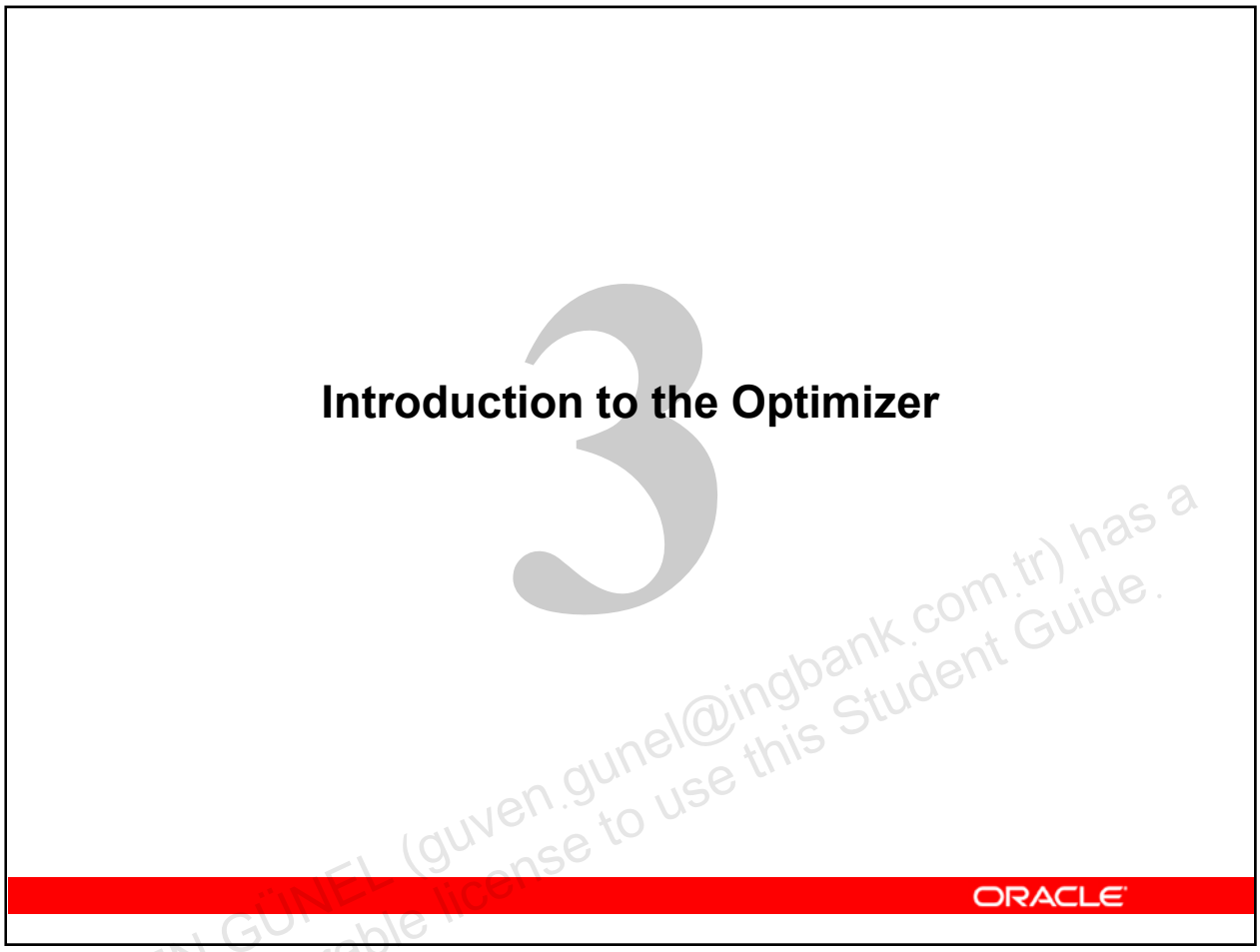
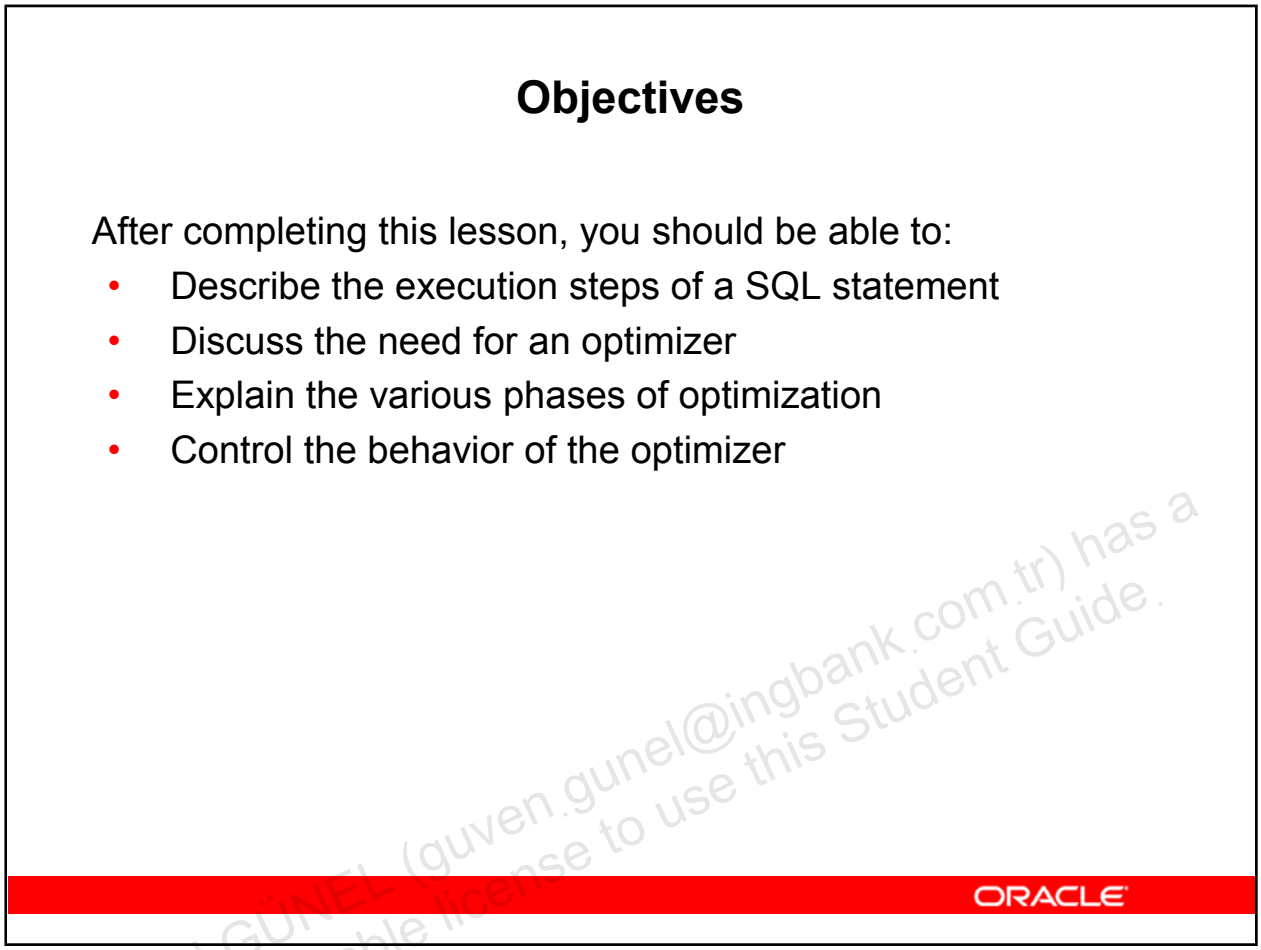
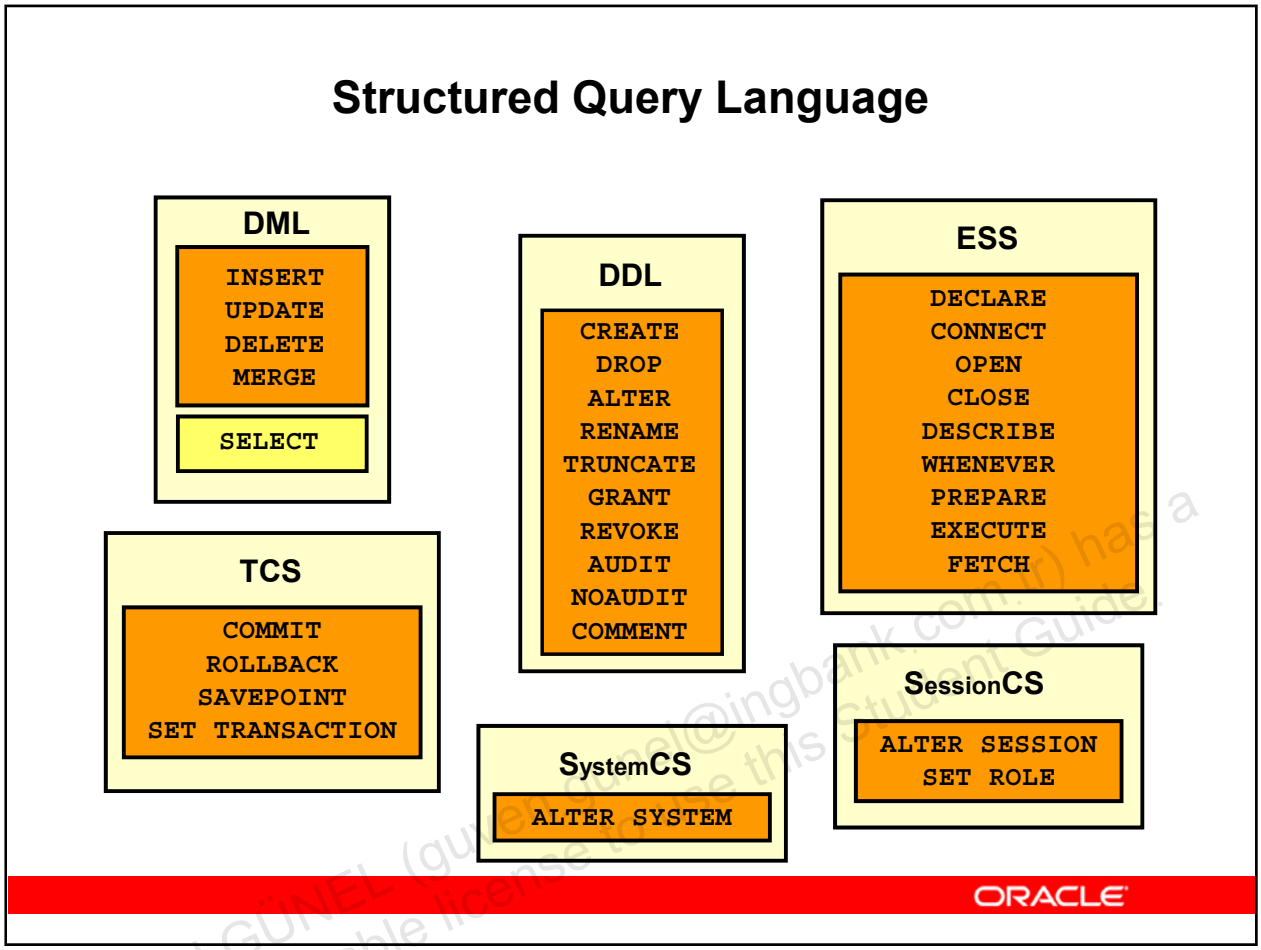
Introduction to the Optimizer

Chapter 3





Structured Query Language



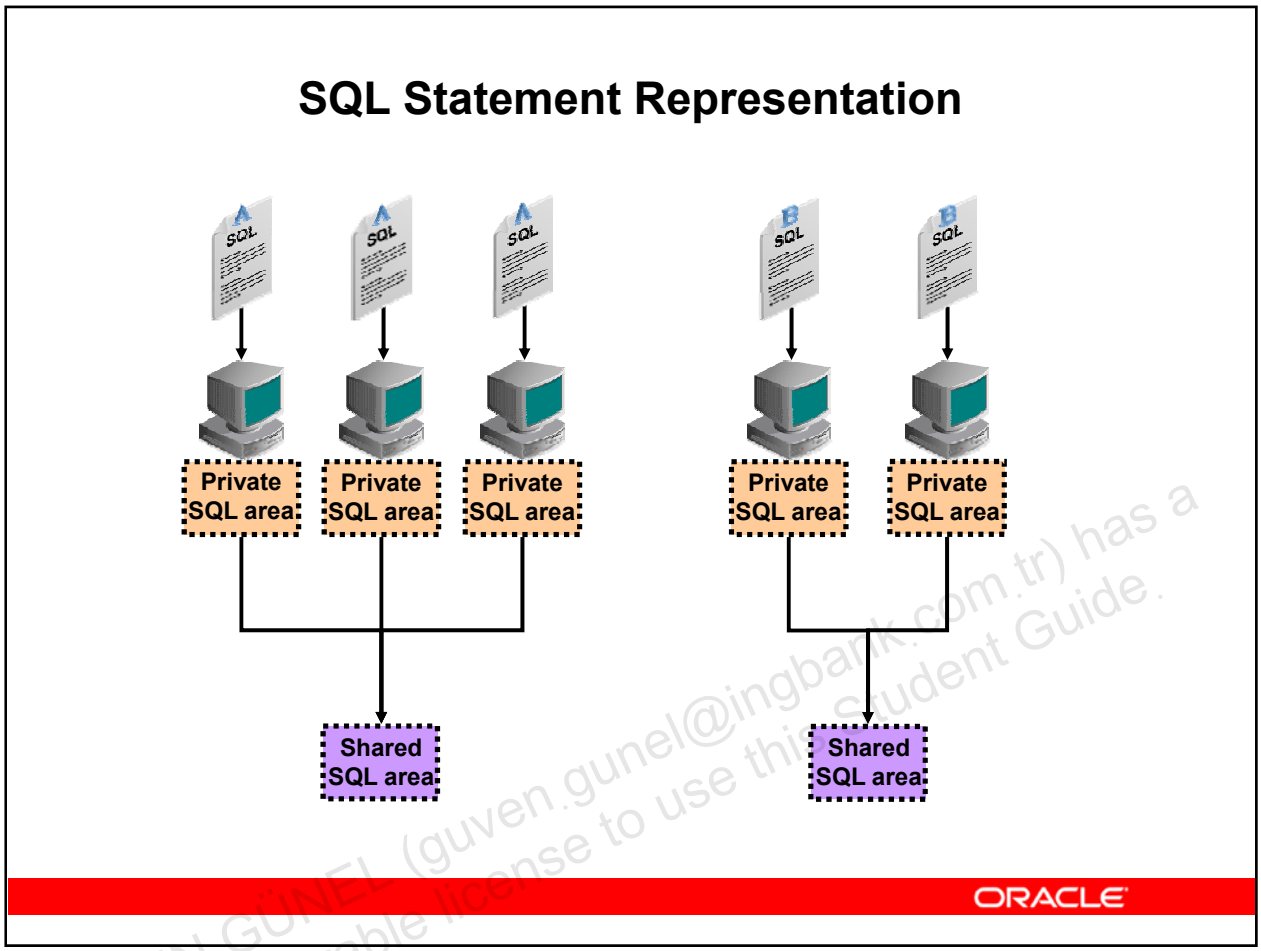
Structured Query Language

All programs and users access data in an Oracle Database with the language SQL Oracle tools and Application programs often allow users to access the database without using SQL directly, but then these applications must use SQL when executing user requests. Oracle Corp strives to comply with industry-accepted standards and participates in SQL standards committees (ANSI and ISO). You can categorize SQL statements into six main sets:

- Data manipulation language (DML) statements manipulate or query data in existing schema objects.
- Data definition language (DDL) statements define, alter the structure of, and drop schema objects.
- Transaction control statements (TCS) manage the changes made by DML statements and group DML statements into transactions.
- System Control statements change the properties of the Oracle Database instance.
- Session Control statements manage the properties of a particular user's session.
- Embedded SQL statements incorporate DDL, DML, and TCS within a procedural language program, such as PL/SQL and Oracle precompilers. This incorporation is done using the statements listed in the slide under the ESS category.

Note: `SELECT` statements are the most used statements. While his course focuses mainly on queries, it is important to note that any type of SQL statement is subject to optimization.

SQL Statement Representation



SQL Statement Representation

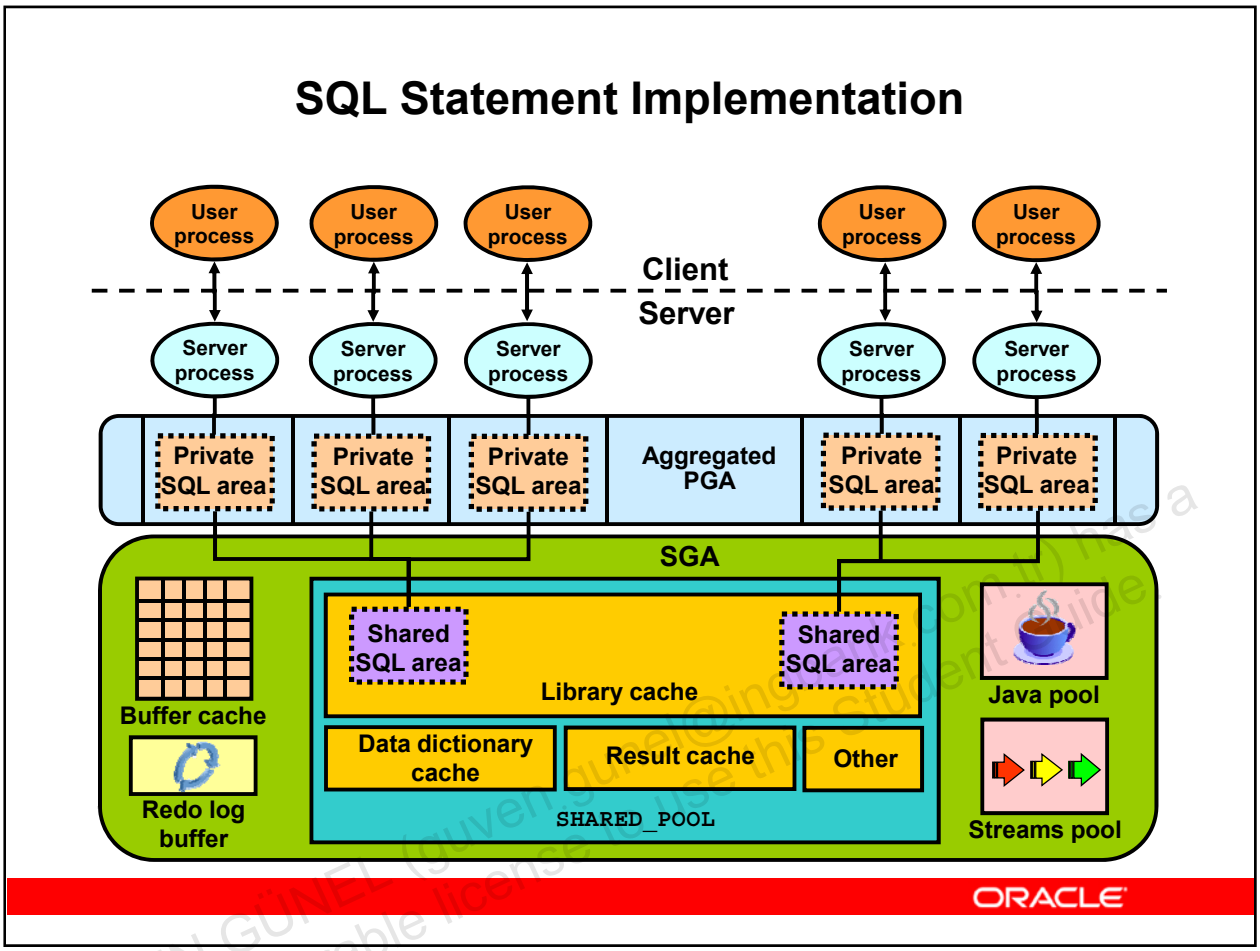
Oracle Database represents each SQL statement it runs with a shared SQL area and a private SQL area. Oracle Database recognizes when two users execute the same SQL statement and reuses the shared SQL area for those users. However, each user must have a separate copy of the statement's private SQL area.

A shared SQL area contains all optimization information necessary to execute the statement whereas a private SQL area contains all run-time information related to a particular execution of the statement.

Oracle Database saves memory by using one shared SQL area for SQL statements run multiple times, which often happens when many users run the same application.

Note: In evaluating whether statements are similar or identical, Oracle Database considers SQL statements issued directly by users and applications, as well as recursive SQL statements issued internally by a DDL statement.

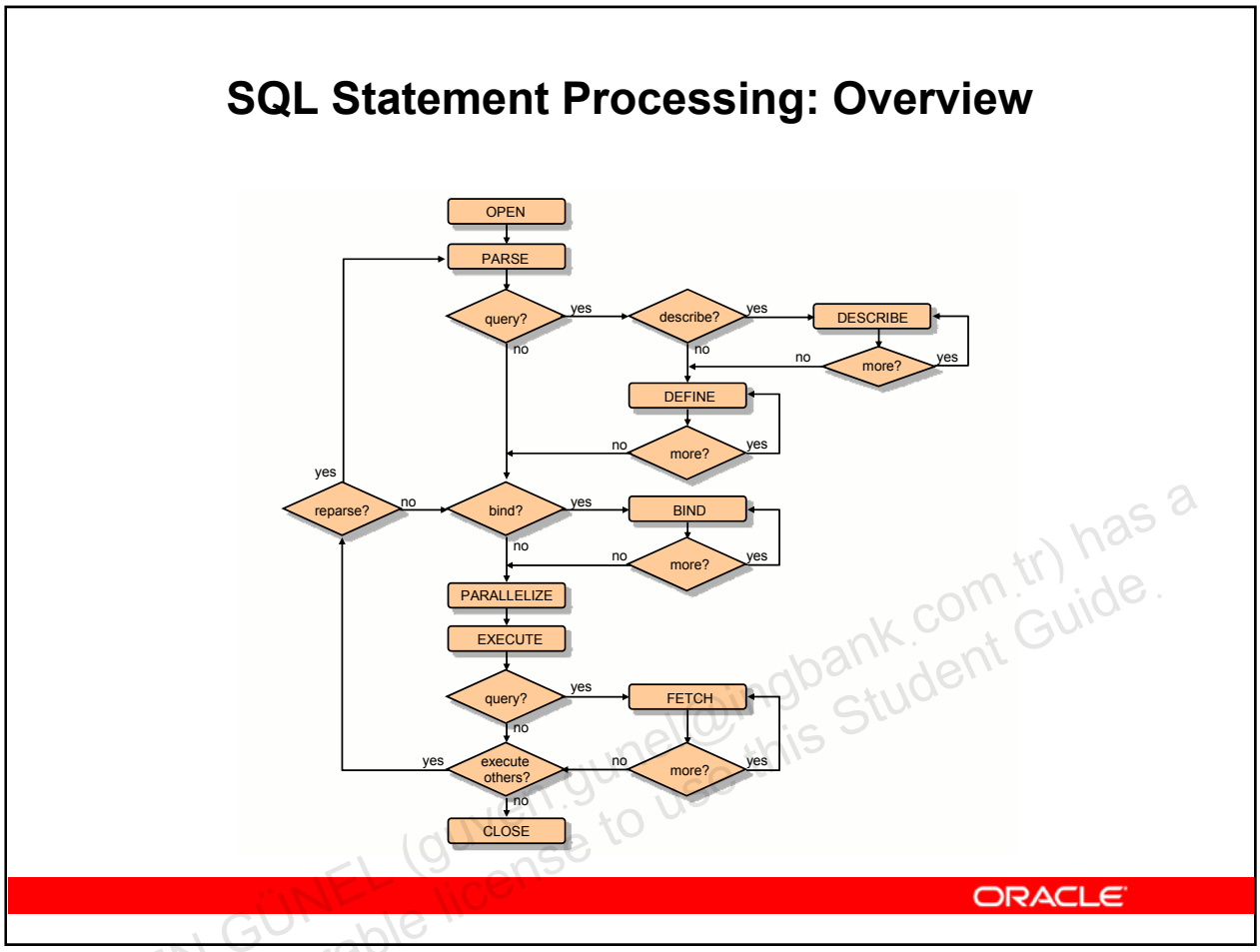
SQL Statement Implementation



SQL Statement Implementation

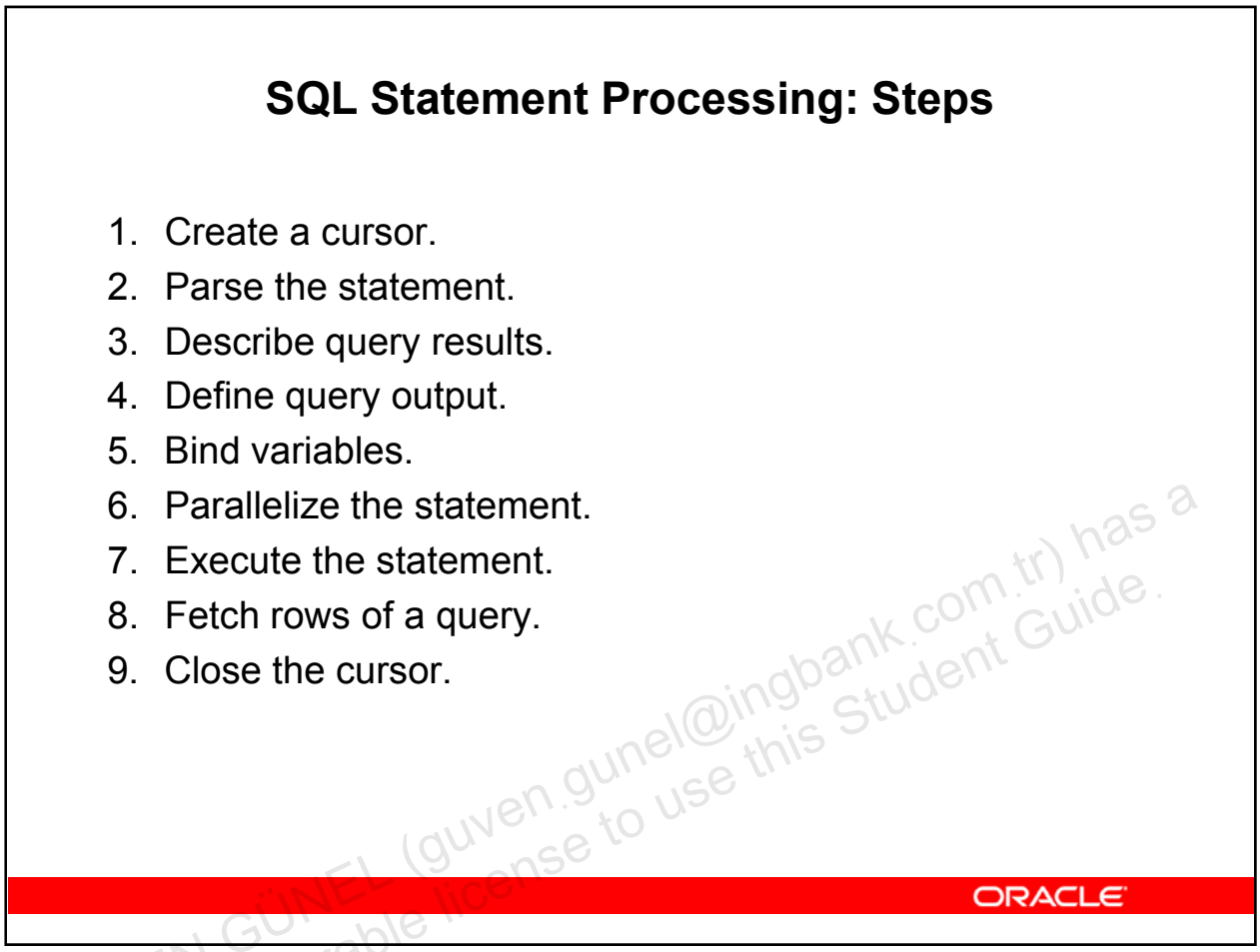
Oracle Database creates and uses memory structures for various purposes. For example, memory stores program codes that are run, data that is shared among users, and private data areas for each connected user.

Oracle Database allocates memory from the shared pool when a new SQL statement is parsed, to store in the shared SQL area. The size of this memory depends on the complexity of the statement. If the entire shared pool has already been allocated, Oracle Database can deallocate items from the pool using a modified least recently used (LRU) algorithm until there is enough free space for the new statement's shared SQL area. If Oracle Database deallocates a shared SQL area, the associated SQL statement must be reparsed and reassigned to another shared SQL area at its next execution.



SQL Statement Processing: Overview

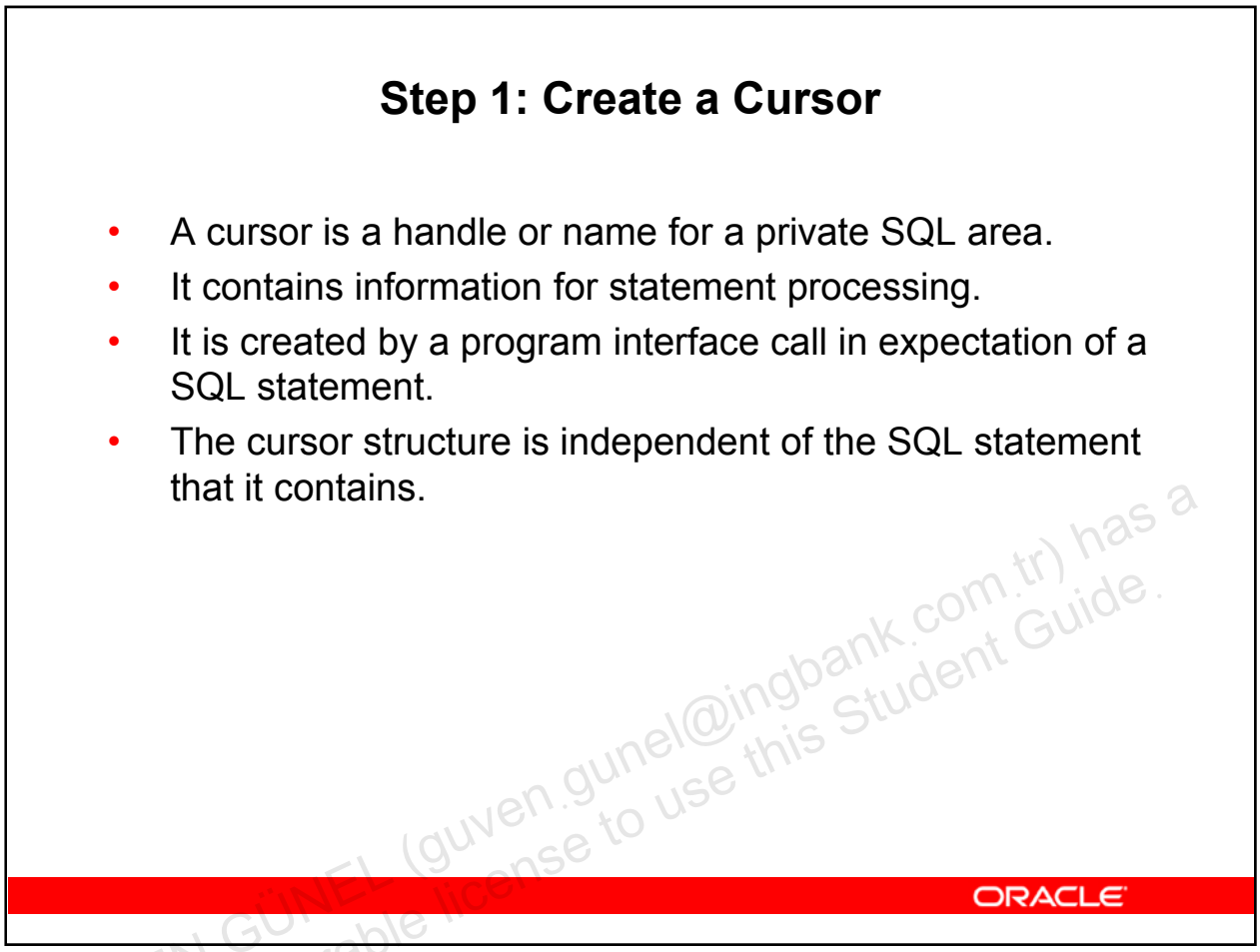
The graphic in the slide shows all the steps involved in query execution and these steps can be found in *Oracle® Database Concepts 11g Release 1 (11.1)*.



SQL Statement Processing: Steps

Note that not all statements require all these steps. For example, nonparallel DDL statements are required in only two steps: Create and Parse.

Parallelizing the statement involves deciding that it can be parallelized as opposed to actually building parallel execution structures.



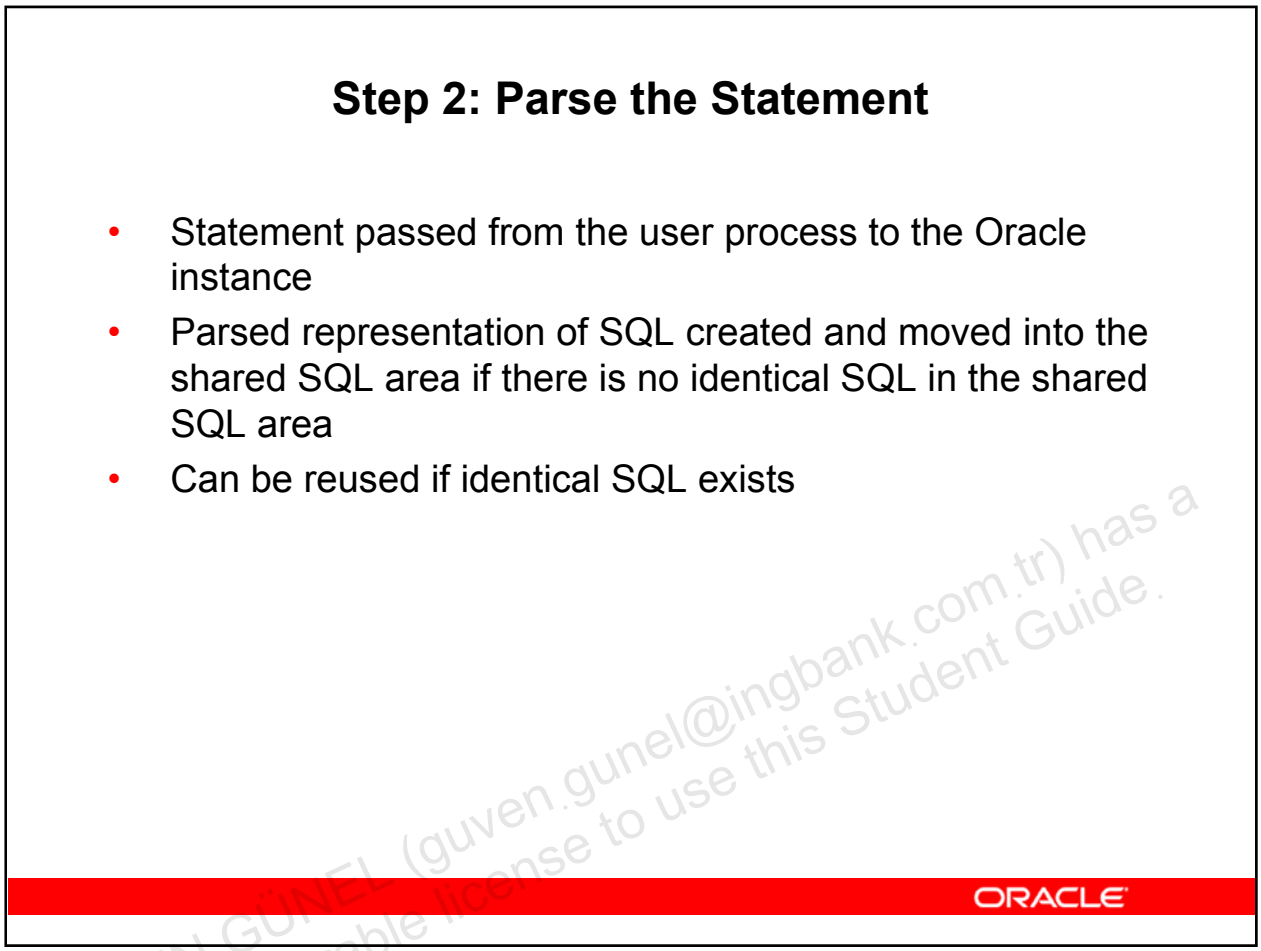
Step 1: Create a Cursor

A cursor can be thought of as an association between a cursor data area in a client program and Oracle server's data structures. Most Oracle tools hide much of cursor handling from the user, but Oracle Call Interface (OCI) programs need the flexibility to be able to process each part of query execution separately. Therefore, precompilers allow explicit cursor declaration. Most of this can also be done using the `DBMS_SQL` package as well.

A handle is similar to the handle on a mug. When you have a hold of the handle, you have a hold of the cursor. It is a unique identifier for a particular cursor that can only be obtained by one process at a time.

Programs must have an open cursor to process a SQL statement. The cursor contains a pointer to the current row. The pointer moves as rows are fetched until there are no more rows left to process.

The following slides use the `DBMS_SQL` package to illustrate cursor management. This may be confusing to people unfamiliar with it; however, it is more friendly than PRO*C or OCI. It is slightly problematic in that it performs `FETCH` and `EXECUTE` together, so the execute phase cannot be separately identified in the trace.



Step 2: Parse the Statement

During parsing, the SQL statement is passed from the user process to the Oracle instance, and a parsed representation of the SQL statement is loaded into a shared SQL area.

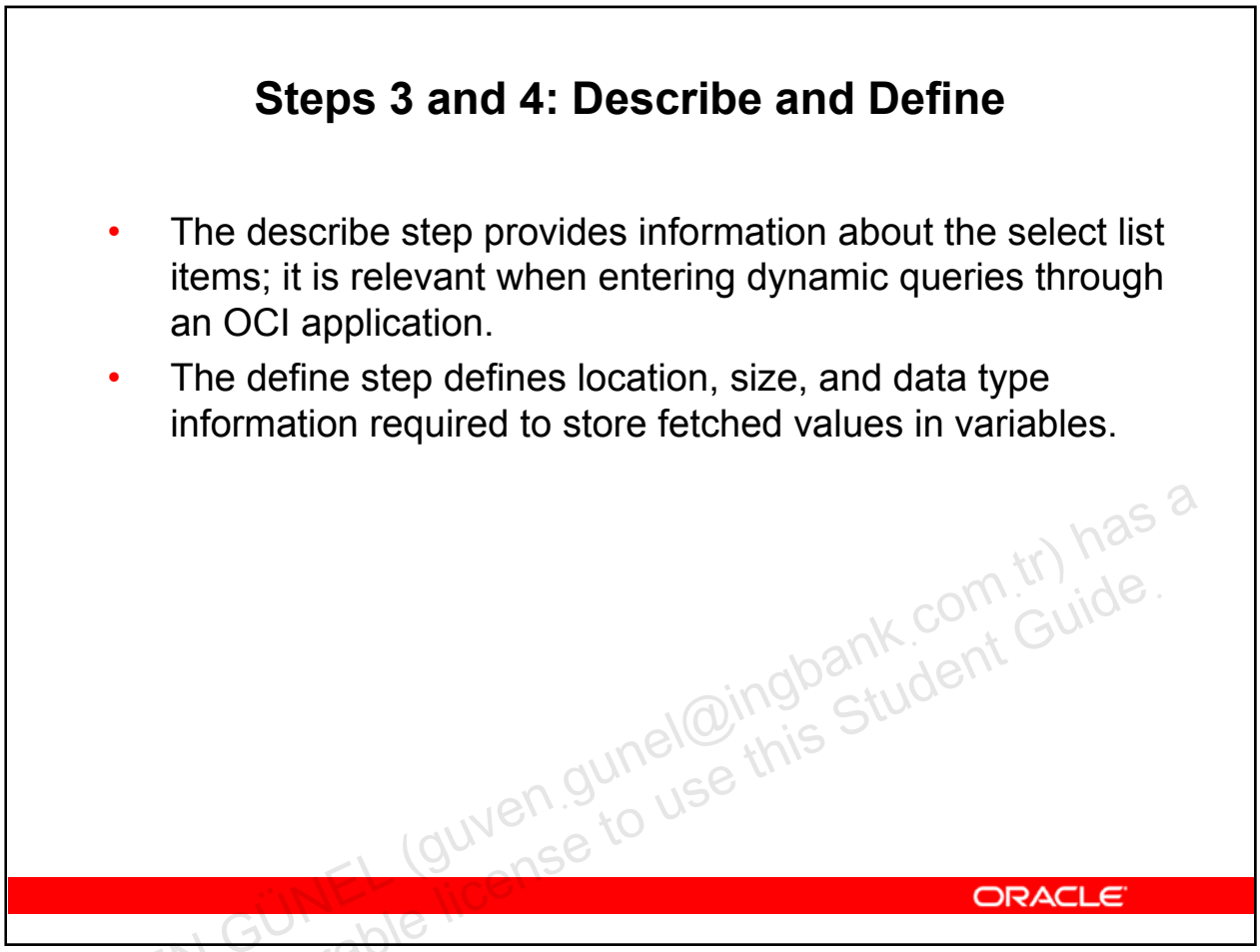
Translation and verification involve checking if the statement already exists in the library cache.

For distributed statements, check for the existence of database links.

Typically, the parse phase is represented as the stage where the query plan is generated.

The parse step can be deferred by the client software to reduce network traffic. What this means is that the `PARSE` is bundled with the `EXECUTE`, so there are fewer round-trips to the server.

Note: When checking if statements are identical, they must be identical in every way including case and spacing.



Steps 3 and 4: Describe and Define

Step 3: Describe

The describe stage is necessary only if the characteristics of a query's result are not known, for example, when a query is entered interactively by a user. In this case, the describe stage determines the characteristics (data types, lengths, and names) of a query's result. Describe tells the application what select list items are required. If, for example, you enter a query such as:

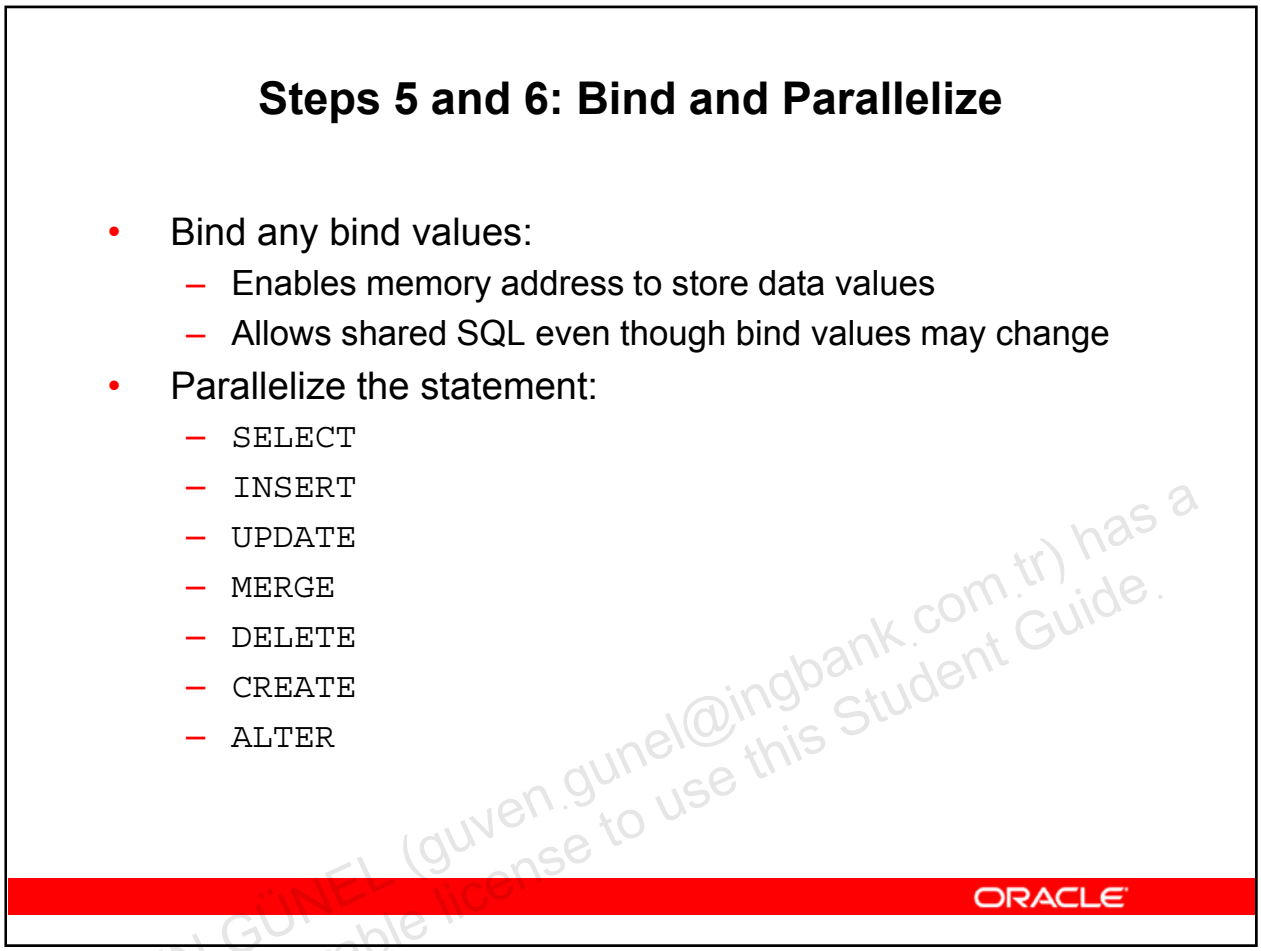
```
SQL> select * from employees;,
```

information about the columns in the employees table is required.

Step 4: Define

In the define stage, you specify the location, size, and data type of variables defined to receive each fetched value. These variables are called define variables. Oracle Database performs data type conversion, if necessary.

These two steps are generally hidden from users in tools such as SQL*Plus. However, with `DBMS_SQL` or OCI, it is necessary to tell the client what the output data is and which the setup areas are.



Steps 5 and 6: Bind and Parallelize

Step 5: Bind

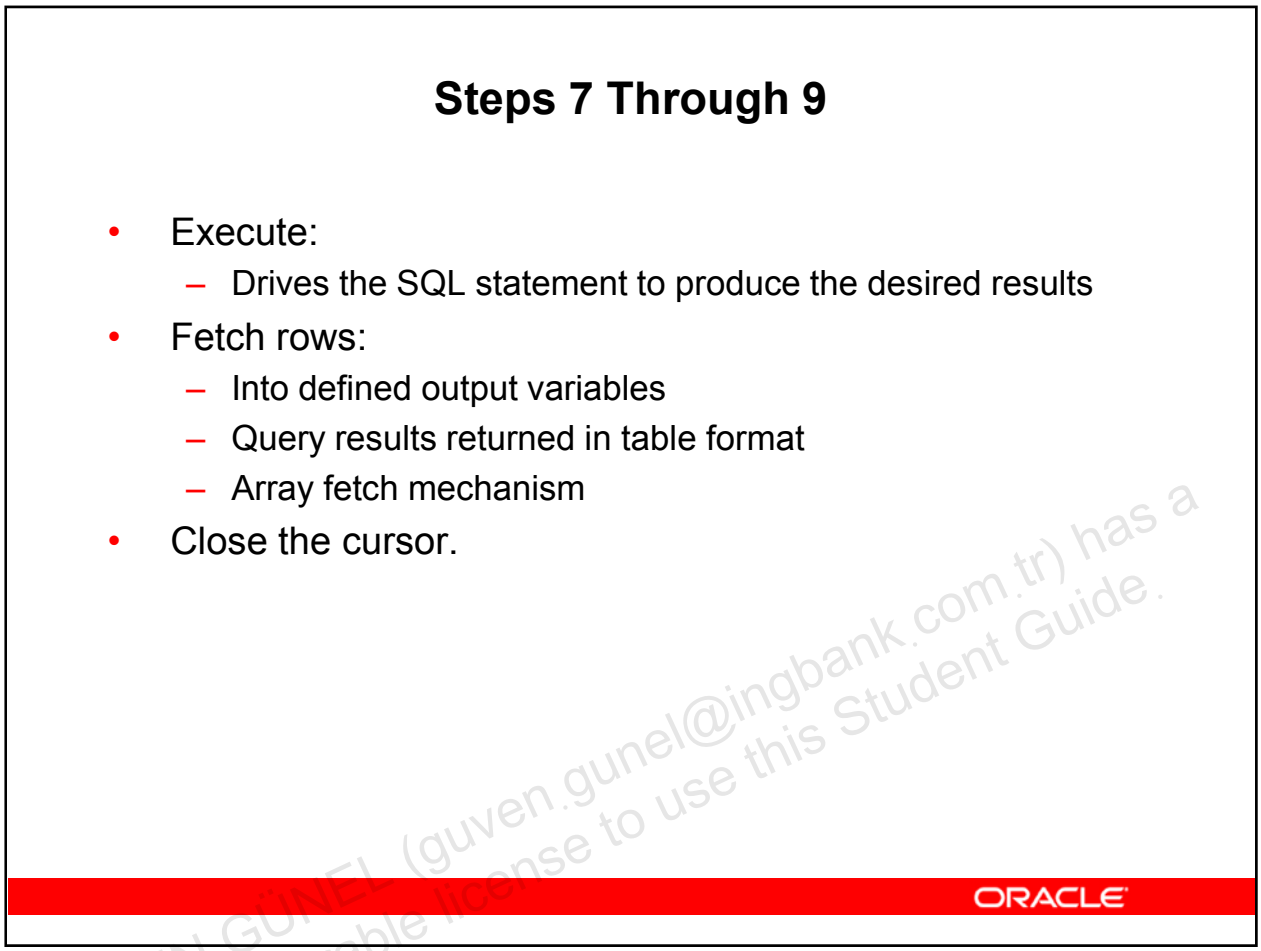
At this point, Oracle Database knows the meaning of the SQL statement, but still does not have enough information to run the statement. Oracle Database needs values for any variables listed in the statement. The process of obtaining these values is called binding variables.

Step 6: Parallelize

Oracle Database can parallelize the execution of SQL statements (such as `SELECT`, `INSERT`, `UPDATE`, `MERGE`, `DELETE`), and some DDL operations, such as index creation, creating a table with a subquery, and operations on partitions. Parallelization causes multiple server processes to perform the work of the SQL statement, so it can complete faster.

Parallelization involves dividing the work of a statement among a number of slave processes.

Parsing has already identified if a statement can be parallelized or not and has built the appropriate parallel plan. At execution time, this plan is then implemented if sufficient resource is available.



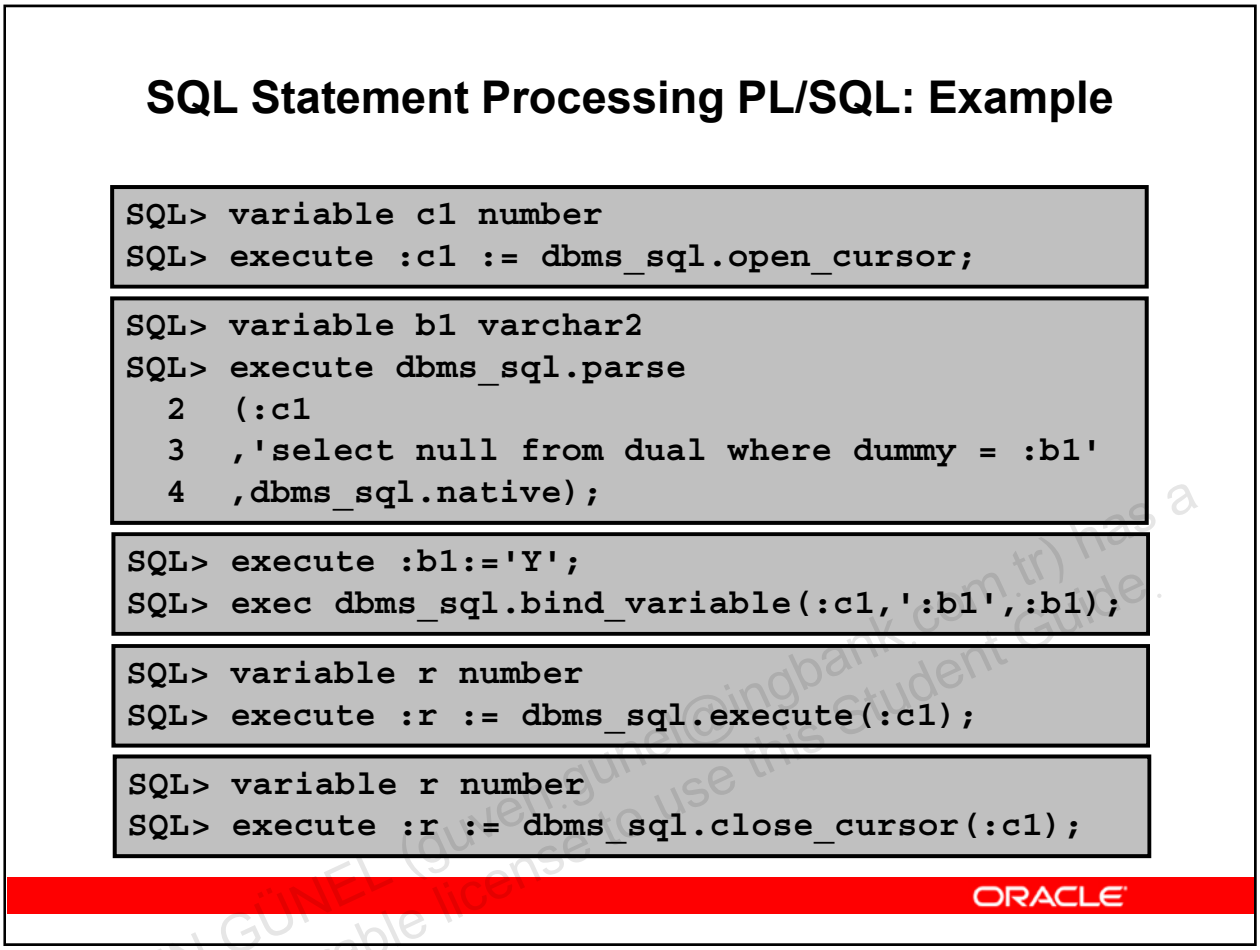
Steps 7 Through 9

At this point, Oracle Database has all the necessary information and resources, so the statement is run. If the statement is a query (without the `FOR UPDATE` clause) statement, no rows need to be locked because no data is changed. If the statement is an `UPDATE` or a `DELETE` statement, however, all rows that the statement affects are locked until the next `COMMIT`, `ROLLBACK`, or `SAVEPOINT` for the transaction. This ensures data integrity.

For some statements, you can specify a number of executions to be performed. This is called array processing. Given *n* number of executions, the bind and define locations are assumed to be the beginning of an array of size *n*.

In the fetch stage, rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result until the last row has been fetched.

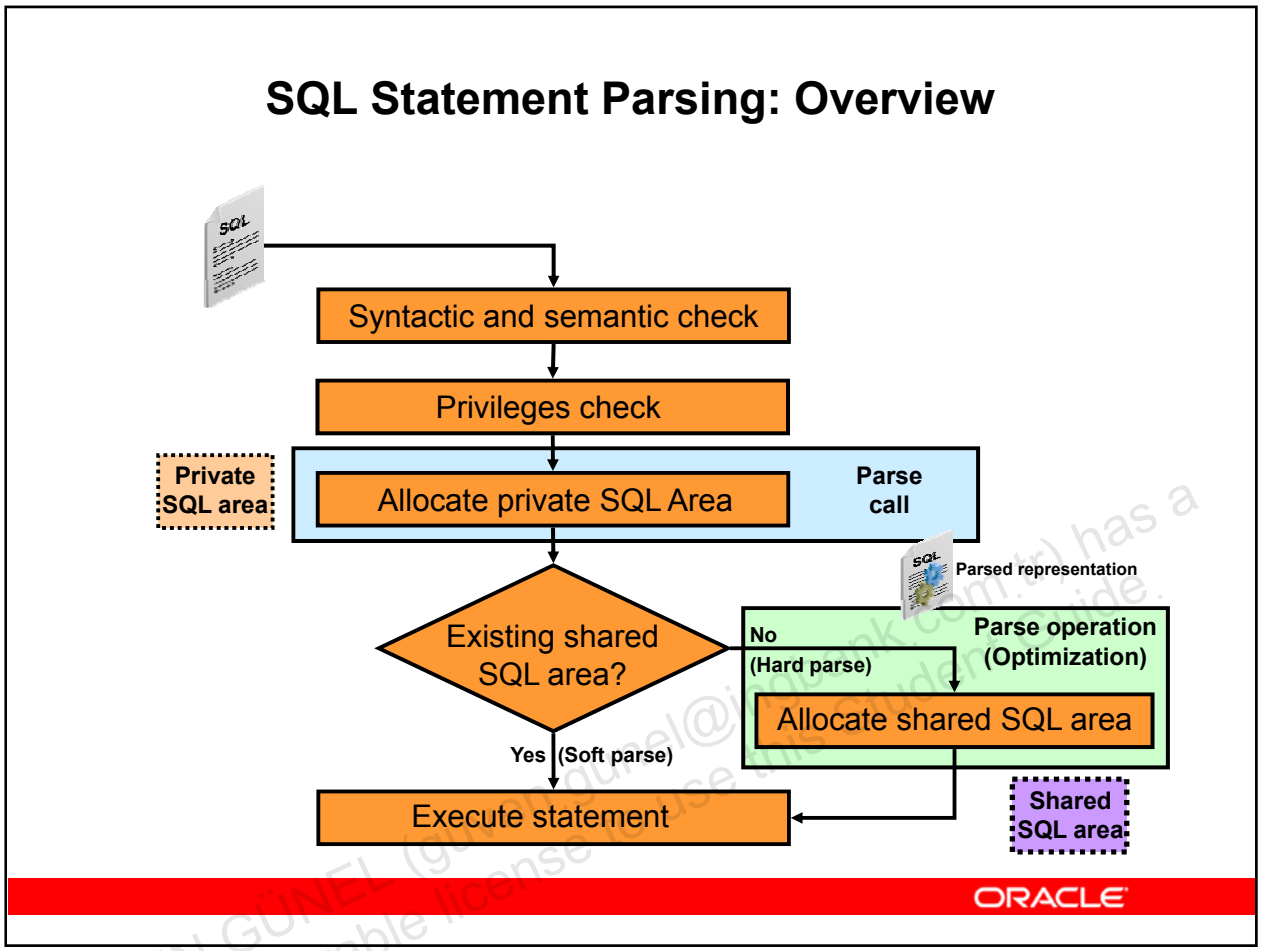
The final stage of processing a SQL statement is closing the cursor.



SQL Statement Processing PL/SQL: Example

This example summarizes the various steps discussed previously.

Note: In this example, you do not show the fetch operation. It is also possible to combine both the `EXECUTE` and `FETCH` operations in `EXECUTE_AND_FETCH` to perform `EXECUTE` and `FETCH` together in one call. This may reduce the number of network round-trips when used against a remote database.



SQL Statement Parsing: Overview

Parsing is one stage in the processing of a SQL statement. When an application issues a SQL statement, the application makes a parse call to Oracle Database. During the parse call, Oracle Database performs the following actions:

- Checks the statement for syntactic and semantic validity
- Determines whether the process issuing the statement has the privileges to run it
- Allocates a private SQL area for the statement
- Determines whether or not there is an existing shared SQL area containing the parsed representation of the statement in the library cache. If so, the user process uses this parsed representation and runs the statement immediately. If not, Oracle Database generates the parsed representation of the statement, and the user process allocates a shared SQL area for the statement in the library cache and stores its parsed representation there.

Note the difference between an application making a parse call for a SQL statement and Oracle Database actually parsing the statement.

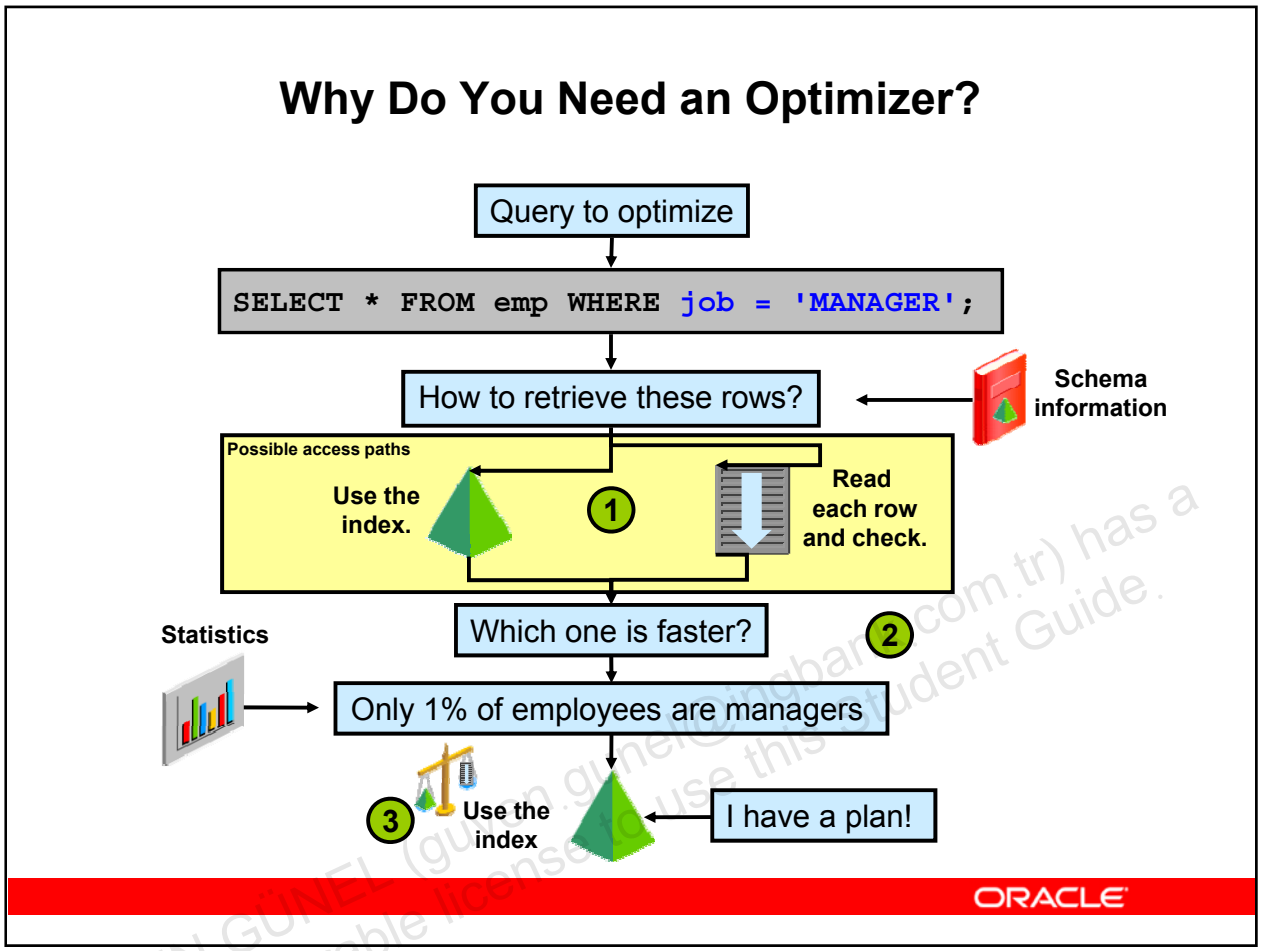
- A parse call by the application associates a SQL statement with a private SQL area. After a statement has been associated with a private SQL area, it can be run repeatedly without your application making a parse call.

- A parse operation by Oracle Database allocates a shared SQL area for a SQL statement. After a shared SQL area has been allocated for a statement, it can be run repeatedly without being reparsed.

Both parse calls and parsing can be expensive relative to execution, so perform them as rarely as possible.

Note: Although parsing a SQL statement validates that statement, parsing only identifies errors that can be found before statement execution. Thus, some errors cannot be caught by parsing. For example, errors in data conversion or errors in data (such as an attempt to enter duplicate values in a primary key) and deadlocks are all errors or situations that can be encountered and reported only during the execution stage.

Why Do You Need an Optimizer?



Why Do You Need an Optimizer?

The optimizer should always return the correct result as quickly as possible.

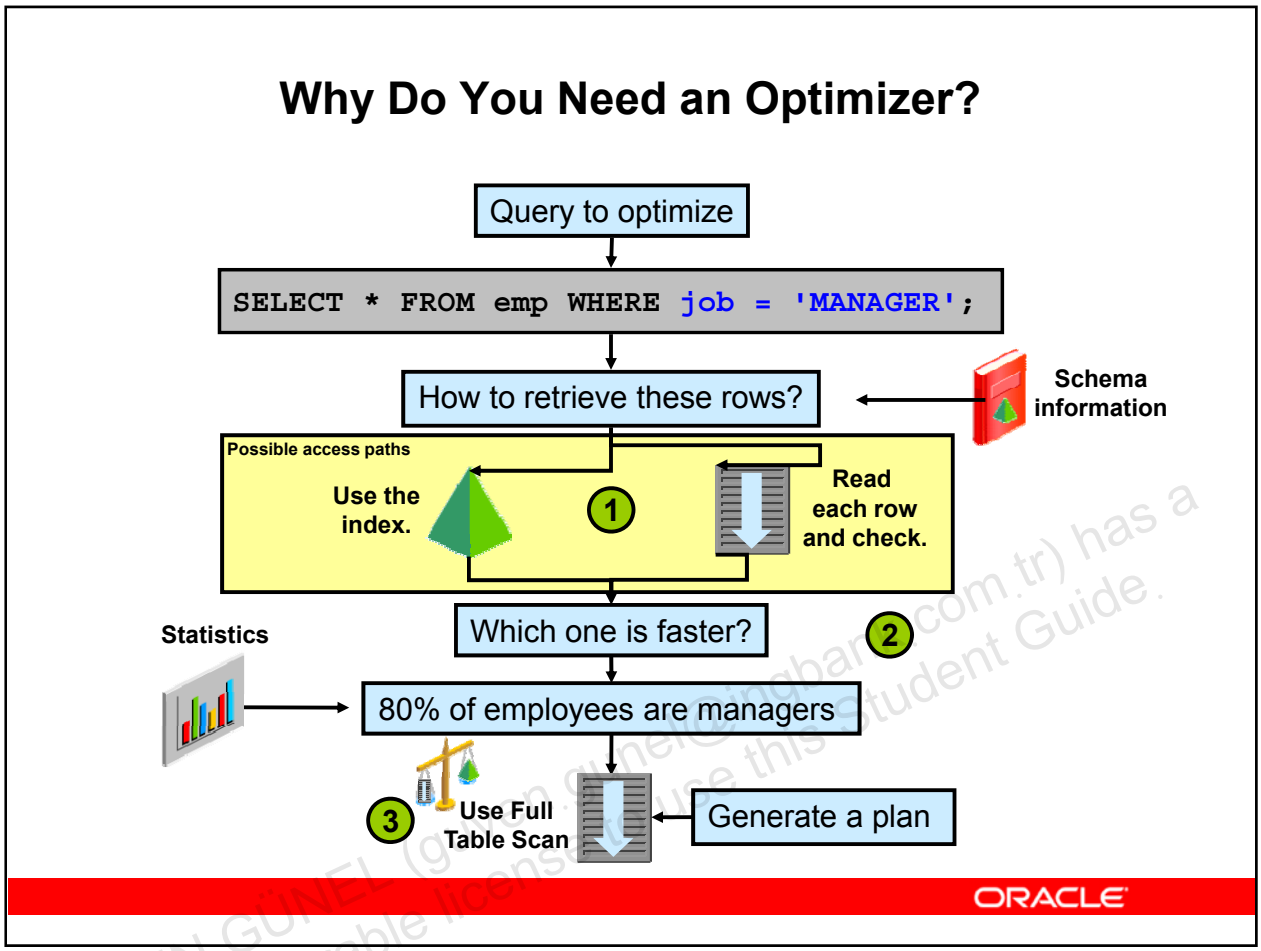
The query optimizer tries to determine which execution plan is most efficient by considering available access paths and by factoring in information based on statistics for the schema objects (tables or indexes) accessed by the SQL statement.

The query optimizer performs the following steps:

1. The optimizer generates a set of potential plans for the SQL statement based on available access paths.
2. The optimizer estimates the cost of each plan based on statistics in the data dictionary for the data distribution and storage characteristics of the tables, and indexes accessed by the statement.
3. The optimizer compares the costs of the plans and selects the one with the lowest cost.

Note: Because of the complexity of finding the best possible execution plan for a particular query, the optimizer's goal is to find a "good" plan that is generally called the best cost plan.

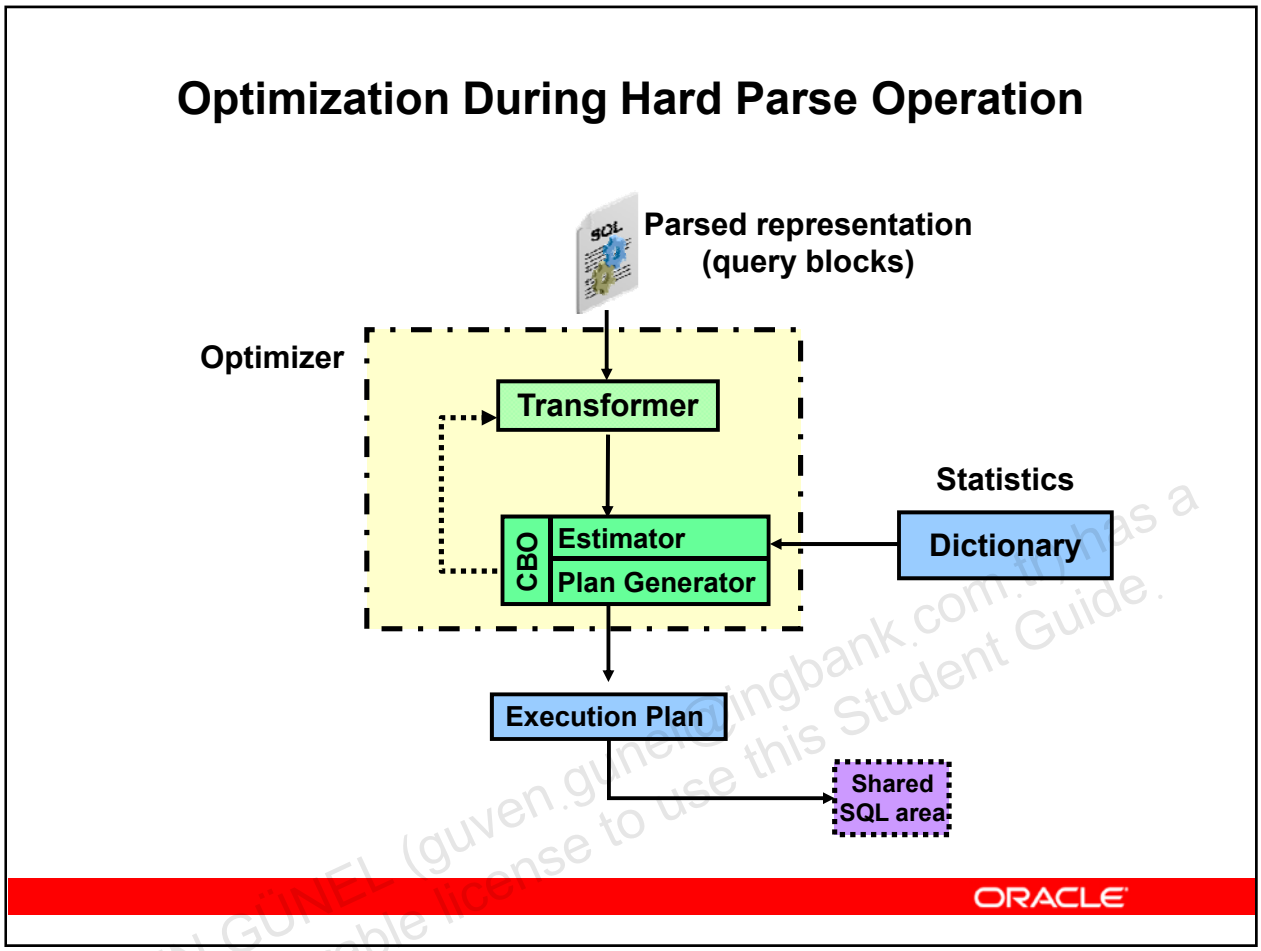
Why Do You Need an Optimizer?



Why Do You Need an Optimizer? (continued)

The example in the slide shows you that if statistics change, the optimizer adapts its execution plan. In this case, statistics show that 80 percent of the employees are managers. In the hypothetical case, a full table scan is probably a better solution than using the index.

Optimization During Hard Parse Operation



Optimization During Hard Parse Operation

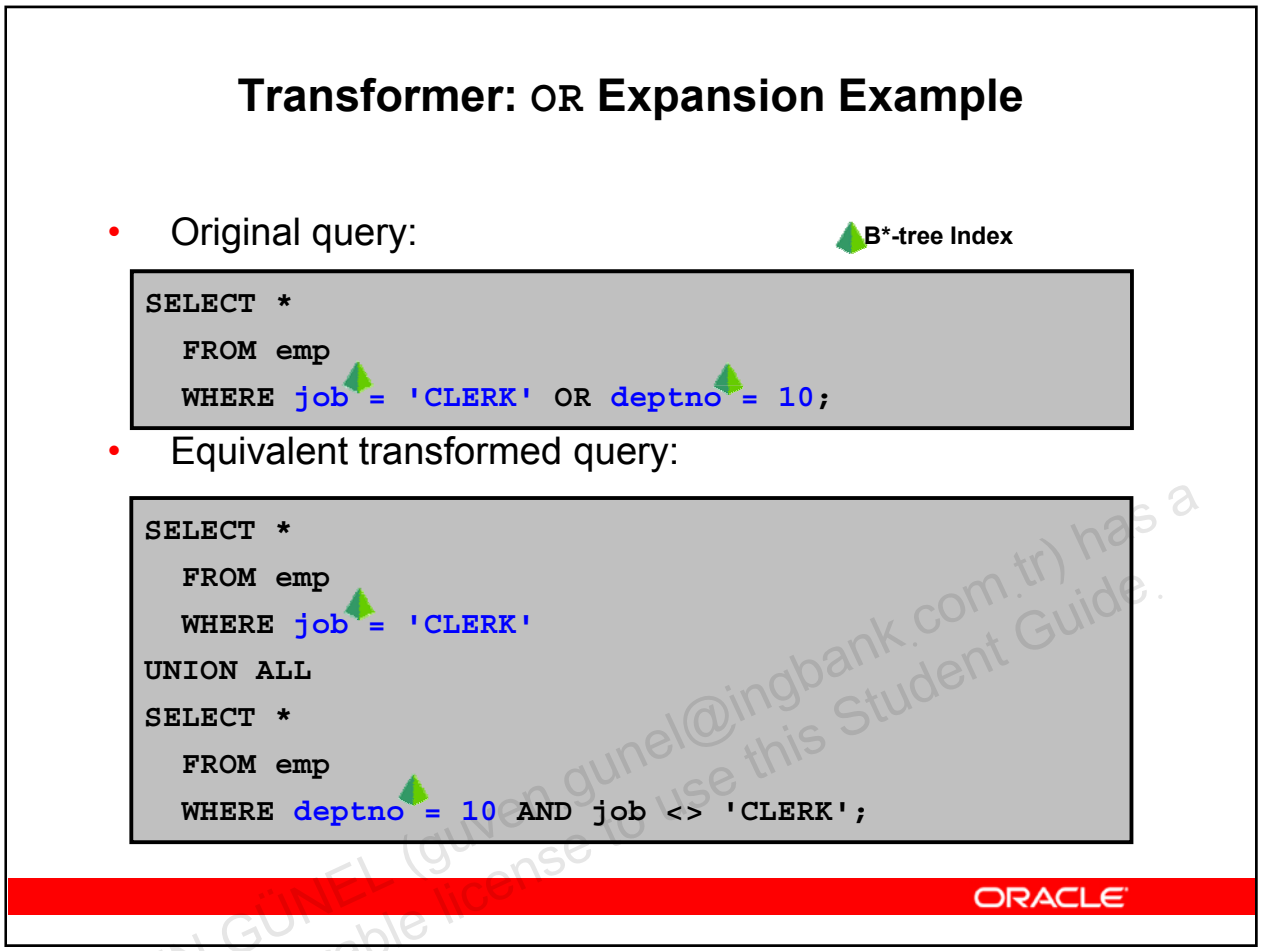
The optimizer creates the execution plan for a SQL statement.

SQL queries submitted to the system first run through the parser, which checks syntax and analyzes semantics. The result of this phase is called a parsed representation of the statement, and is constituted by a set of query blocks. A query block is a self-contained DML against a table. A query block can be a top-level DML or a subquery. This parsed representation is then sent to the optimizer, which handles three main functionalities: Transformation, estimation, and execution plan generation.

Before performing any cost calculation, the system may transform your statement into an equivalent
statement and calculate the cost of the equivalent statement. Depending on the version of Oracle Database, there are transformations that cannot be done, some that are always done, and some that are done, costed, and discarded.

The input to the query transformer is a parsed query, which is represented by a set of interrelated query blocks. The main objective of the query transformer is to determine if it is advantageous to change the structure of the query so that it enables generation of a better query plan. Several query transformation techniques are employed by the query transformer, such as transitivity, view merging, predicate pushing, subquery unnesting, query rewrite, star transformation, and `OR` expansion.

Transformer: OR Expansion Example



Transformer: OR Expansion Example

If a query contains a WHERE clause with multiple conditions combined with OR operators, the optimizer transforms it into an equivalent compound query that uses the UNION ALL set operator, if this makes the query execute more efficiently.

For example, if each condition individually makes an index access path available, the optimizer can make the transformation. The optimizer selects an execution plan for the resulting statement that accesses the table multiple times using the different indexes and then puts the results together. This transformation is done if the cost estimation is better than the cost of the original statement.

In the example in the slide, it is assumed that there are indexes on both the JOB and DEPTNO columns. Then, the optimizer might transform the original query into the equivalent transformed query shown in the slide. When the cost-based optimizer (CBO) decides whether to make a transformation, the optimizer compares the cost of executing the original query using a full table scan with that of executing the resulting query.

Transformer: Subquery Unnesting Example

- Original query:

```
SELECT *
  FROM accounts
  WHERE custno IN
       (SELECT custno FROM customers);
```

- Equivalent transformed query:

```
SELECT accounts.*
  FROM accounts, customers
  WHERE accounts.custno = customers.custno;
```

Primary or unique key

ORACLE

Transformer: Subquery Unnesting Example

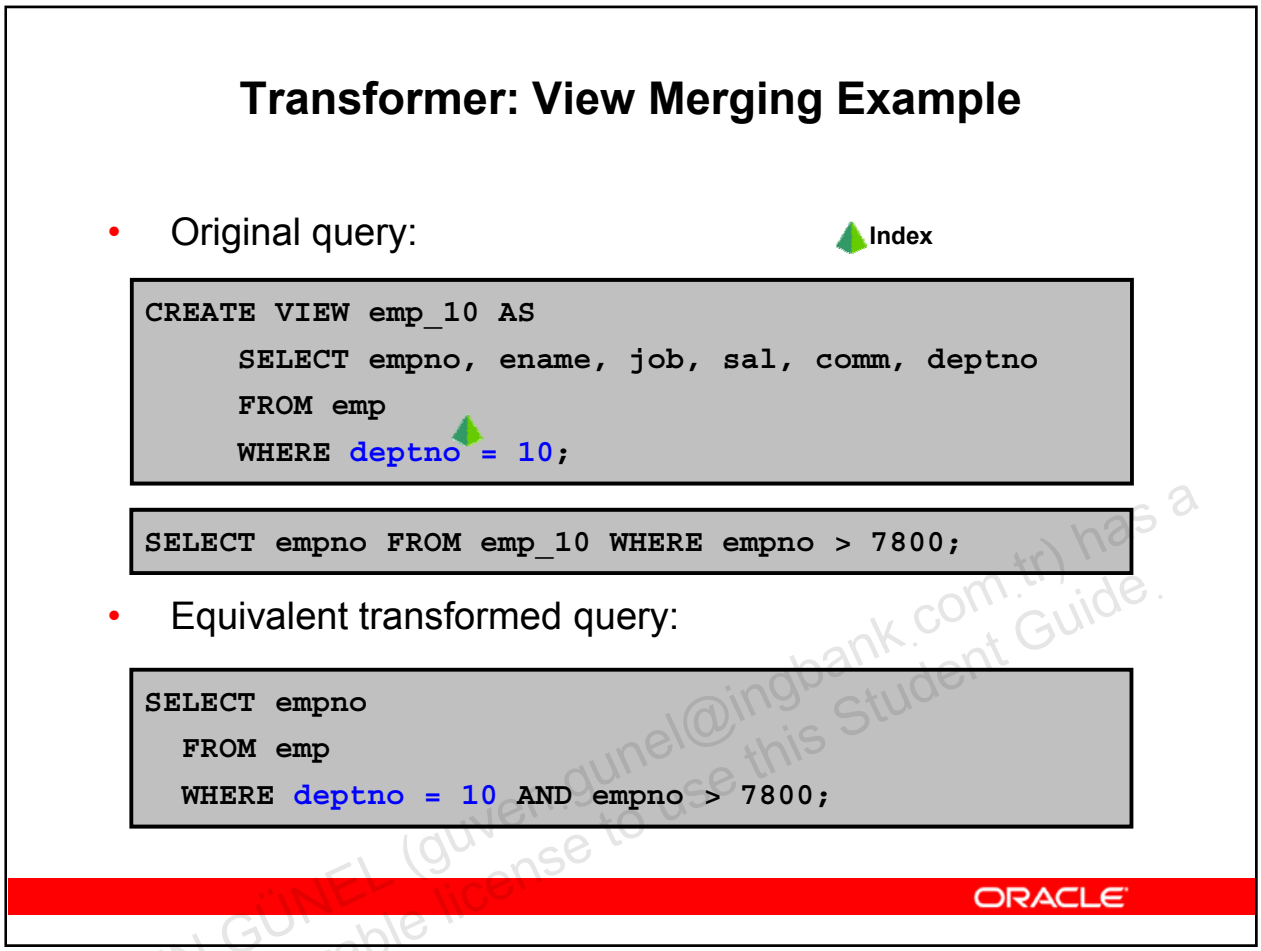
To unnest a query, the optimizer may choose to transform the original query into an equivalent `JOIN` statement, and then optimize the `JOIN` statement.

The optimizer may do this transformation only if the resulting `JOIN` statement is guaranteed to return exactly the same rows as the original statement. This transformation allows the optimizer to take advantage of the join optimizer techniques.

In the example in the slide, if the `CUSTNO` column of the customers table is a primary key or has a `UNIQUE` constraint, the optimizer can transform the complex query into the shown `JOIN` statement that is guaranteed to return the same data.

If the optimizer cannot transform a complex statement into a `JOIN` statement, it selects execution plans for the parent statement and the subquery as though they were separate statements. The optimizer then executes the subquery and uses the rows returned to execute the parent query.

Note: Complex queries whose subqueries contain aggregate functions such as `AVG` cannot be transformed into `JOIN` statements.



Transformer: View Merging Example

To merge the view's query into a referencing query block in the accessing statement, the optimizer replaces the name of the view with the names of its base tables in the query block and adds the condition of the view's query's `WHERE` clause to the accessing query block's `WHERE` clause.

This optimization applies to select-project-join views, which contain only selections, projections, and joins. That is, views that do not contain set operators, aggregate functions, `DISTINCT`, `GROUP BY`, `CONNECT BY`, and so on.

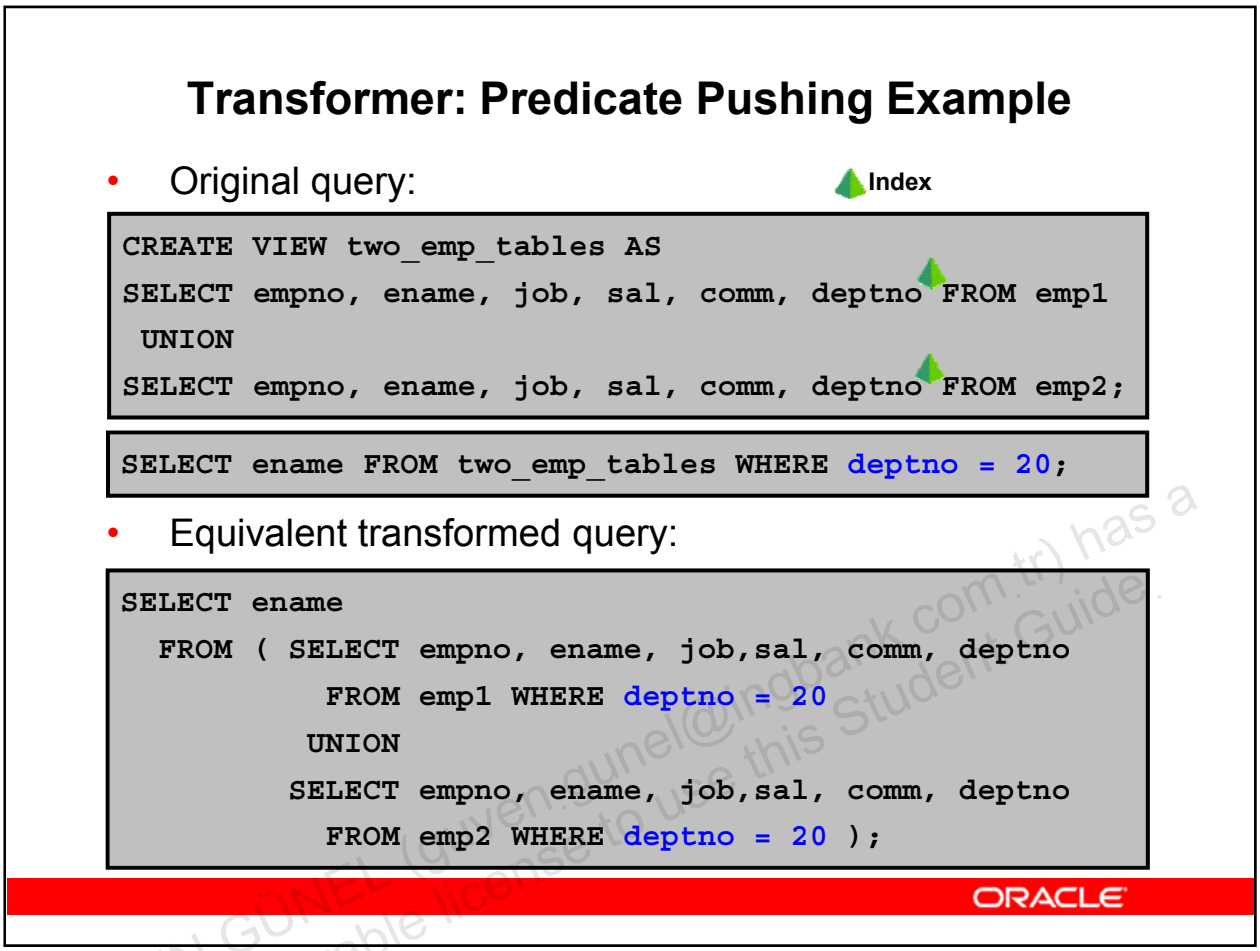
The view in this example is of all employees who work in department 10.

The query that follows the view's definition in the slide accesses the view. The query selects the IDs greater than 7800 of employees who work in department 10.

The optimizer may transform the query into the equivalent transformed query shown in the slide that accesses the view's base table.

If there are indexes on the `DEPTNO` or `EMPNO` columns, the resulting `WHERE` clause makes them available.

Transformer: Predicate Pushing Example



Transformer: Predicate Pushing Example

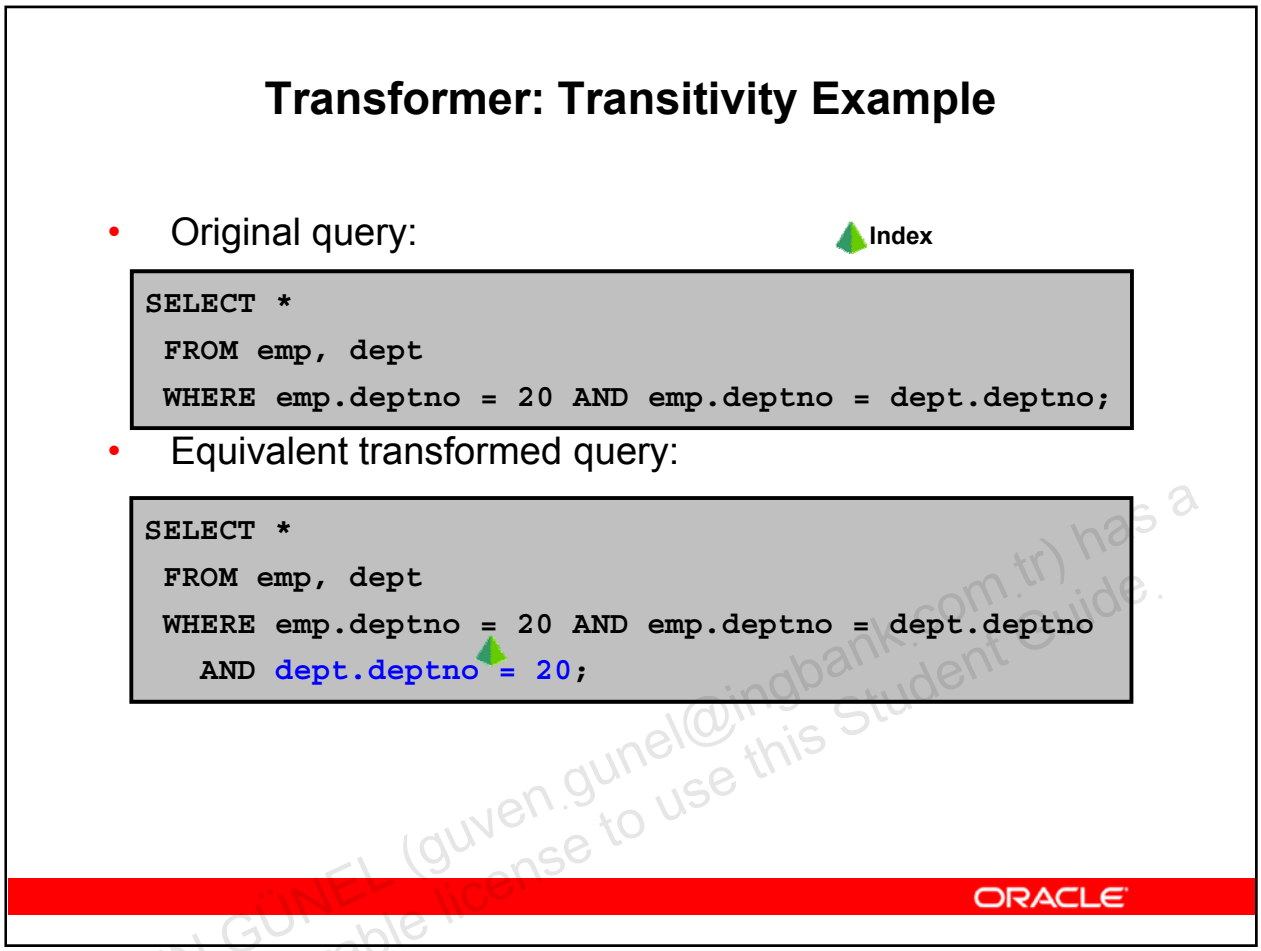
The optimizer can transform a query block that accesses a nonmergeable view by pushing the query block's predicates inside the view's query.

In the example in the slide, the `two_emp_tables` view is the union of two employee tables. The view is defined with a compound query that uses the `UNION` set operator.

The query that follows the view's definition in the slide accesses the view. The query selects the IDs and names of all employees in either table who work in department 20.

Because the view is defined as a compound query, the optimizer cannot merge the view's query into the accessing query block. Instead, the optimizer can transform the accessing statement by pushing its predicate, the `WHERE` clause condition `deptno = 20`, into the view's compound query. The equivalent transformed query is shown in the slide.

If there is an index in the `DEPTNO` column of both tables, the resulting `WHERE` clauses make them available.



Transformer: Transitivity Example

If two conditions in the `WHERE` clause involve a common column, the optimizer sometimes can infer a third condition, using the transitivity principle. The optimizer can then use the inferred condition to optimize the statement.

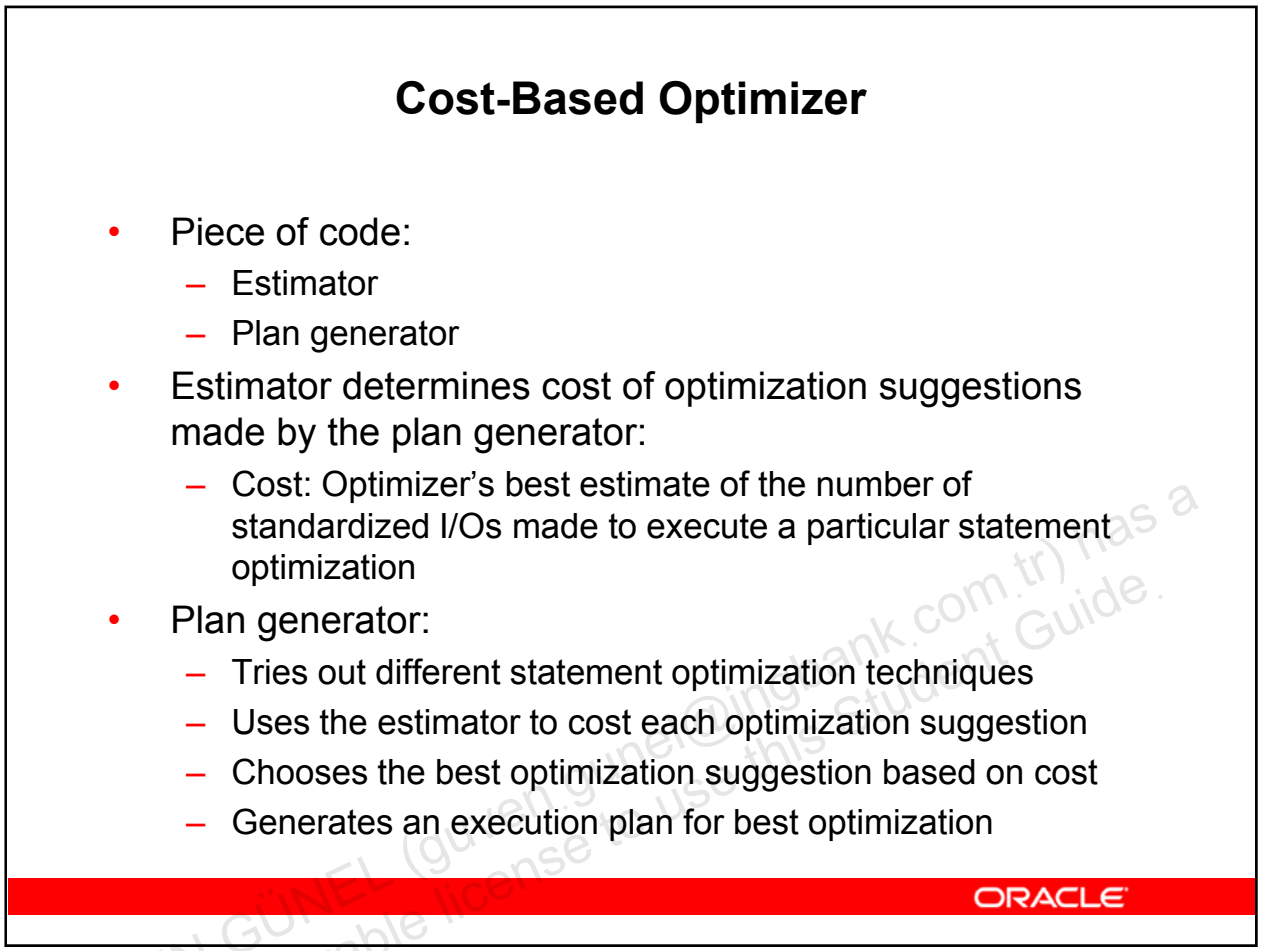
The inferred condition can make available an index access path that was not made available by the original conditions.

This is demonstrated with the example in the slide. The `WHERE` clause of the original query contains two conditions, each of which uses the `EMP.DEPTNO` column. Using transitivity, the optimizer infers the following condition: `dept.deptno = 20`

If an index exists in the `DEPT.DEPTNO` column, this condition makes access paths available using that index.

Note: The optimizer only infers conditions that relate columns to constant expressions, rather than columns to other columns.

Cost-Based Optimizer



Cost-Based Optimizer

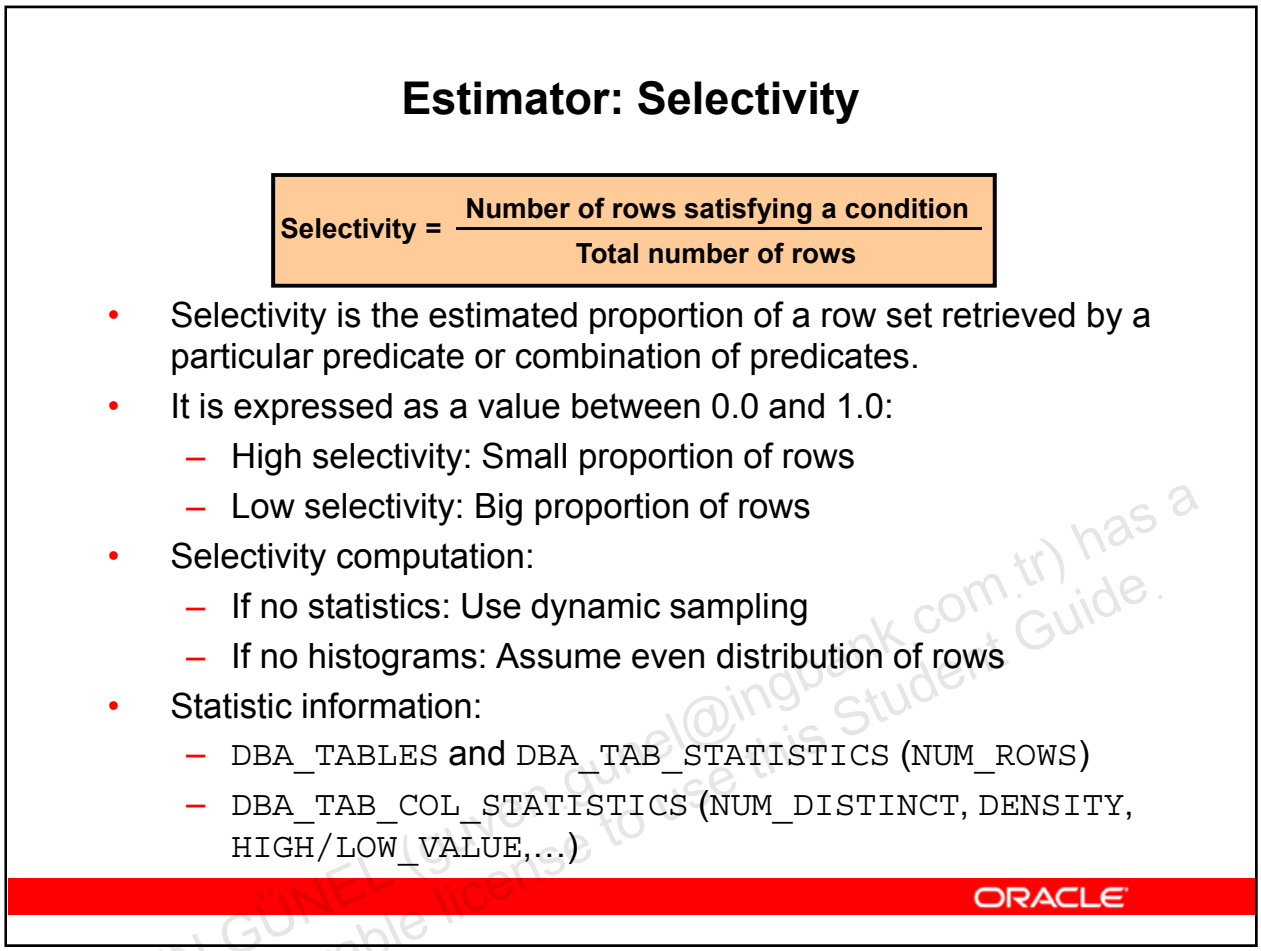
The combination of the estimator and plan generator code is commonly called the cost-based optimizer (CBO).

The estimator generates three types of measures: selectivity, cardinality, and cost. These measures are related to each other. Cardinality is derived from selectivity and often the cost depends on cardinality. The end goal of the estimator is to estimate the overall cost of a given plan. If statistics are available, the estimator uses these to improve the degree of accuracy when computing the measures.

The main function of the plan generator is to try out different possible plans for a given query and pick the one that has the lowest cost. Many different plans are possible because of the various combinations of different access paths, join methods, and join orders that can be used to access and process data in different ways and produce the same result. The number of possible plans for a query block is proportional to the number of join items in the `FROM` clause. This number rises exponentially with the number of join items.

The optimizer uses various pieces of information to determine the best path: `WHERE` clause, statistics, initialization parameters, supplied hints, and schema information.

Estimator: Selectivity

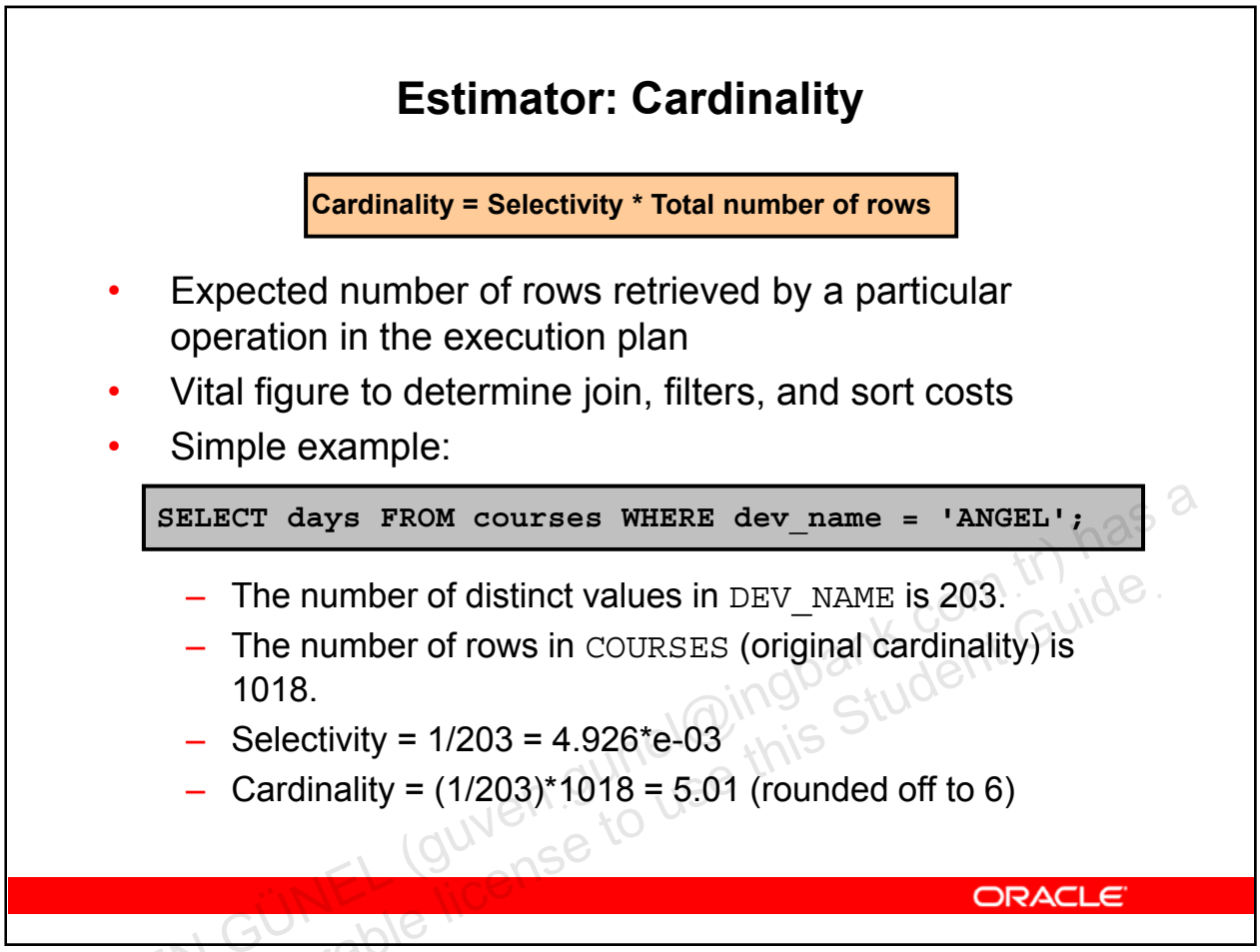


Estimator: Selectivity

Selectivity represents a fraction of rows from a row set. The row set can be a base table, a view, or the result of a join or a `GROUP BY` operator. The selectivity is tied to a query predicate, such as `last_name = 'Smith'`, or a combination of predicates, such as `last_name = 'Smith' AND job_type = 'Clerk'`. A predicate acts as a filter that filters a certain number of rows from a row set. Therefore, the selectivity of a predicate indicates the percentage of rows from a row set that passes the predicate test. Selectivity lies in a value range from 0.0 to 1.0. A selectivity of 0.0 means that no rows are selected from a row set, and a selectivity of 1.0 means that all rows are selected.

If no statistics are available, the optimizer either uses dynamic sampling or an internal default value, depending on the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter. When statistics are available, the estimator uses them to estimate selectivity. For example, for an equality predicate (`last_name = 'Smith'`), selectivity is set to the reciprocal of the number *n* of distinct values of `LAST_NAME` because the query selects rows that contain one out of *n* distinct values. Thus, even distribution is assumed. If a histogram is available in the `LAST_NAME` column, the estimator uses it instead of the number of distinct values. The histogram captures the distribution of different values in a column, so it yields better selectivity estimates.

Note: It is important to have histograms in columns that contain values with large variations in the number of duplicates (data skew).



Estimator: Cardinality

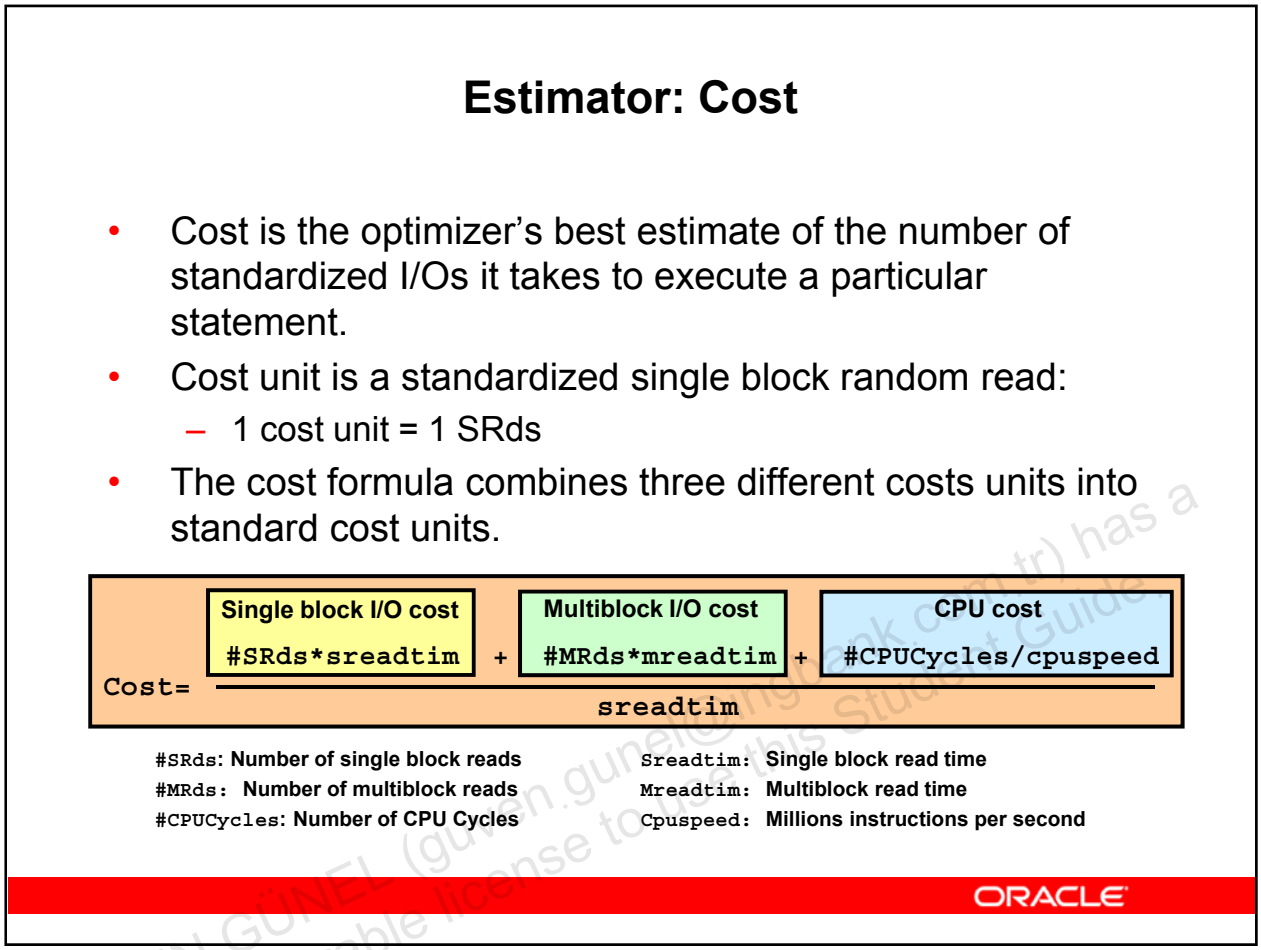
The cardinality of a particular operation in the execution plan of a query represents the estimated number of rows retrieved by that particular operation. Most of the time, the row source can be a base table, a view, or the result of a join or `GROUP BY` operator.

When costing a join operation, it is important to know the cardinality of the driving row source. With nested loops join, for example, the driving row source defines how often the system probes the inner row source.

Because sort costs are dependent on the size and number of rows to be sorted, cardinality figures are also vital for sort costing.

In the example in the slide, based on assumed statistics, the optimizer knows that there are 203 different values in the `DEV_NAME` column, and that the total number of rows in the `COURSES` table is 1018. Based on this assumption, the optimizer deduces that the selectivity of the `DEV_NAME='ANGEL'` predicate is 1/203 (assuming there are no histograms), and also deduces the cardinality of the query to be (1/203)*1018. This number is then rounded off to the nearest integer, 6.

Estimator: Cost



Estimator: Cost

The cost of a statement represents the optimizer's best estimate of the number of standardized inputs/outputs (I/Os) it takes to execute that statement. Basically, the cost is a normalized value in terms of a number of single block random reads

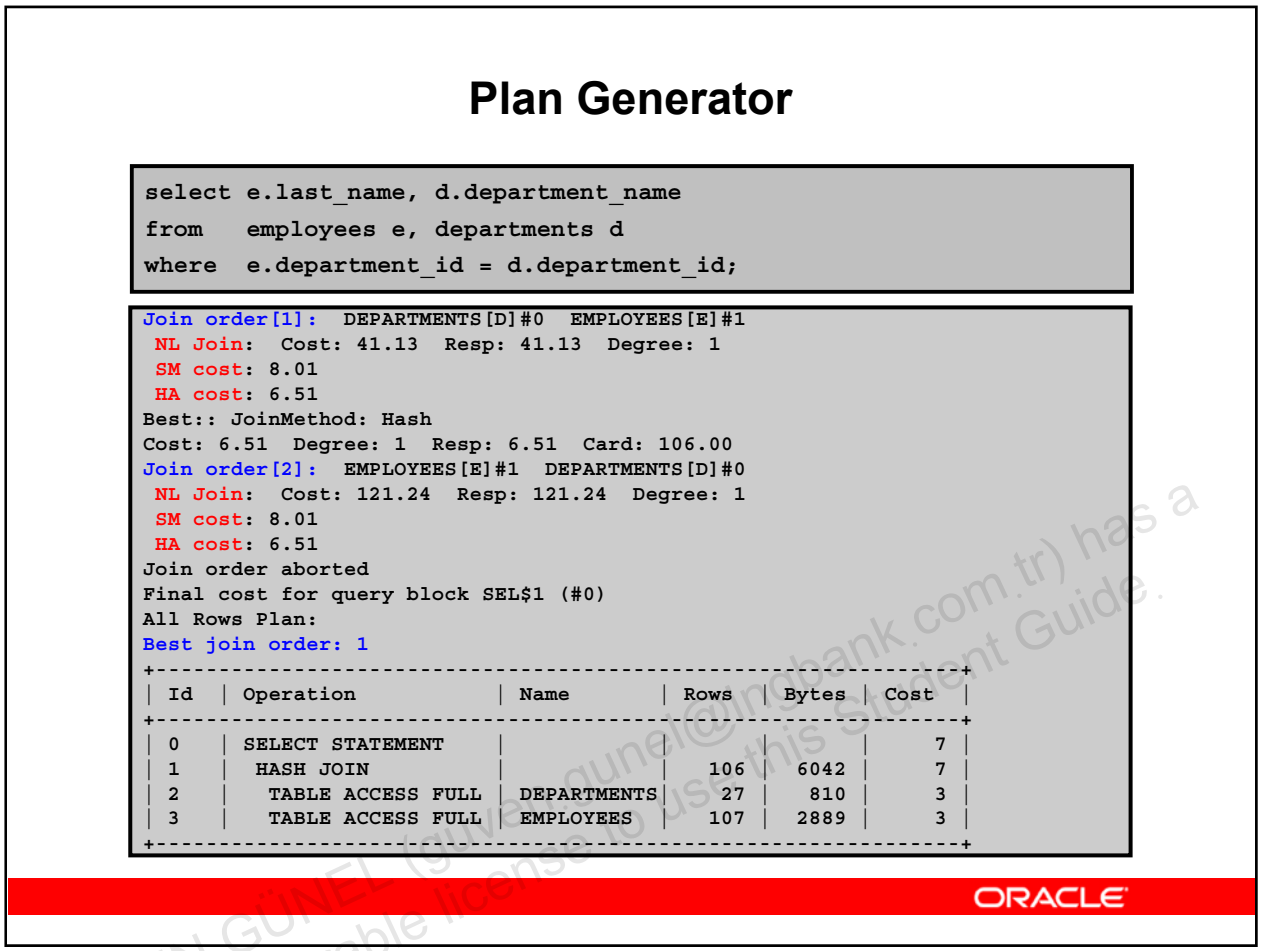
The standard cost metric measured by the optimizer is in terms of number of single block random reads, so one cost unit corresponds to one single block random read. The formula shown in the slide combines three different cost units:

- Estimated time to do all the single-block random reads
- Estimated time to do all the multiblock reads
- Estimated time for the CPU to process the statement into one standard cost unit

The model includes CPU costing because in most cases CPU utilization is as important as I/O; often it is the only contribution to the cost (in cases of in-memory sort, hash, predicate evaluation, and cached I/O).

This model is straightforward for serial execution. For parallel execution, necessary adjustments are made while computing estimates for `#SRds`, `#MRds`, and `#CPUCycles`.

Note: `#CPUCycles` includes CPU cost of query processing (pure CPU cost) and CPU cost of data retrieval (CPU cost of the buffer cache get).



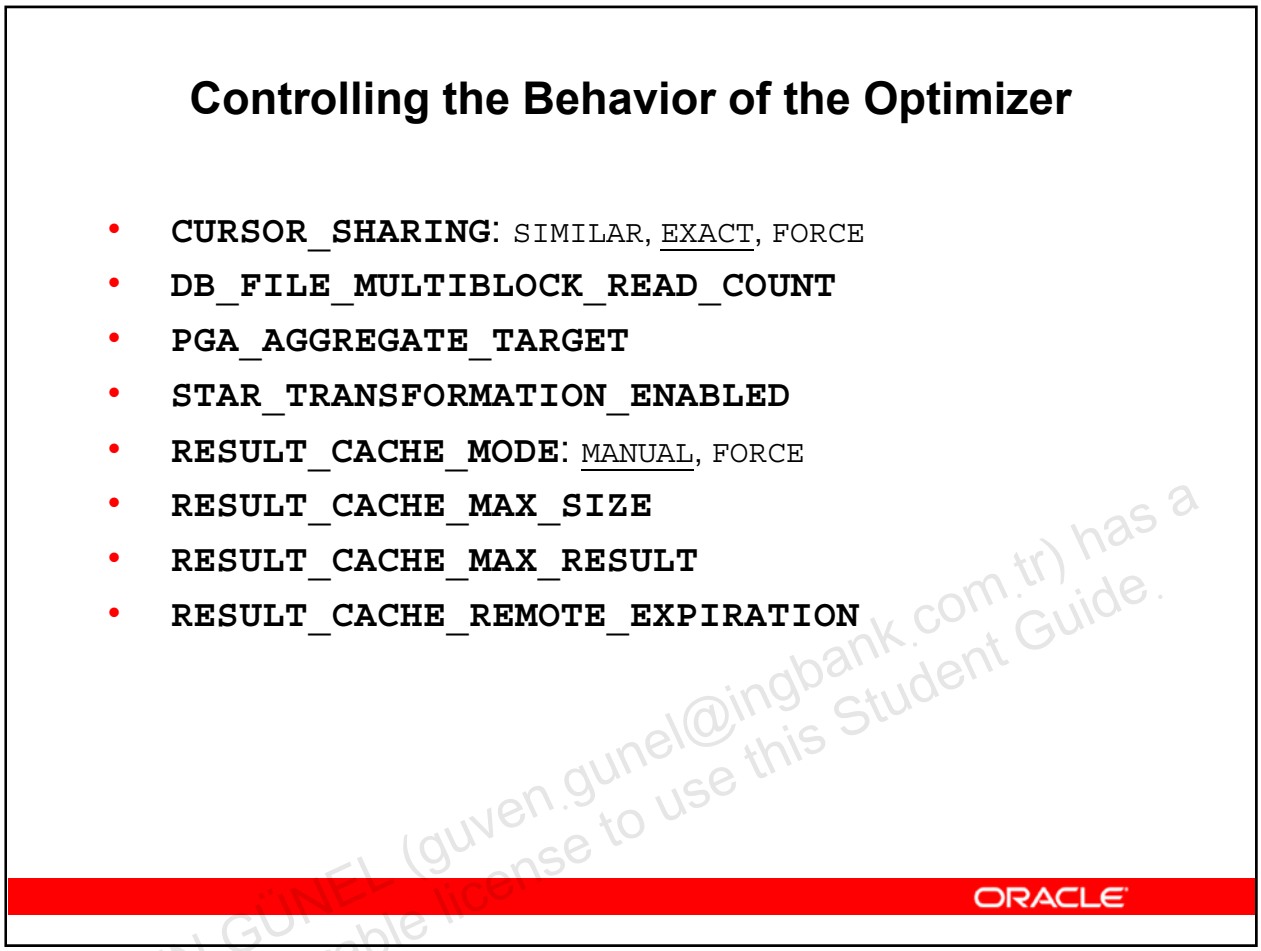
Plan Generator

The plan generator explores various plans for a query block by trying out different access paths, join methods, and join orders. Ultimately, the plan generator delivers the best execution plan for your statement. The slide shows you an extract of an optimizer trace file generated for the select statement. As you can see from the trace, the plan generator has six possibilities, or six different plans to test: Two join orders, and for each, three different join methods. It is assumed that there are no indexes in this example.

To retrieve the rows, you can start to join the `DEPARTMENTS` table to the `EMPLOYEES` table. For that particular join order, you can use three possible join mechanisms that the optimizer knows: Nested Loop, Sort Merge, or Hash Join. For each possibility, you have the cost of the corresponding plan. The best plan is the one shown at the end of the trace.

The plan generator uses an internal cutoff to reduce the number of plans it tries when finding the one with the lowest cost. The cutoff is based on the cost of the current best plan. If the current best cost is large, the plan generator tries harder (in other words, explores more alternate plans) to find a better plan with lower cost. If the current best cost is small, the plan generator ends the search swiftly because further cost improvement is not significant. The cutoff works well if the plan generator starts with an initial join order that produces a plan with a cost close to optimal. Finding a good initial join order is a difficult problem.

Note: Access path, join methods, and plan are discussed in more detail in the lessons titled “Optimizer Operators” and “Interpreting Execution Plans.”



Controlling the Behavior of the Optimizer

These parameters control the optimizer behavior:

- `CURSOR_SHARING` determines what kind of SQL statements can share the same cursors:
  - `FORCE`: Forces statements that may differ in some literals, but are otherwise identical, to share a cursor, unless the literals affect the meaning of the statement
  - `SIMILAR`: Causes statements that may differ in some literals, but are otherwise identical, to share a cursor, unless the literals affect either the meaning of the statement or the degree to which the plan is optimized. Forcing cursor sharing among similar (but not identical) statements can have unexpected results in some decision support system (DSS) applications, or applications that use stored outlines.
  - `EXACT`: Only allows statements with identical text to share the same cursor. This is the default.
- `DB_FILE_MULTIBLOCK_READ_COUNT` is one of the parameters you can use to minimize I/O during table scans or index fast full scan. It specifies the maximum number of blocks read in one I/O operation during a sequential scan. The total number of I/Os needed to perform a full table scan or an index fast full scan depends on factors, such as the size of the segment, the multiblock read count, and whether parallel execution is

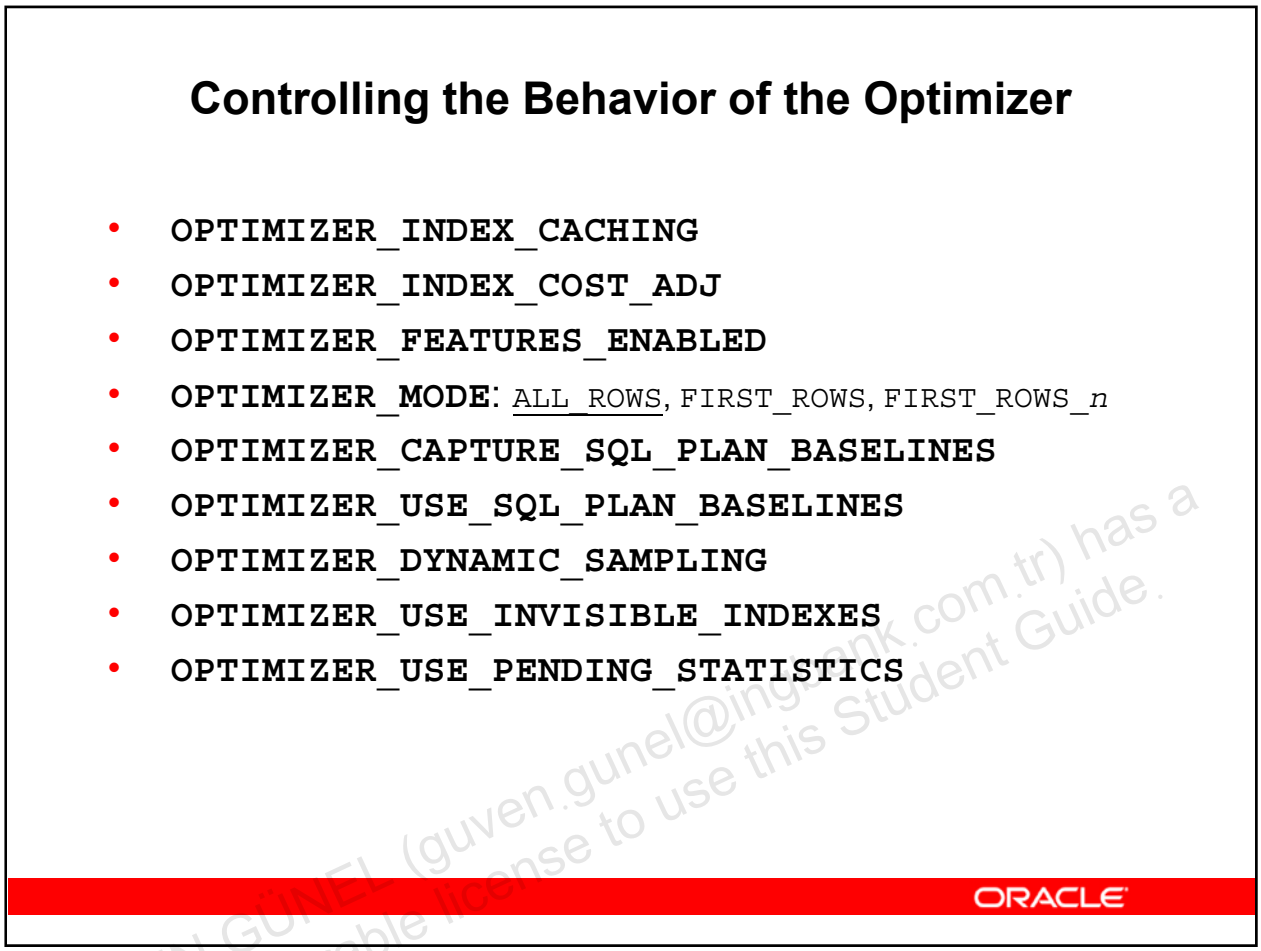
being utilized for the operation. As of Oracle Database 10*g*, Release 2, the default value of this parameter is a value that corresponds to the maximum I/O size that can be performed efficiently. This value is platform-dependent and is calculated at instance startup for most platforms.

Because the parameter is expressed in blocks, it automatically computes a value that is equal to the maximum I/O size that can be performed efficiently divided by the standard block size. Note that if the number of sessions is extremely large, the multiblock read count value is decreased to avoid the buffer cache getting flooded with too many table scan buffers. Even though the default value may be a large value, the optimizer does not favor large plans if you do not set this parameter. It would do so only if you explicitly set this parameter to a large value. Basically, if this parameter is not set explicitly (or is set is 0), the optimizer uses a default value of 8 when costing full table scans and index fast full scans. Online transaction processing (OLTP) and batch environments typically have values in the range of 4 to 16 for this parameter. DSS and data warehouse environments tend to benefit most from maximizing the value of this parameter. The optimizer is more likely to select a full table scan over an index, if the value of this parameter is high.

- `PGA_AGGREGATE_TARGET` specifies the target aggregate PGA memory available to all server processes attached to the instance. Setting `PGA_AGGREGATE_TARGET` to a nonzero value has the effect of automatically setting the `WORKAREA_SIZE_POLICY` parameter to `AUTO`. This means that SQL working areas used by memory-intensive SQL operators (such as sort, group-by, hash-join, bitmap merge, and bitmap create) are automatically sized. A nonzero value for this parameter is the default because, unless you specify otherwise, the system sets it to 20% of the SGA or 10 MB, whichever is greater. Setting `PGA_AGGREGATE_TARGET` to 0 automatically sets the `WORKAREA_SIZE_POLICY` parameter to `MANUAL`. This means that SQL work areas are sized using the `*_AREA_SIZE` parameters. The system attempts to keep the amount of private memory below the target specified by this parameter by adapting the size of the work areas to private memory. When increasing the value of this parameter, you indirectly increase the memory allotted to work areas. Consequently, more memory-intensive operations are able to run fully in memory and a less number of them work their way over to disk. When setting this parameter, you should examine the total memory on your system that is available to the Oracle instance and subtract the SGA. You can assign the remaining memory to `PGA_AGGREGATE_TARGET`.
- `STAR_TRANSFORMATION_ENABLED` determines whether a cost-based query transformation is applied to star queries. This optimization is explained in the lesson titled “Case Study: Star Transformation.”
- The query optimizer manages the result cache mechanism depending on the settings of the `RESULT_CACHE_MODE` parameter in the initialization parameter file. You can use this parameter to determine whether or not the optimizer automatically sends the results of queries to the result cache. The possible parameter values are `MANUAL`, and `FORCE`:
  - When set to `MANUAL` (the default), you must specify, by using the `RESULT_CACHE` hint, that a particular result is to be stored in the cache.
  - When set to `FORCE`, all results are stored in the cache. For the `FORCE` setting, if the statement contains a `[NO_]RESULT_CACHE` hint, the hint takes precedence over the parameter setting.
- The memory size allocated to the result cache depends on the memory size of the SGA as well as the memory management system. You can change the memory allocated to the result cache by setting the `RESULT_CACHE_MAX_SIZE` parameter. The result cache

is disabled if you set its value to 0. The value of this parameter is rounded to the largest multiple of 32 KB that is not greater than the specified value. If the rounded value is 0, the feature is disabled.

- Use the `RESULT_CACHE_MAX_RESULT` parameter to specify the maximum amount of cache memory that can be used by any single result. The default value is 5%, but you can specify any percentage value between 1 and 100.
- Use the `RESULT_CACHE_REMOTE_EXPIRATION` parameter to specify the time (in number of minutes) for which a result that depends on remote database objects remains valid. The default value is 0, which implies that results using remote objects should not be cached. Setting this parameter to a nonzero value can produce stale answers, for example, if the remote table used by a result is modified at the remote database.



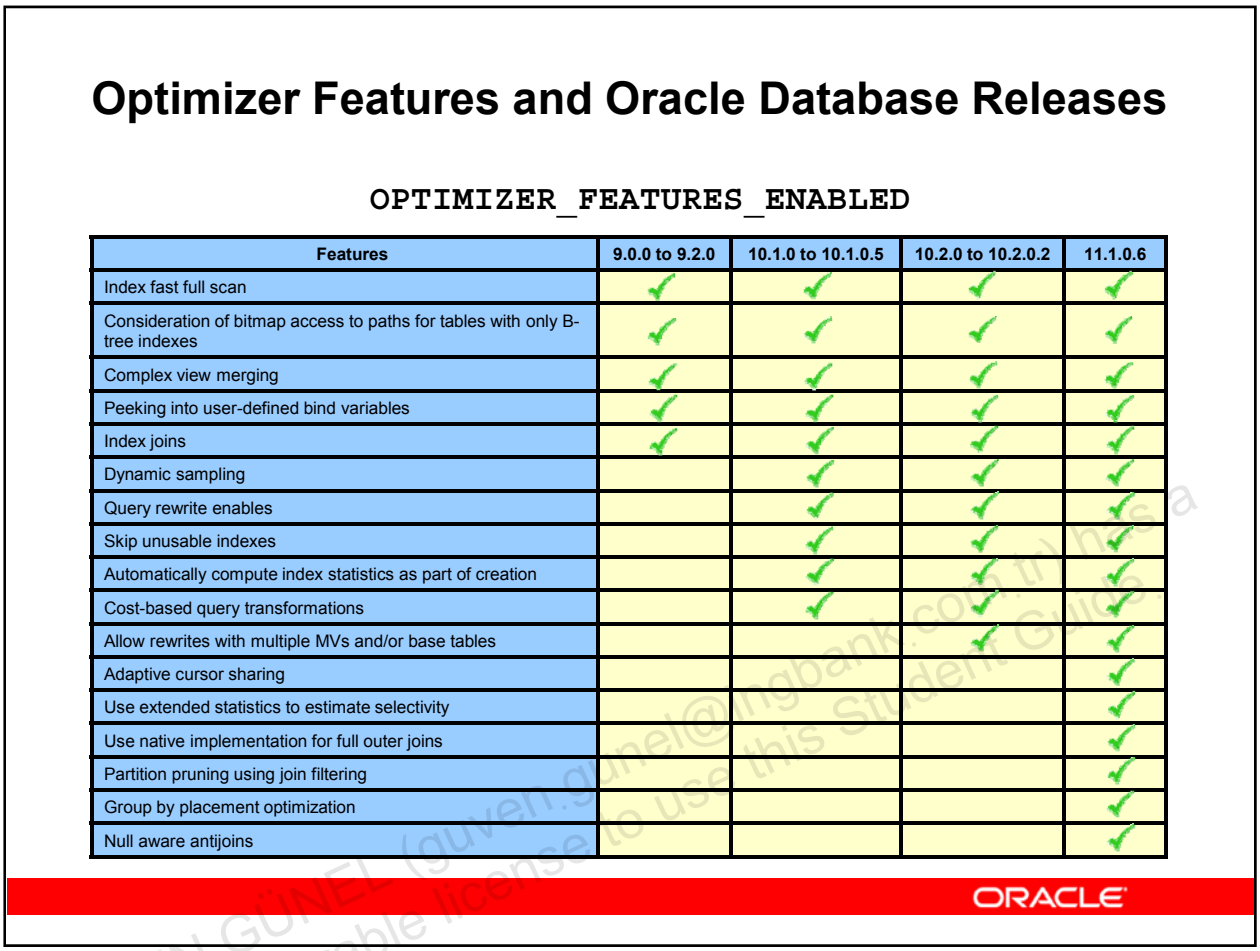
Controlling the Behavior of the Optimizer (continued)

- `OPTIMIZER_INDEX_CACHING`: This parameter controls the costing of an index probe in conjunction with a nested loop or an inlist iterator. The range of values 0 to 100 for `OPTIMIZER_INDEX_CACHING` indicates percentage of index blocks in the buffer cache, which modifies the optimizer's assumptions about index caching for nested loops and inlist iterators. A value of 100 infers that 100% of the index blocks are likely to be found in the buffer cache and the optimizer adjusts the cost of an index probe or nested loop accordingly. The default for this parameter is 0, which results in default optimizer behavior. Use caution when using this parameter because execution plans can change in favor of index caching.
- `OPTIMIZER_INDEX_COST_ADJ` lets you tune optimizer behavior for access path selection to be more or less index friendly, that is, to make the optimizer more or less prone to selecting an index access path over a full table scan. The range of values is 1 to 10000. The default for this parameter is 100 percent, at which the optimizer evaluates index access paths at the regular cost. Any other value makes the optimizer evaluate the access path at that percentage of the regular cost. For example, a setting of 50 makes the index access path look half as expensive as normal.
- `OPTIMIZER_FEATURES_ENABLED` acts as an umbrella parameter for enabling a series of optimizer features based on an Oracle release number.

For example, if you upgrade your database from release 10.1 to release 11.1, but you want to keep the release 10.1 optimizer behavior, you can do so by setting this parameter to 10.1.0. At a later time, you can try the enhancements introduced in releases up to and including release 11.1 by setting the parameter to 11.1.0.6. However, it is not recommended to explicitly set the OPTIMIZER_FEATURES_ENABLE parameter to an earlier release. To avoid possible SQL performance regression that may result from execution plan changes, consider using SQL plan management instead.

- OPTIMIZER_MODE establishes the default behavior for selecting an optimization approach for either the instance or your session. The possible values are:
  - ALL_ROWS: The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best throughput (minimum resource use to complete the entire statement). This is the default value.
  - FIRST_ROWS_n: The optimizer uses a cost-based approach, regardless of the presence of statistics, and optimizes with a goal of best response time to return the first *n* number of rows; *n* can equal 1, 10, 100, or 1000.
  - FIRST_ROWS: The optimizer uses a mix of cost and heuristics to find a best plan for fast delivery of the first few rows. Using heuristics sometimes leads the query optimizer to generate a plan with a cost that is significantly larger than the cost of a plan without applying the heuristic. FIRST_ROWS is available for backward compatibility and plan stability; use FIRST_ROWS_n instead.
- OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES enables or disables the automatic recognition of repeatable SQL statements, as well as the generation of SQL plan baselines for such statements.
- OPTIMIZER_USE_SQL_PLAN_BASELINES enables or disables the use of SQL plan baselines stored in SQL Management Base. When enabled, the optimizer looks for a SQL plan baseline for the SQL statement being compiled. If one is found in SQL Management Base, the optimizer costs each of the baseline plans and picks one with the lowest cost.
- OPTIMIZER_DYNAMIC_SAMPLING controls the level of dynamic sampling performed by the optimizer. If OPTIMIZER_FEATURES_ENABLE is set to:
  - 10.0.0 or later, the default value is 2
  - 9.2.0, the default value is 1
  - 9.0.1 or earlier, the default value is 0
- OPTIMIZER_USE_INVISIBLE_INDEXES enables or disables the use of invisible indexes.
- OPTIMIZER_USE_PENDING_STATISTICS specifies whether or not the optimizer uses pending statistics when compiling SQL statements.

Note: Invisible indexes, pending statistics, and dynamic sampling are discussed later in this course.



Features	9.0.0 to 9.2.0	10.1.0 to 10.1.0.5	10.2.0 to 10.2.0.2	11.1.0.6
Index fast full scan	✓	✓	✓	✓
Consideration of bitmap access to paths for tables with only B-tree indexes	✓	✓	✓	✓
Complex view merging	✓	✓	✓	✓
Peeking into user-defined bind variables	✓	✓	✓	✓
Index joins	✓	✓	✓	✓
Dynamic sampling		✓	✓	✓
Query rewrite enables		✓	✓	✓
Skip unusable indexes		✓	✓	✓
Automatically compute index statistics as part of creation		✓	✓	✓
Cost-based query transformations		✓	✓	✓
Allow rewrites with multiple MVs and/or base tables			✓	✓
Adaptive cursor sharing				✓
Use extended statistics to estimate selectivity				✓
Use native implementation for full outer joins				✓
Partition pruning using join filtering				✓
Group by placement optimization				✓
Null aware antijoins				✓

Optimizer Features and Oracle Database Releases

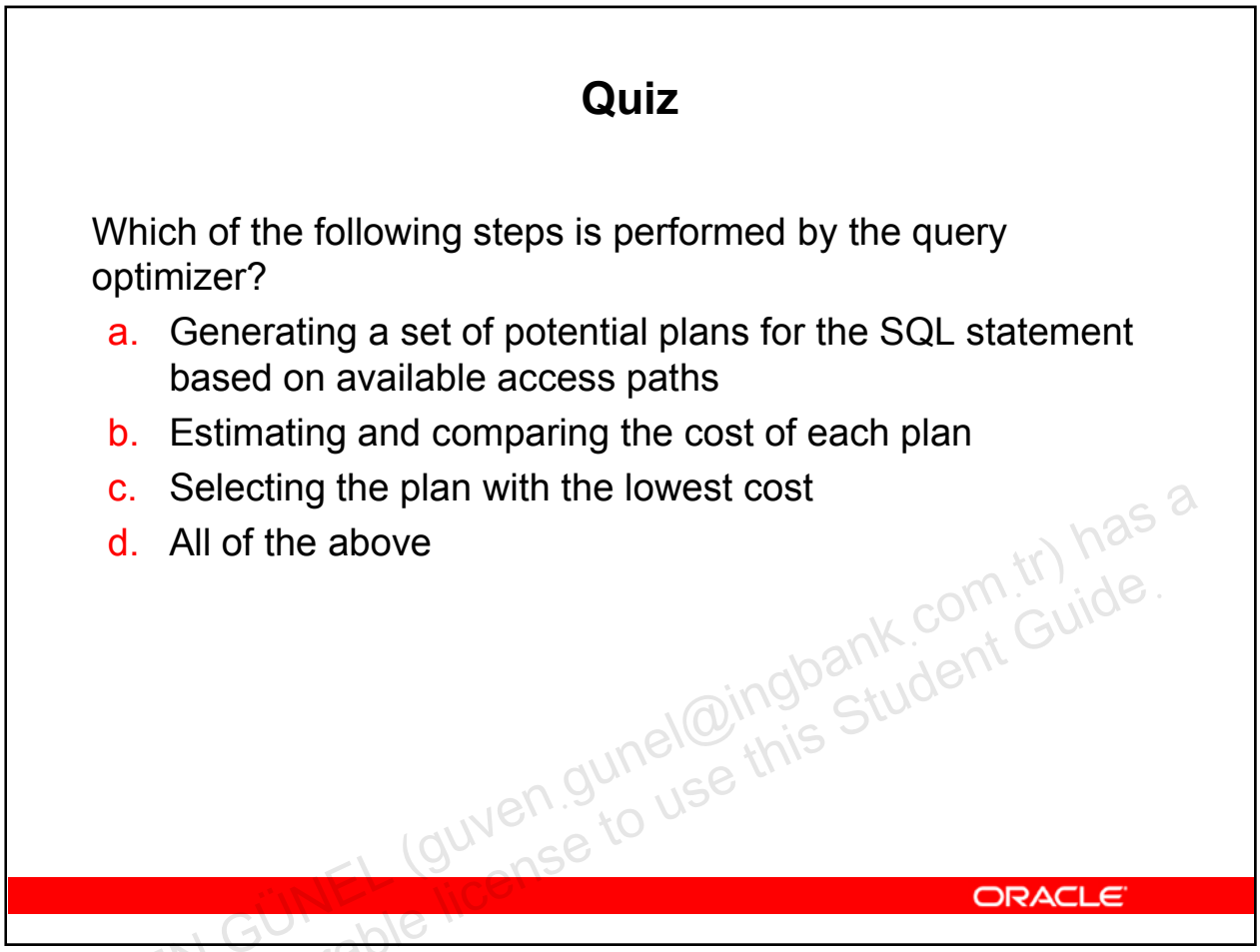
`OPTIMIZER_FEATURES_ENABLED` acts as an umbrella parameter for enabling a series of optimizer features based on an Oracle release number. The table in the slide describes some of the optimizer features that are enabled depending on the value specified for the `OPTIMIZER_FEATURES_ENABLED` parameter.

Quiz

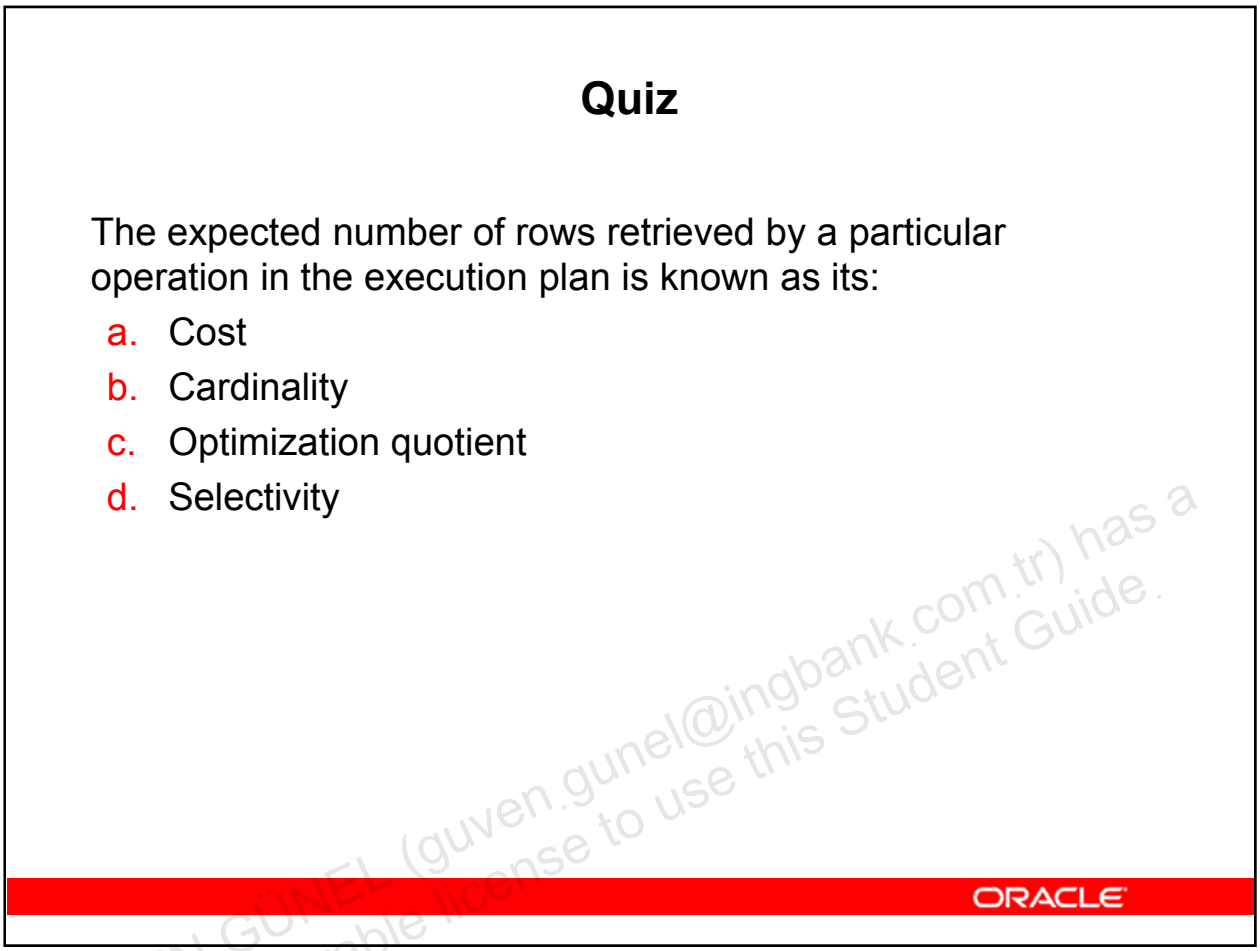
The _____step provides information about the select list items and is relevant when entering dynamic queries through an OCI application.

a. Parse
b. Define
c. Describe
d. Parallelize

Answer: c



Answer: d



Answer: b

Summary

In this lesson, you should have learned how to:

- Describe the execution steps of a SQL statement
- Describe the need for an optimizer
- Explain the various phases of optimization
- Control the behavior of the optimizer

Practice 3: Overview

This practice covers exploring a trace file to understand the optimizer's decisions.