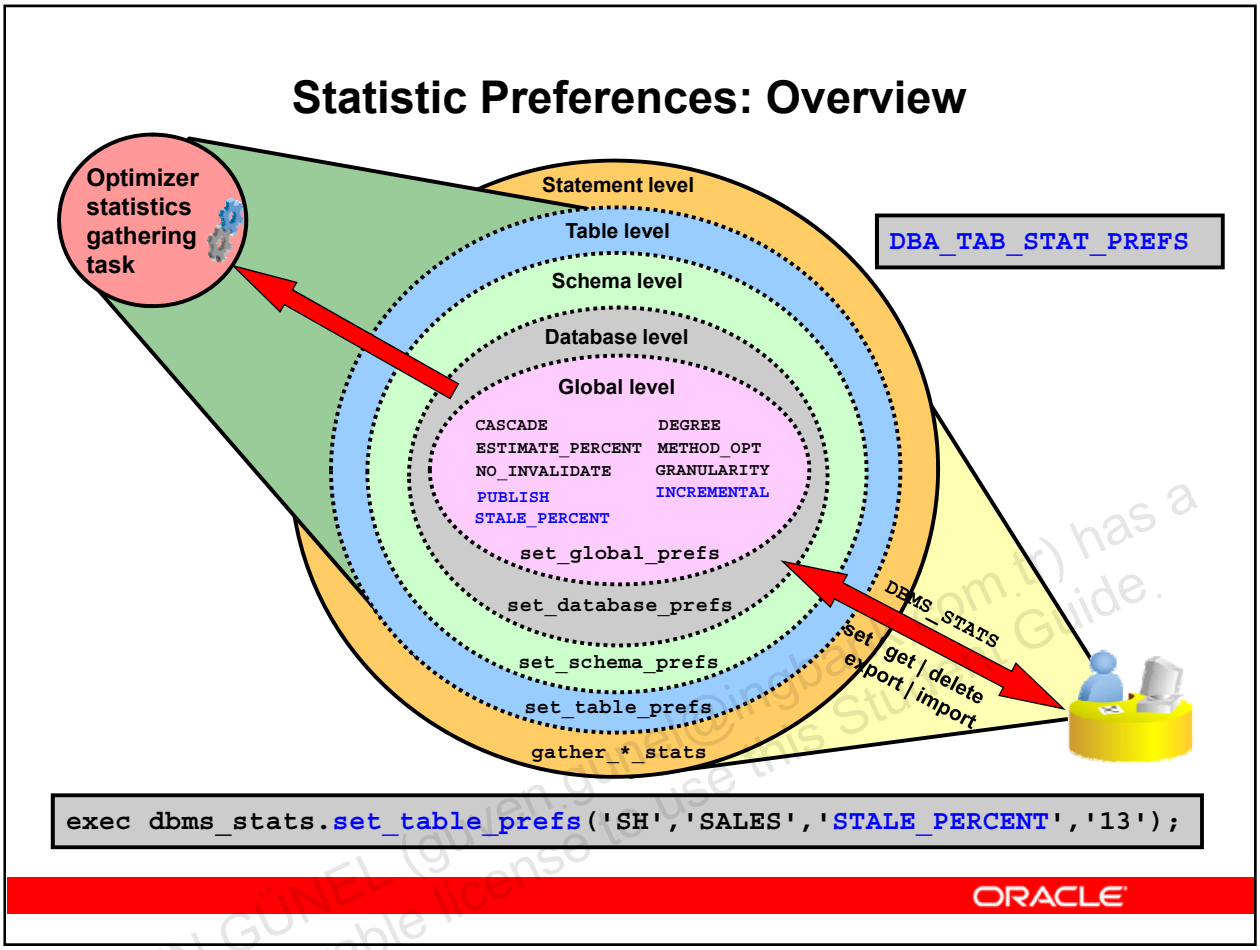


Mechanisms for Gathering Statistics

Oracle Database provides several mechanisms to gather statistics. These are discussed in more detail in the subsequent slides. It is recommended that you use automatic statistics gathering for objects.

Note: When the system encounters a table with missing statistics, it dynamically gathers the necessary statistics needed by the optimizer. However, for certain types of tables, it does not perform dynamic sampling. These include remote tables and external tables. In those cases and also when dynamic sampling has been disabled, the optimizer uses default values for its statistics.

Statistic Preferences: Overview



Statistic Preferences: Overview

The automated statistics-gathering feature was introduced in Oracle Database 10*g*, Release 1 to reduce the burden of maintaining optimizer statistics. However, there were cases where you had to disable it and run your own scripts instead. One reason was the lack of object-level control. Whenever you found a small subset of objects for which the default gather statistics options did not work well, you had to lock the statistics and analyze them separately by using your own options. For example, the feature that automatically tries to determine adequate sample size (`ESTIMATE_PERCENT=AUTO_SAMPLE_SIZE`) does not work well against columns that contain data with very high frequency skews. The only way to get around this issue was to manually specify the sample size in your own script.

The Statistic Preferences feature in Oracle Database 11*g* introduces flexibility so that you can rely more on the automated statistics-gathering feature to maintain the optimizer statistics when some objects require settings that are different from the database default.

This feature allows you to associate the statistics-gathering options that override the default behavior of the `GATHER_*_STATS` procedures and the automated Optimizer Statistics Gathering task at the object or schema level. You can use the `DBMS_STATS` package to manage the gathering statistics options shown in the slide.

You can set, get, delete, export, and import those preferences at the table, schema, database, and global levels. Global preferences are used for tables that do not have preferences,

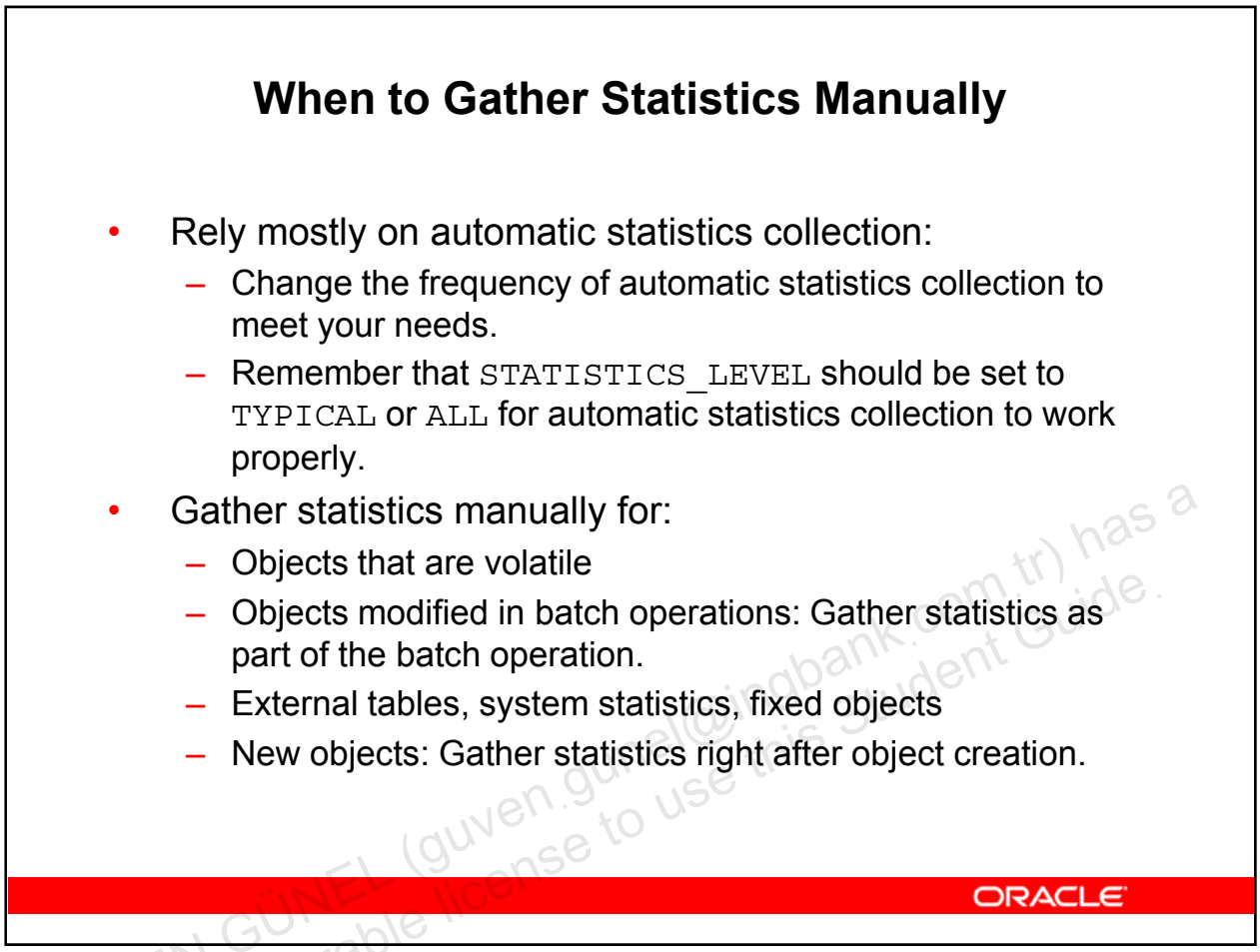
whereas database preferences are used to set preferences on all tables. The preference values that are specified in various ways take precedence from the outer circles to the inner ones (as shown in the slide).

In the graphic in the slide, the last three highlighted options are new in Oracle Database 11*g*, Release 1:

- `CASCADE` gathers statistics on the indexes as well. Index statistics gathering is not parallelized.
- `ESTIMATE_PERCENT` is the estimated percentage of rows used to compute statistics (Null means all rows): The valid range is [0.000001,100]. Use the constant `DBMS_STATS.AUTO_SAMPLE_SIZE` to have the system determine the appropriate sample size for good statistics. This is the recommended default.
- `NO_INVALIDATE` controls the invalidation of dependent cursors of the tables for which statistics are being gathered. It does not invalidate the dependent cursors if set to `TRUE`. The procedure invalidates the dependent cursors immediately if set to `FALSE`. Use `DBMS_STATS.AUTO_INVALIDATE` to have the system decide when to invalidate dependent cursors. This is the default.
- `PUBLISH` is used to decide whether to publish the statistics to the dictionary or to store them in a pending area before.
- `STALE_PERCENT` is used to determine the threshold level at which an object is considered to have stale statistics. The value is a percentage of rows modified since the last statistics gathering. The example changes the 10 percent default to 13 percent for `SH.SALES` only.
- `DEGREE` determines the degree of parallelism used to compute statistics. The default for degree is null, which means use the table default value specified by the `DEGREE` clause in the `CREATE TABLE` or `ALTER TABLE` statement. Use the constant `DBMS_STATS.DEFAULT_DEGREE` to specify the default value based on the initialization parameters. The `AUTO_DEGREE` value determines the degree of parallelism automatically. This is either 1 (serial execution) or `DEFAULT_DEGREE` (the system default value based on the number of CPUs and initialization parameters), depending on the size of the object.
- `METHOD_OPT` is a SQL string used to collect histogram statistics. The default value is `FOR ALL COLUMNS SIZE AUTO`.
- `GRANULARITY` is the granularity of statistics to collect for partitioned tables.
- `INCREMENTAL` is used to gather global statistics on partitioned tables in an incremental way.

It is important to note that you can change default values for the above parameters using the `DBMS_STATS.SET_GLOBAL_PREFS` procedure.

Note: You can describe all the effective statistics preference settings for all relevant tables by using the `DBA_TAB_STAT_PREFS` view.

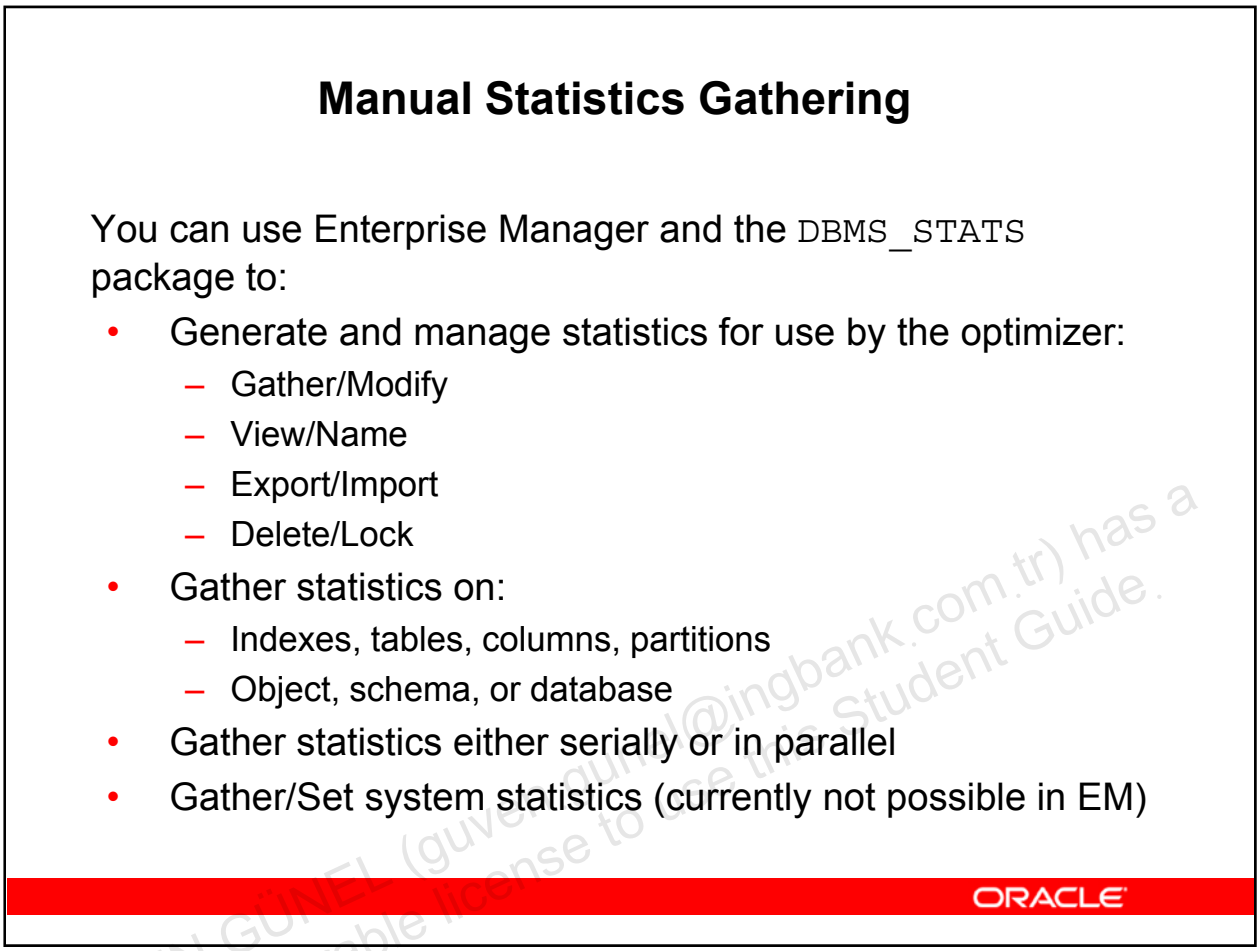


When to Gather Statistics Manually

The automatic statistics gathering mechanism gather statistics on schema objects in the database for which statistics are absent or stale. It is important to determine when and how often to gather new statistics. The default gathering interval is nightly, but you can change this interval to suit your business needs. You can do so by changing the characteristics of your maintenance windows. Some cases may require manual statistics gathering. For example, the statistics on tables that are significantly modified during the day may become stale. There are typically two types of such objects:

- Volatile tables that are modified significantly during the course of the day
- Objects that are the target of large bulk loads that add 10% or more to the object's total size between statistics-gathering intervals

For external tables, statistics are only collected manually using `GATHER_TABLE_STATS`. Sampling on external tables is not supported, so the `ESTIMATE_PERCENT` option should be explicitly set to null. Because data manipulation is not allowed against external tables, it is sufficient to analyze external tables when the corresponding file changes. Other areas in which statistics need to be manually gathered are the system statistics and fixed objects, such as the dynamic performance tables. These statistics are not automatically gathered.



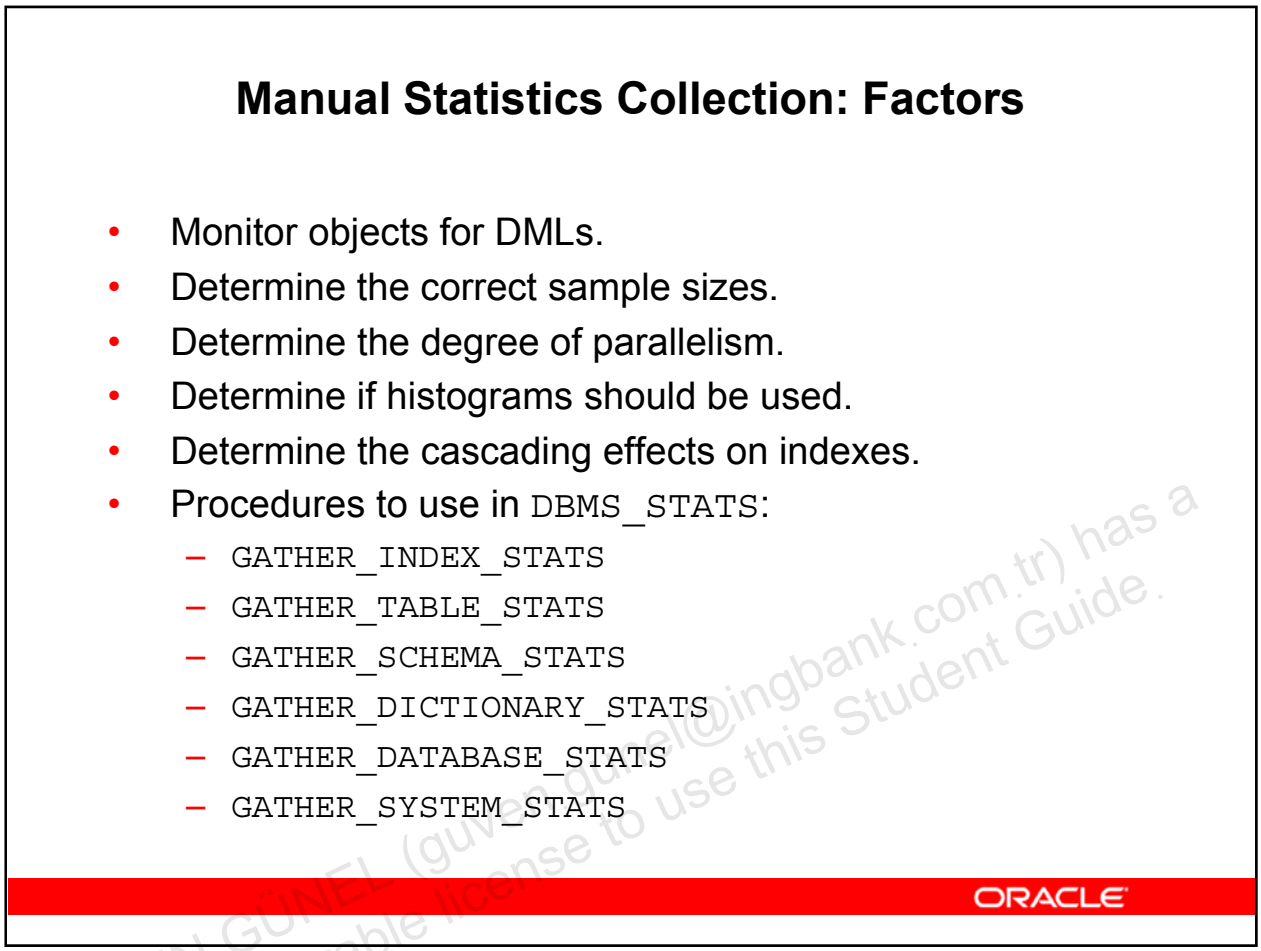
Manual Statistics Gathering

Both Enterprise Manager and the DBMS_STATS package enable you to manually generate and manage statistics for the optimizer. You can use the DBMS_STATS package to gather, modify, view, export, import, lock, and delete statistics. You can also use this package to identify or name gathered statistics. You can gather statistics on indexes, tables, columns, and partitions at various granularity: object, schema, and database level.

DBMS_STATS gathers only statistics needed for optimization; it does not gather other statistics. For example, the table statistics gathered by DBMS_STATS include the number of rows, number of blocks currently containing data, and average row length, but not the number of chained rows, average free space, or number of unused data blocks.

Note: Do not use the COMPUTE and ESTIMATE clauses of the ANALYZE statement to collect optimizer statistics. These clauses are supported solely for backward compatibility and may be removed in a future release. The DBMS_STATS package collects a broader, more accurate set of statistics, and gathers statistics more efficiently. You may continue to use the ANALYZE statement for other purposes not related to the optimizer statistics collection:

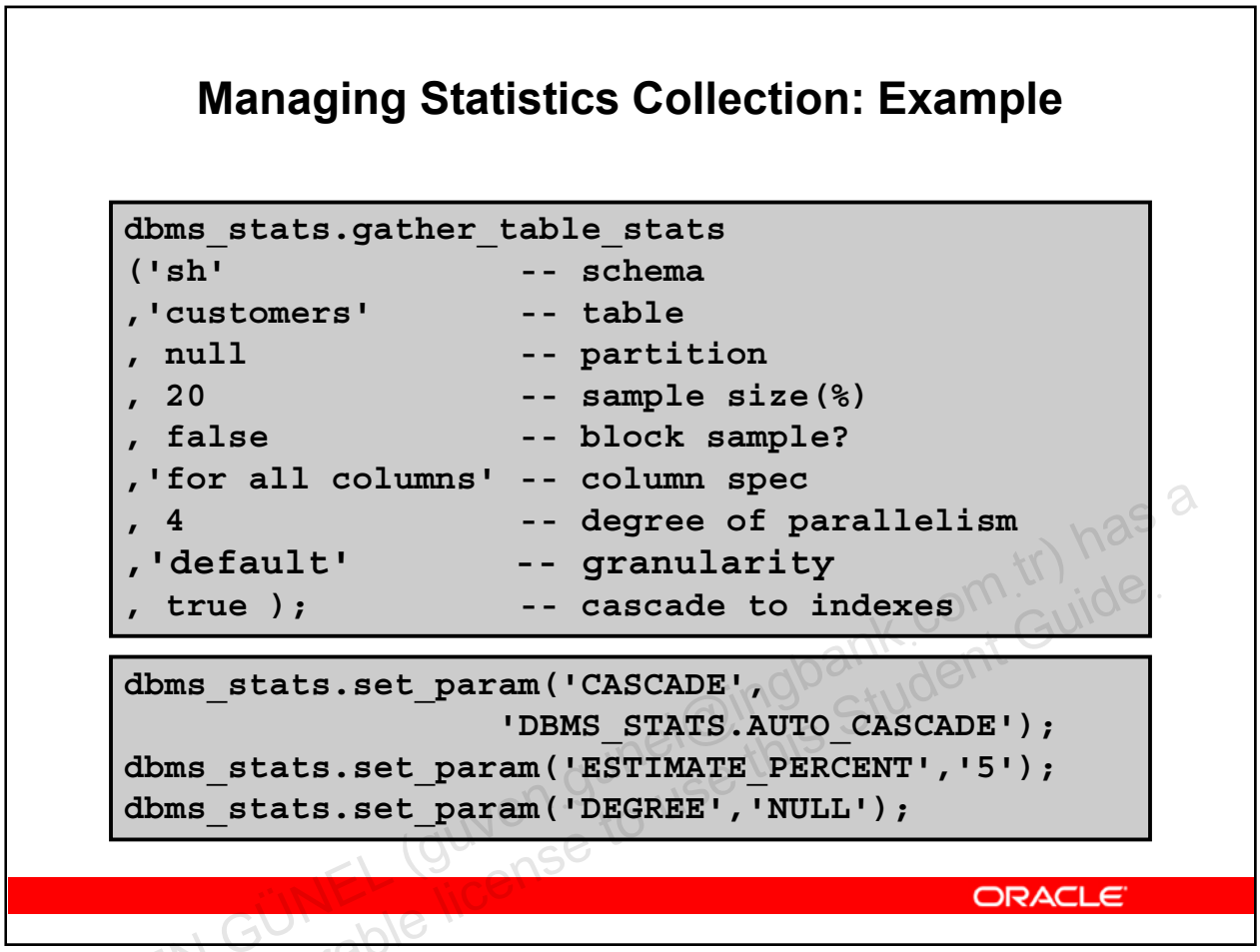
- To use the VALIDATE or LIST CHAINED ROWS clauses
- To collect information on free list blocks



Manual Statistics Collection: Factors

When you manually gather optimizer statistics, you must pay special attention to the following factors:

- Monitoring objects for mass data manipulation language (DML) operations and gathering statistics if necessary
- Determining the correct sample sizes
- Determining the degree of parallelism to speed up queries on large objects
- Determining if histograms should be created on columns with skewed data
- Determining whether changes on objects cascade to any dependent indexes



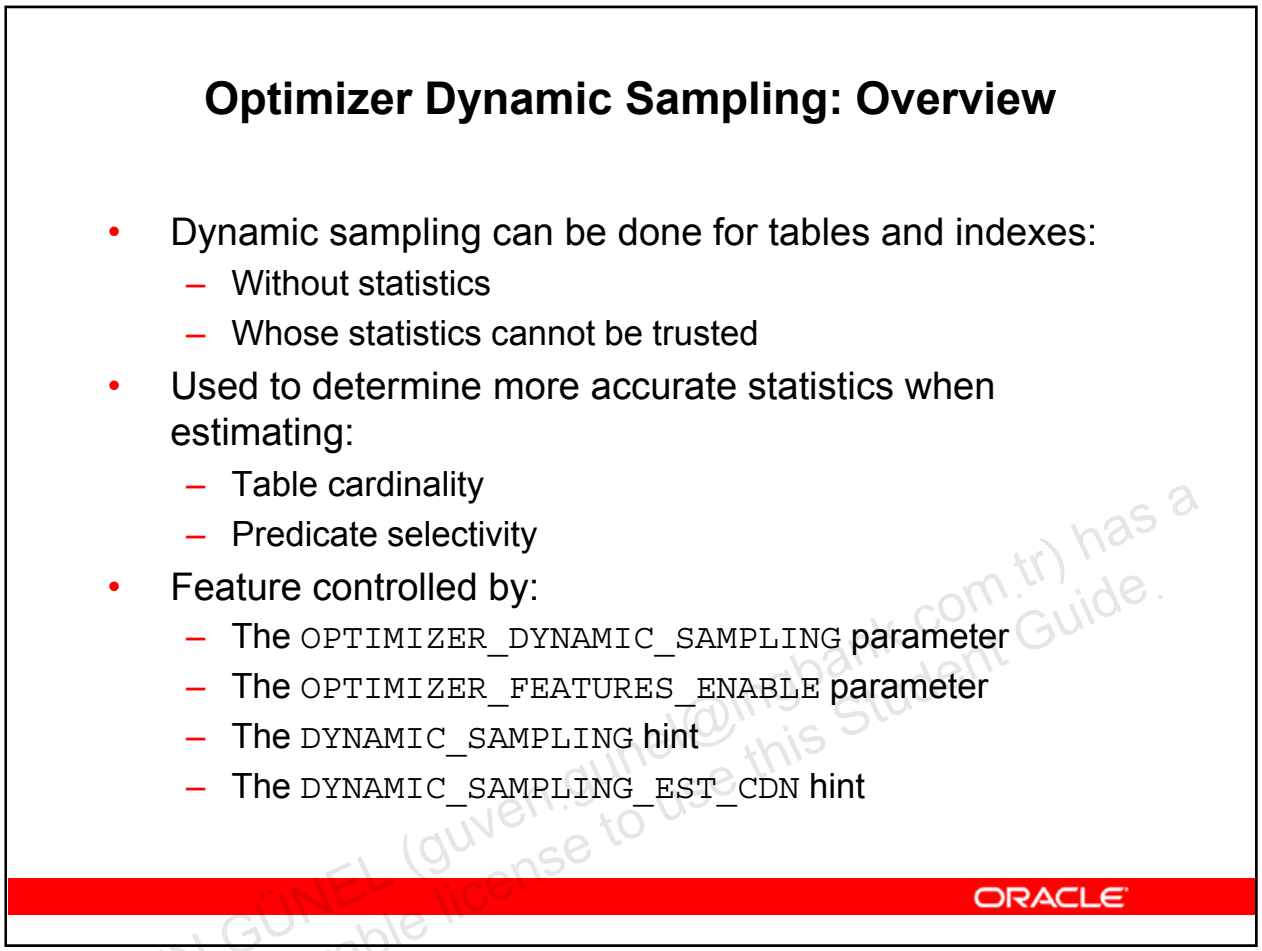
Managing Statistics Collection: Example

The first example uses the DBMS_STATS package to gather statistics on the CUSTOMERS table of the SH schema. It uses some of the options discussed in the previous slides.

Setting Parameter Defaults

You can use the SET_PARAM procedure in DBMS_STATS to set default values for parameters of all DBMS_STATS procedures. The second example in the slide shows this usage. You can also use the GET_PARAM function to get the current default value of a parameter.

Note: Granularity of statistics to collect is pertinent only if the table is partitioned. This parameter determines at which level statistics should be gathered. This can be at the partition, subpartition, or table level.



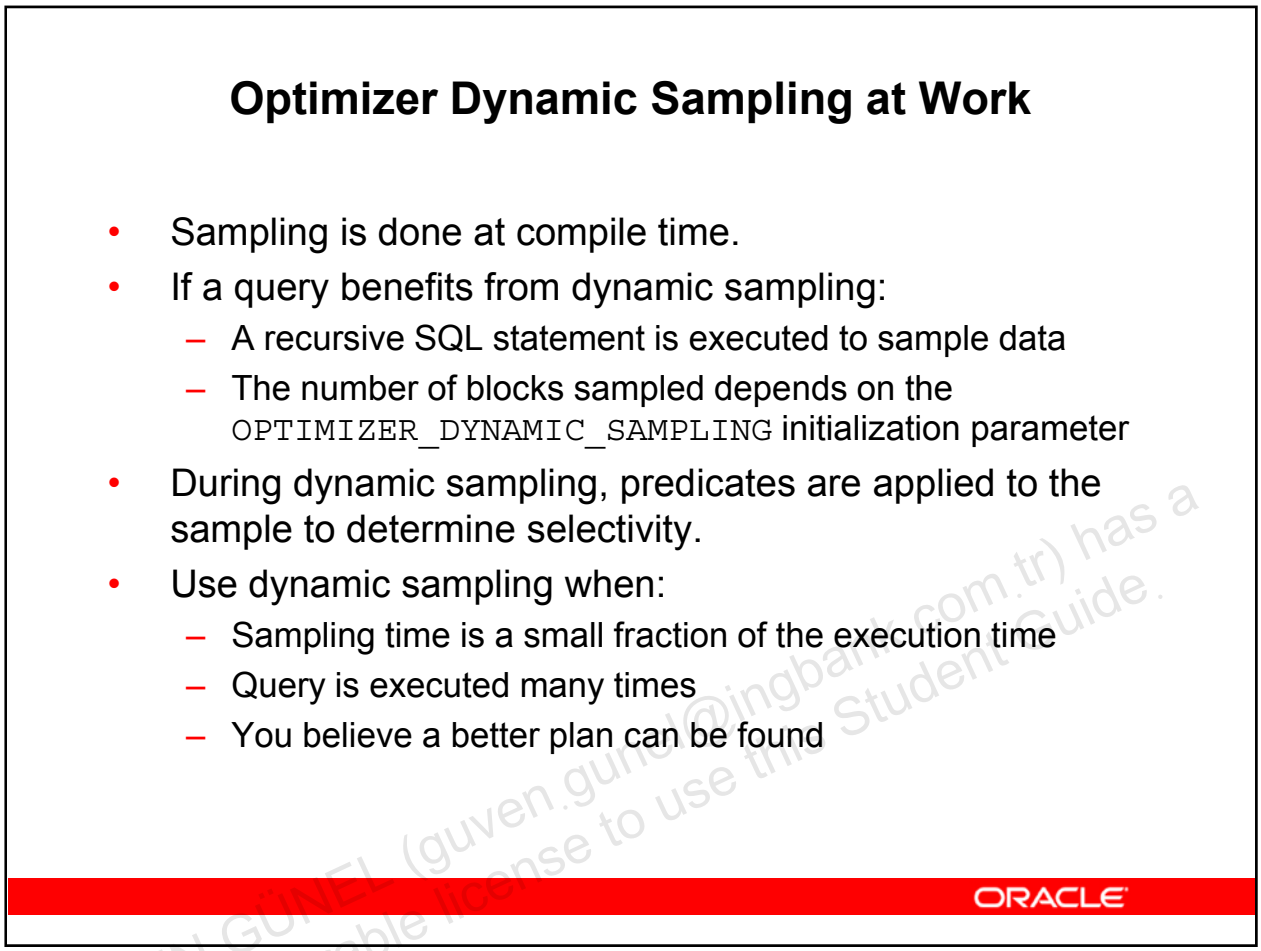
Optimizer Dynamic Sampling: Overview

Dynamic sampling improves server performance by determining more accurate selectivity and cardinality estimates that allow the optimizer to produce better performing plans. For example, although it is recommended that you collect statistics on all of your tables for use by the CBO, you may not gather statistics for your temporary tables and working tables used for intermediate data manipulation. In these cases, the CBO provides a value through a simple algorithm that can lead to a suboptimal execution plan. You can use dynamic sampling to:

- Estimate single-table predicate selectivities when collected statistics cannot be used or are likely to lead to significant errors in estimation
- Estimate table cardinality for tables and relevant indexes without statistics or for tables whose statistics are too outdated to be reliable

You control dynamic sampling with the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter. The `DYNAMIC_SAMPLING` and `DYNAMIC_SAMPLING_EST_CDN` hints can be used to further control dynamic sampling.

Note: The `OPTIMIZER_FEATURES_ENABLE` initialization parameter turns off dynamic sampling if set to a version prior to 9.2.



Optimizer Dynamic Sampling at Work

The primary performance attribute is compile time. The system determines at compile time whether a query would benefit from dynamic sampling. If so, a recursive SQL statement is issued to scan a small random sample of the table's blocks, and to apply the relevant single table predicates to estimate predicate selectivities.

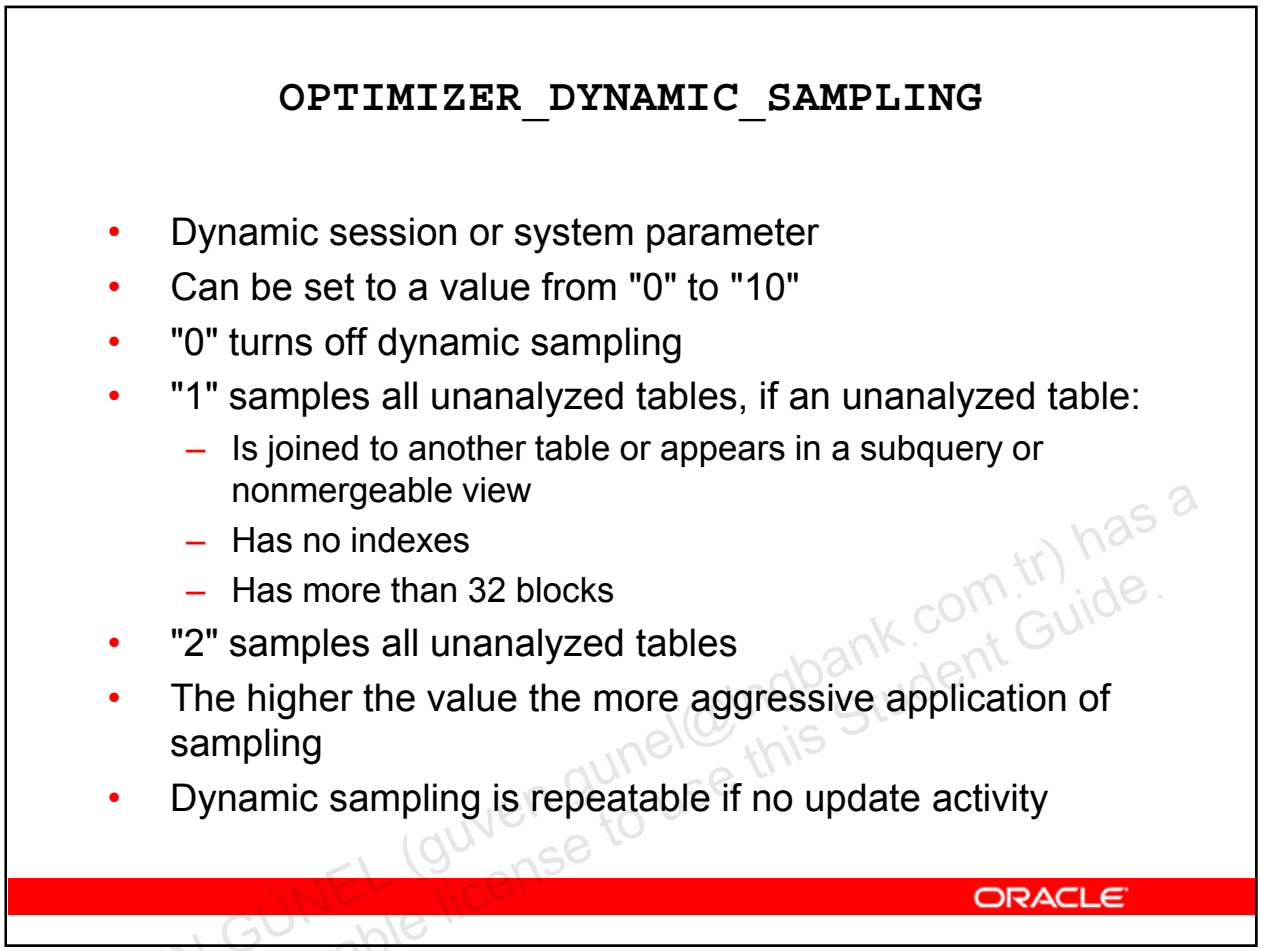
Depending on the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter, a certain number of blocks is read by the dynamic sampling query.

For a query that normally completes quickly (in less than a few seconds), you do not want to incur the cost of dynamic sampling. However, dynamic sampling can be beneficial under any of the following conditions:

- A better plan can be found using dynamic sampling.
- The sampling time is a small fraction of total execution time for the query.
- The query is executed many times.

Note: Dynamic sampling can be applied to a subset of a single table's predicates and combined with standard selectivity estimates of predicates for which dynamic sampling is not done.

OPTIMIZER_DYNAMIC_SAMPLING



OPTIMIZER_DYNAMIC_SAMPLING

You control dynamic sampling with the `OPTIMIZER_DYNAMIC_SAMPLING` parameter, which can be set to a value from "0" to "10." A value of "0" means dynamic sampling is not done.

A value of "1" means dynamic sampling is performed on all unanalyzed tables if the following criteria are met:

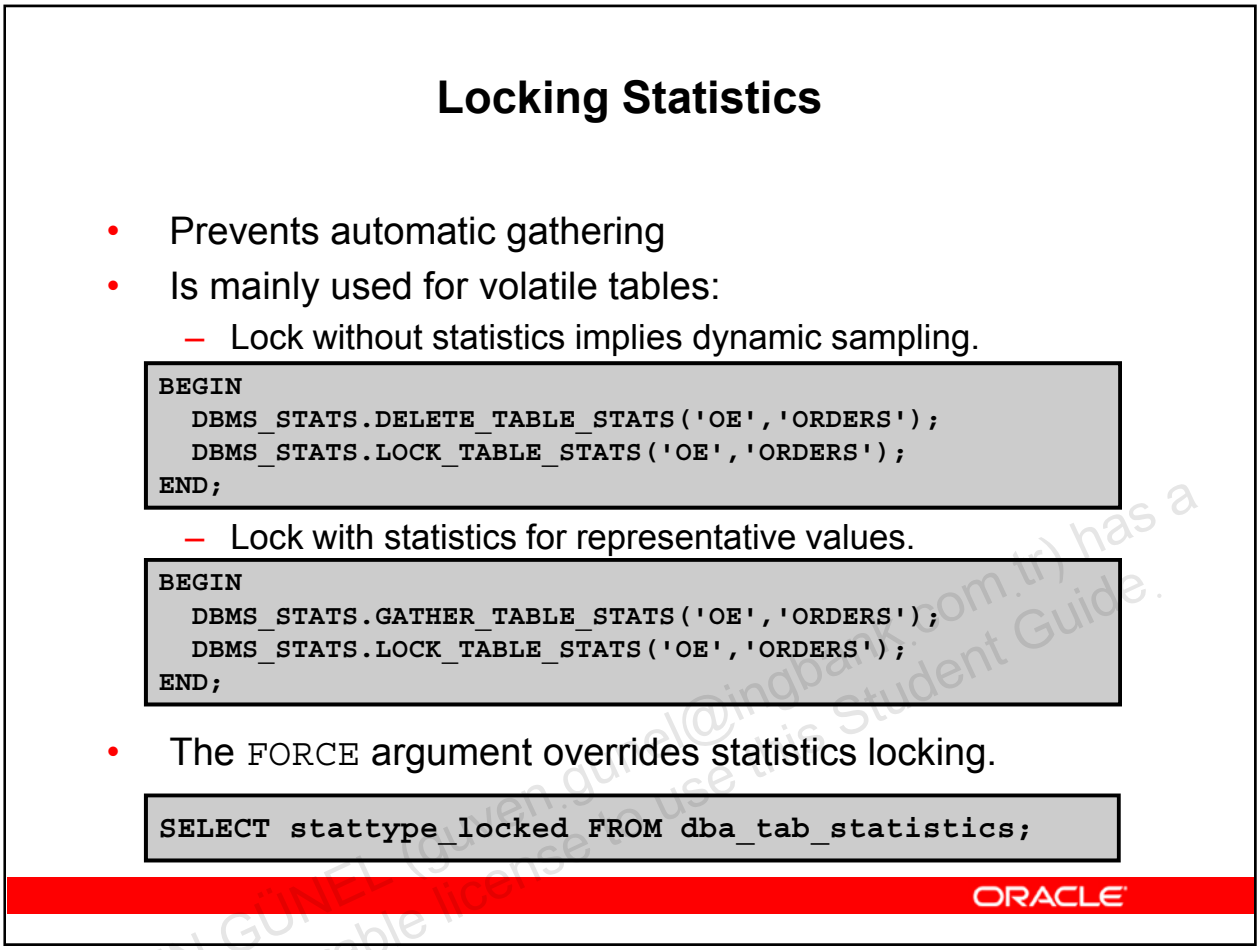
- There is at least one unanalyzed table in the query.
- This unanalyzed table is joined to another table or appears in a subquery or nonmergeable view.
- This unanalyzed table has no indexes.
- This unanalyzed table has more blocks than the default number of blocks that would be used for dynamic sampling of this table. This default number is 32.

The default value is "2" if `OPTIMIZER_FEATURES_ENABLE` is set to 10.0.0 or higher. At this level, the system applies dynamic sampling to all unanalyzed tables. The number of blocks sampled is two times the default number of dynamic sampling blocks (32).

Increasing the value of the parameter results in more aggressive application of dynamic sampling, in terms of both the type of tables sampled (analyzed or unanalyzed) and the amount of I/O spent on sampling.

Note: Dynamic sampling is repeatable if no rows have been inserted, deleted, or updated in the table being sampled since the previous sample operation.

Locking Statistics



Locking Statistics

Starting with Oracle Database 10*g*, you can lock statistics on a specified table with the `LOCK_TABLE_STATS` procedure of the `DBMS_STATS` package. You can lock statistics on a table without statistics or set them to `NULL` using the `DELETE_*_STATS` procedures to prevent automatic statistics collection so that you can use dynamic sampling on a volatile table with no statistic. You can also lock statistics on a volatile table at a point when it is fully populated so that the table statistics are more representative of the table population.

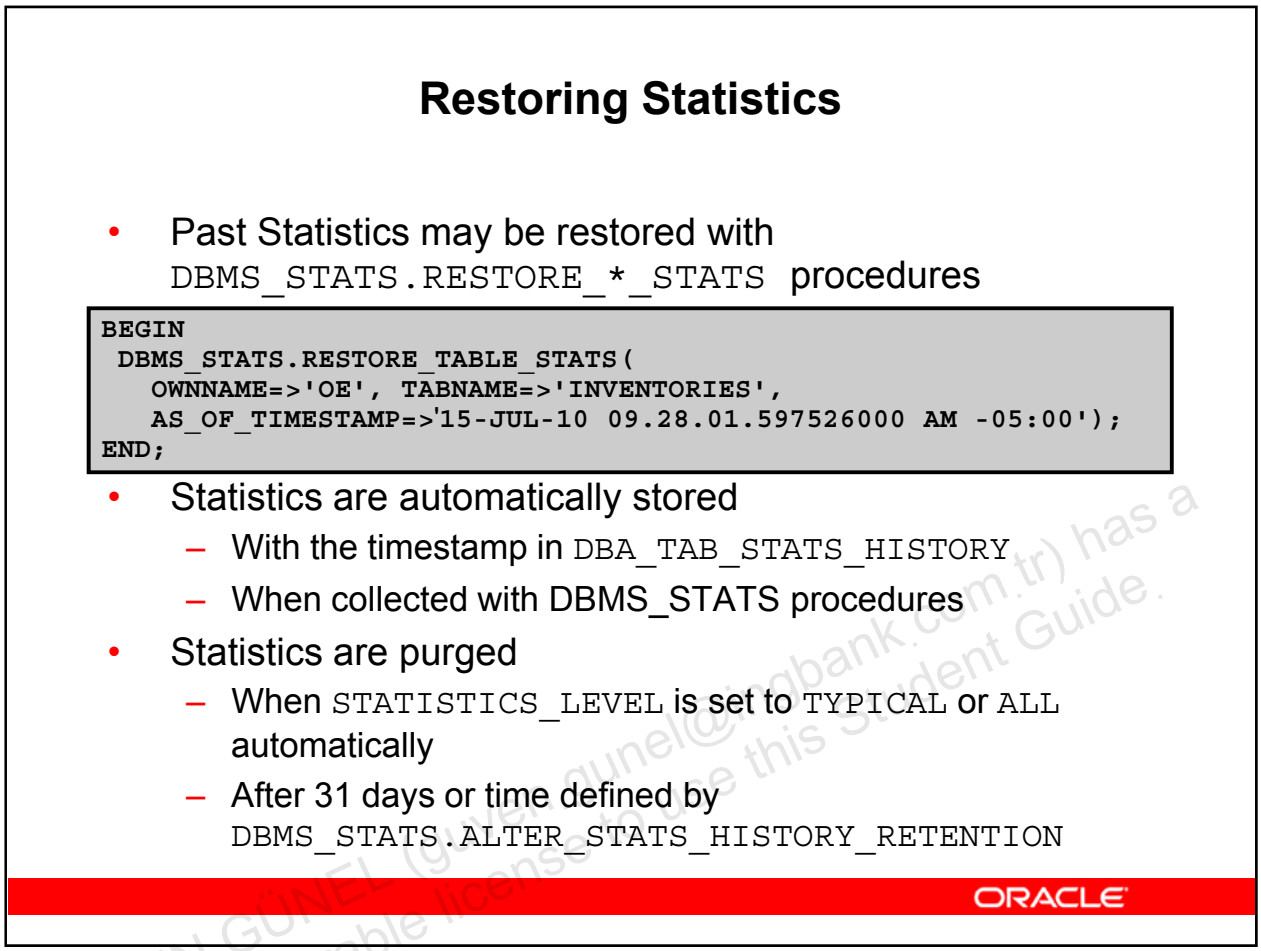
You can also lock statistics at the schema level by using the `LOCK_SCHEMA_STATS` procedure. You can query the `STATTYPE_LOCKED` column in the {`USER` | `ALL` | `DBA`}`_TAB_STATISTICS` view to determine whether the statistics on the table are locked.

You can use the `UNLOCK_TABLE_STATS` procedure to unlock the statistics on a specified table.

You can set the value of the `FORCE` parameter to `TRUE` to overwrite the statistics even if they are locked. The `FORCE` argument is found in the following `DBMS_STATS` procedures: `DELETE_*_STATS`, `IMPORT_*_STATS`, `RESTORE_*_STATS`, and `SET_*_STATS`.

Note: When you lock the statistics on a table, all the dependent statistics are considered locked. This includes table statistics, column statistics, histograms, and dependent index statistics.

Restoring Statistics



Restoring Statistics

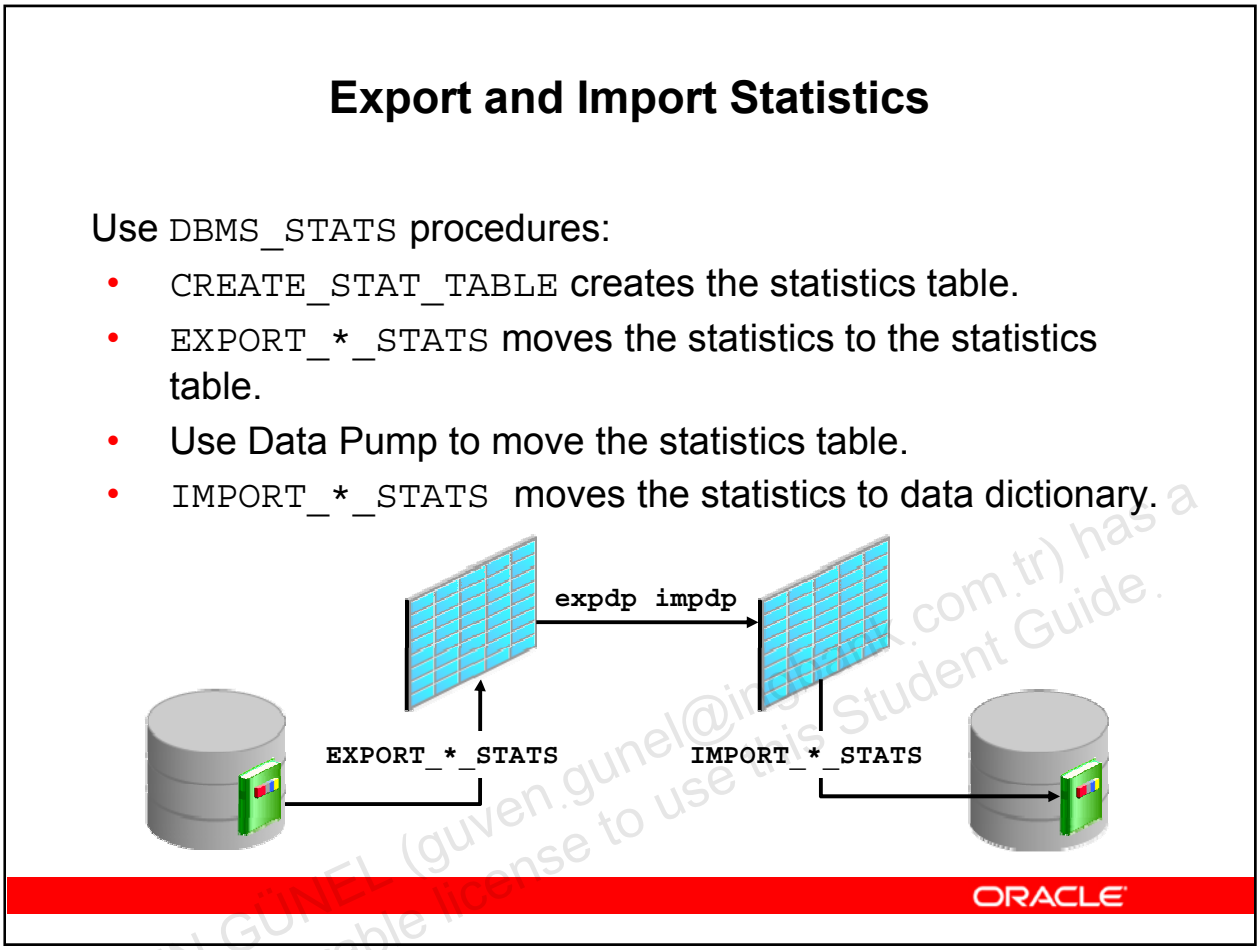
Old versions of statistics are saved automatically whenever statistics in dictionary are modified with the DBMS_STATS procedures. You can restore statistics using RESTORE procedures of DBMS_STATS package. These procedures use a time stamp as an argument and restore statistics as of that time stamp. This is useful when newly collected statistics lead to sub-optimal execution plans and the administrator wants to revert to the previous set of statistics.
Note: the ANALYZE command does not store old statistics.

There are dictionary views that can be used to determine the time stamp for restoration of statistics. The views *_TAB_STATS_HISTORY views (ALL, DBA, or USER) contain a history of table statistics modifications. For the example in the slide the timestamp was determined by:

```
select stats_update_time from dba_tab_stats_history
    where table_name = 'INVENTORIES'
```

The database purges old statistics automatically at regular intervals based on the statistics history retention setting and the time of the recent analysis of the system. You can configure retention using the DBMS_STATS .ALTER_STATS_HISTORY_RETENTION procedure. The default value is 31 days, which means that you would be able to restore the optimizer statistics to any time in last 31 days.

Export and Import Statistics



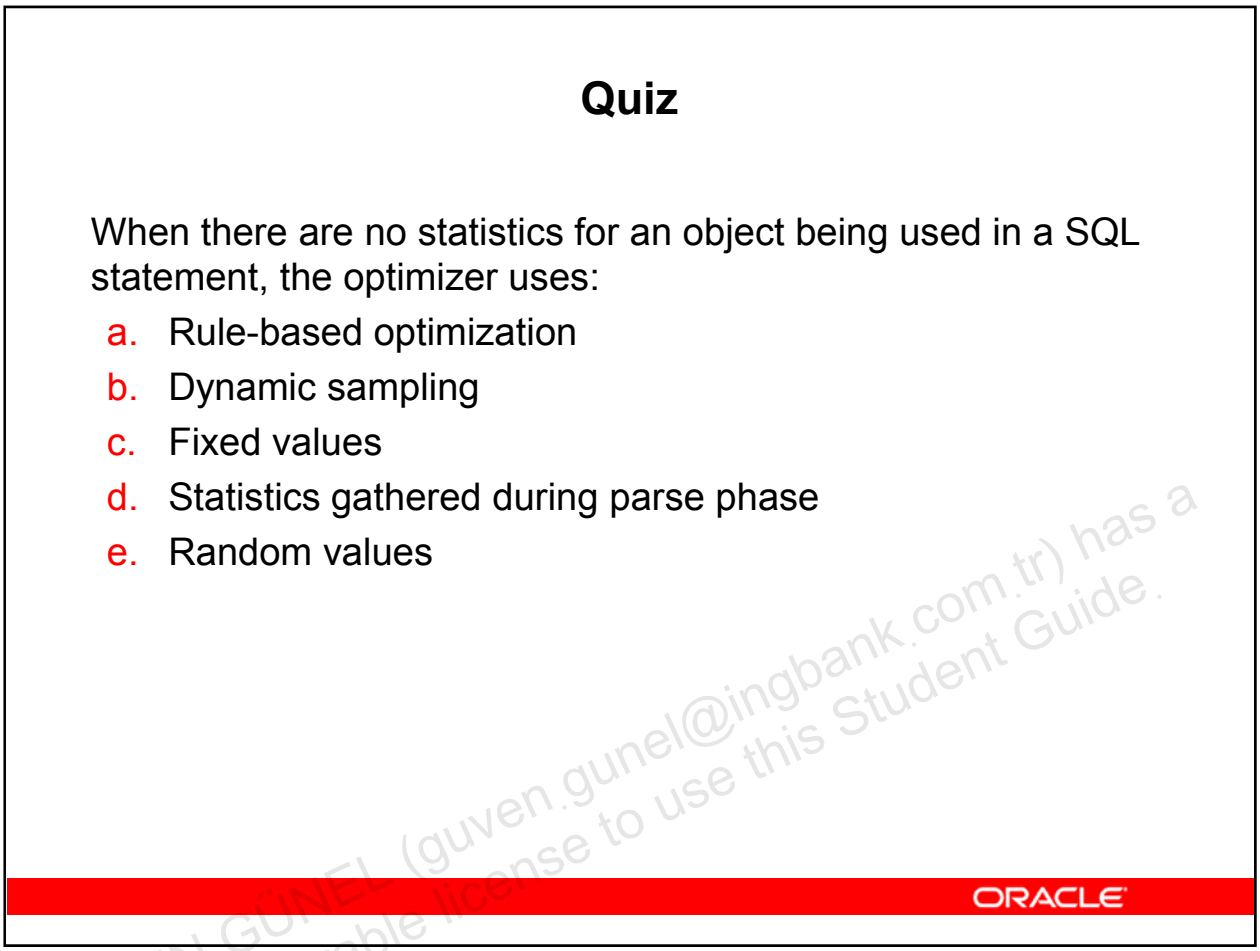
Export and Import Statistics

You can export and import statistics from the data dictionary to user-owned tables, enabling you to create multiple versions of statistics for the same schema. You can also copy statistics from one database to another database. You may want to do this to copy the statistics from a production database to a scaled-down test database.

Before exporting statistics, you first need to create a table for holding the statistics. The procedure DBMS_STATS.CREATE_STAT_TABLE creates the statistics table. After table creation, you can export statistics from the data dictionary into the statistics table using the DBMS_STATS.EXPORT_*_STATS procedures. You can then import statistics using the DBMS_STATS.IMPORT_*_STATS procedures.

The optimizer does not use statistics stored in a user-owned table. The only statistics used by the optimizer are the statistics stored in the data dictionary. To have the optimizer use the statistics in a user-owned tables, you must import those statistics into the data dictionary using the statistics import procedures.

To move statistics from one database to another, you must first export the statistics on the first database, then copy the statistics table to the second database, using the Data Pump Export and Import utilities or other mechanisms, and finally import the statistics into the second database.



Answer: b, c

Quiz

The optimizer depends on accurate statistics to produce the best execution plans. The automatic statistics gathering (AGS) task does not gather statistics on everything. Which objects require you to gather statistics manually?

a. External tables
b. Data dictionary
c. Fixed objects
d. Volatile tables
e. System statistics

Answer: a, c, e

Quiz

There is a very volatile table in the database. The size of the table changes by more than 50% daily. What steps are part of the procedure to force dynamic sampling?

a. Delete statistics.
b. Lock statistics.
c. Gather statistics when the table is at its largest.
d. Set `DYNAMIC_SAMPLING=9`.
e. Set `DYNAMIC_SAMPLING=0`.
f. Allow the `DYNAMIC_SAMPLING` parameter to default.

ORACLE

Answer: a, b, f

Summary

In this lesson, you should have learned how to:

- Collect optimizer statistics
- Collect system statistics
- Set statistic preferences
- Use dynamic sampling
- Manipulate optimizer statistics

Practice 10: Overview

This practice covers the following topics:

- Using system statistics
- Using automatic statistics gathering