Facilitator Notes

3 P's

- Purpose :
- To learn an overview of Performance Tuning
- Product :
- Learn to understand the basics of Tuning
  - Describe what attributes of a SQL statement can make it perform poorly
  - List the Oracle tools that can be used to tune SQL
- Process:
- Instructor led training with practical experience

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 1

Coverage

- Introduction to SQL tuning
- Describe why the SQL statements are performing poorly
- Introduction to Oracle Optimizer
- Discuss the need for Optimizer
- Explain the various phases of Optimization
- Gather Execution Plans
- Interpret Execution Plans
- Interpret the output of TKPROF
- Gather Optimizer statistics
- Use Hints appropriately

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

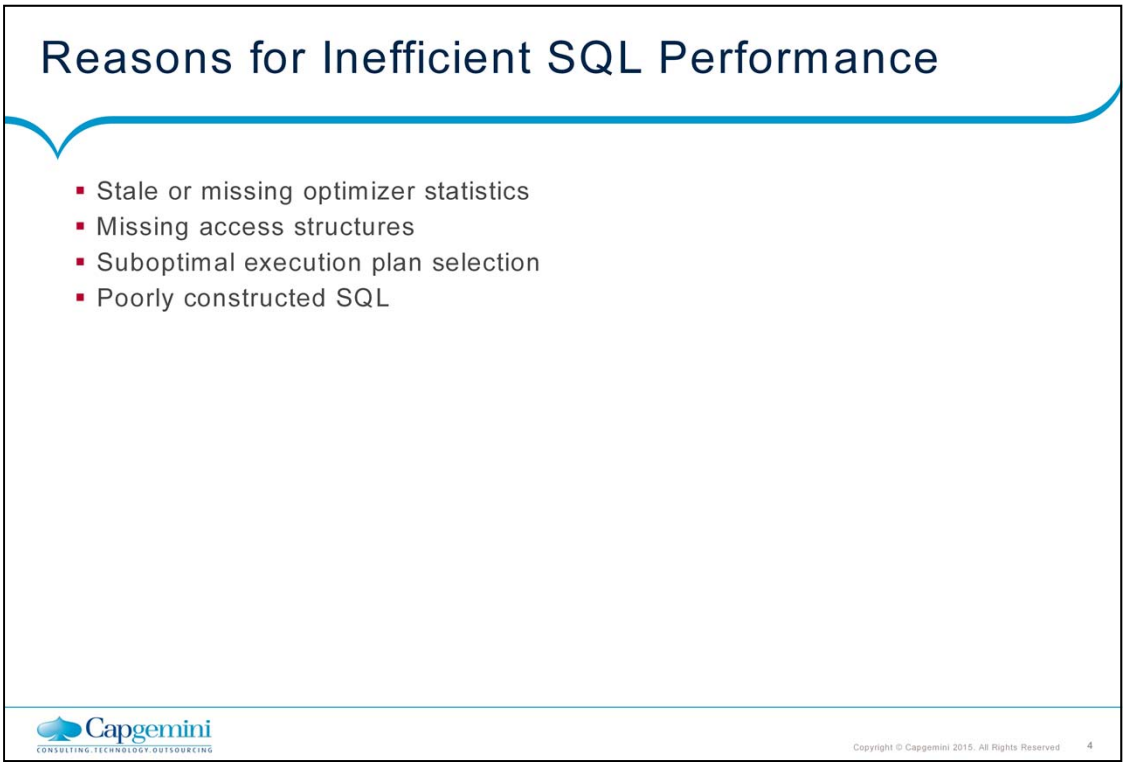
 2

Introduction to SQL tuning

- Reasons for Inefficient SQL Performance
- SQL Tuning tasks
- Proactive tuning Methodology

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 3



Reasons for Inefficient SQL Performance

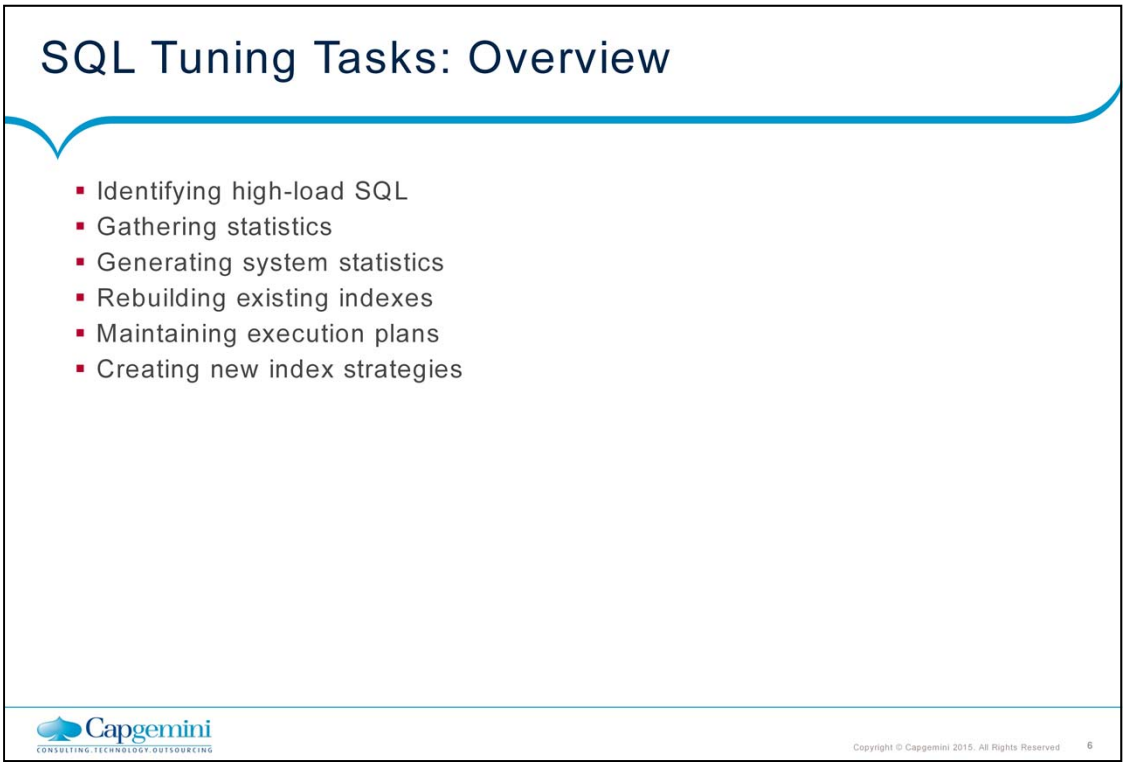
- SQL statements can perform poorly for a variety of reasons:
  - **Stale optimizer statistics:** SQL execution plans are generated by the cost-based optimizer (CBO). For CBO to effectively choose the most efficient plan, it needs accurate information on the data volume and distribution of tables and indexes referenced in the queries. Without accurate optimizer statistics, the CBO can easily be mislead and generate suboptimal execution plans.
  - **Missing access structures:** Absence of access structures, such as indexes, materialized views, and partitions is a common reason for poor SQL performance. The right set of access structures can improve SQL performance by several orders of magnitude.
  - **Suboptimal execution plan selection:** The CBO can sometimes select a suboptimal execution plan for a SQL statement. This happens for most part because of incorrect estimates of some attributes of that SQL statement, such as its cost, cardinality, or predicate selectivity.

Inefficient SQL Queries

```
SELECT COUNT(*) FROM products p   WHERE prod_list_price < 1.15 (SELECT
   avg(unit_cost) FROM costs c
WHERE c.prod_id = p.prod_id)

 SELECT * FROM job_history jh, employees e
 WHERE substr(to_char(e.employee_id),2) =
 substr(to_char(jh.employee_id),2)

 SELECT * FROM orders WHERE order_id_char = 1205

 SELECT * FROM employees
    WHERE to_char(salary) = :sal

 SELECT * FROM parts_old
 UNION
 SELECT * FROM parts_new
```

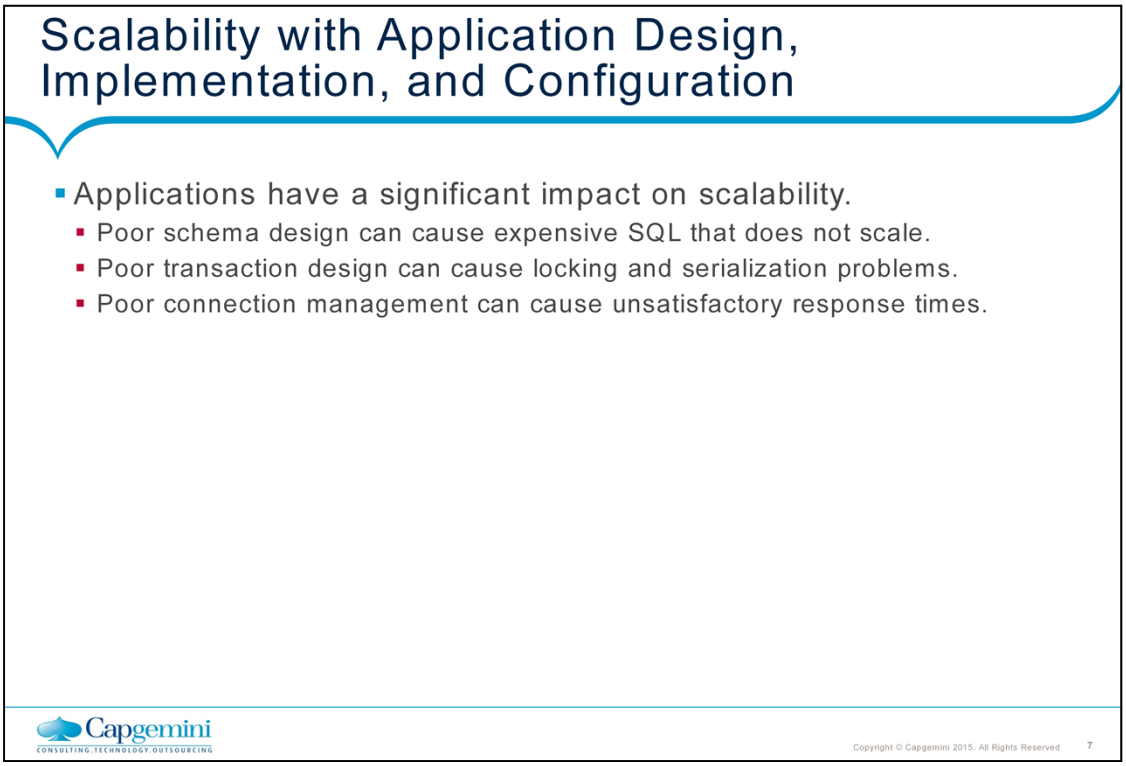
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 5



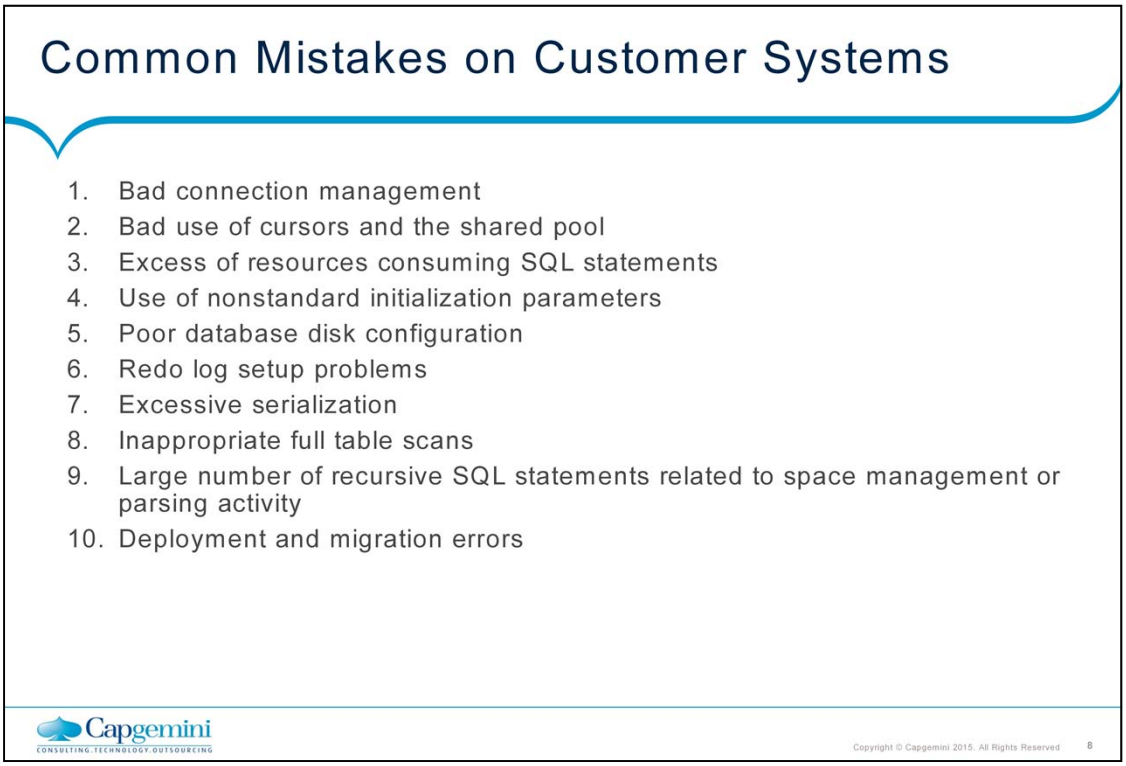
SQL Tuning Tasks: Overview

- Many SQL tuning tasks should be performed on a regular basis. You may see a way to rewrite a `WHERE` clause, but it may depend on a new index being built. This list of tasks gives you a background of some important tasks that must be performed, and gives you an idea of what dependencies you may have as you tune SQL:
  - Identifying high-load SQL statements is one of the most important tasks you should perform. The ADDM is the ideal tool for this particular task.
  - By default, the Oracle Database gathers optimizer statistics automatically. For this, a job is scheduled to run in the maintenance windows.
  - Operating system statistics provide information about the usage and performance of the main hardware components as well as the performance of the operating system itself.
  - Often, there is a beneficial impact on performance by rebuilding indexes. For example, removing nonselective indexes to speed the data manipulation language (DML), or adding columns to the index to improve selectivity.
  - You can maintain the existing execution plan of SQL statements over time by using stored statistics or SQL plan baselines.



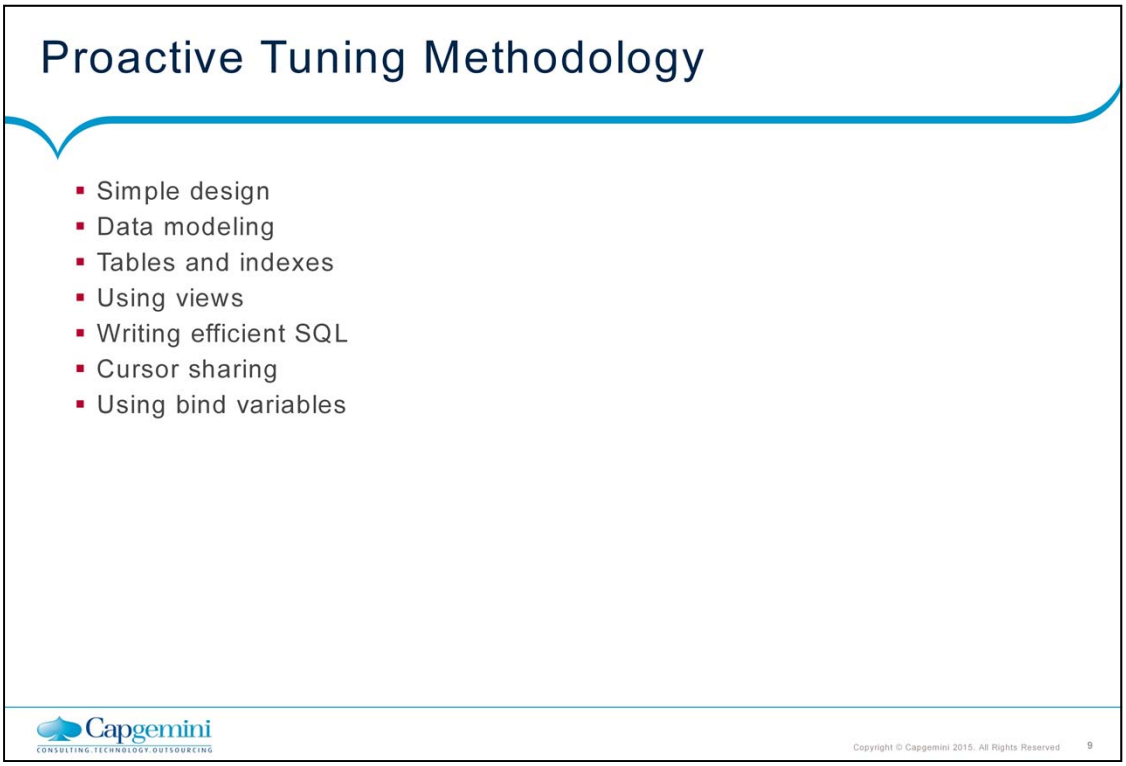
Scalability with Application Design, Implementation, and Configuration

- Poor application design, implementation, and configuration have a significant impact on scalability. This results in:
  - Poor SQL and index design, resulting in a higher number of logical input/output (I/O) for the same number of rows returned
  - Reduced availability because database objects take longer to maintain
- However, design is not the only problem. The physical implementation of the application can be the weak link, as in the following examples:
  - Systems can move to production environments with poorly written SQL that cause high I/O.
  - Infrequent transaction COMMITs or ROLLBACKs can cause long locks on resources.
  - The production environment can use different execution plans than those generated in testing.
  - Memory-intensive applications that allocate a large amount of memory without much thought for freeing the memory can cause excessive memory fragmentation.
  - Inefficient memory usage places high stress on the operating virtual memory subsystem. This affects performance and availability.



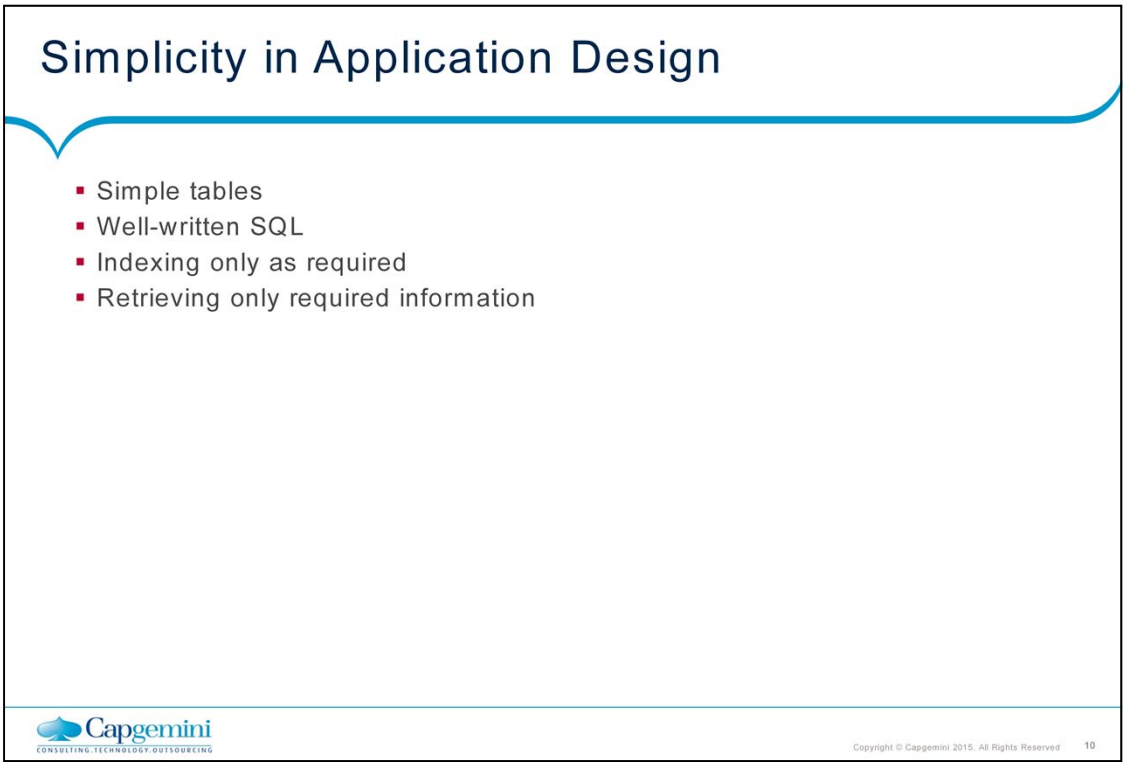
Common Mistakes on Customer Systems

- 1. **Bad connection management:** The application connects and disconnects for each database interaction. This problem is common with stateless middleware in application servers. It has over two orders of magnitude impact on performance, and is not scalable.
- 2. **Bad use of cursors and the shared pool:** Not using cursors results in repeated parses. If bind variables are not used, there may be hard parsing of all similar SQL statements. This has an order of magnitude impact on performance, and it is not scalable. Use cursors with bind variables that open the cursor and execute it many times. Be suspicious of applications generating dynamic SQL.
- 3. **Bad SQL:** Bad SQL is SQL that uses more resources than appropriate for the application. This can be a decision support system (DSS) query that runs for more than 24 hours or a query from an online application that takes more than a minute. SQL that consumes significant system resources should be investigated for potential improvement. ADDM identifies high-load SQL and the SQL Tuning Advisor can be used to provide recommendations for improvement.



Proactive Tuning Methodology

- Tuning usually implies fixing a performance problem. However, tuning should be part of the life cycle of an application, through the analysis, design, coding, production, and maintenance stages. The tuning phase is often left until the system is in production. At that time, tuning becomes a reactive exercise, where the most important bottleneck is identified and fixed.
- The slide lists some of the issues that affect performance and that should be tuned proactively instead of reactively. These are discussed in more detail in the following slides.



Simplicity in Application Design

- Applications are no different from any other designed and engineered product. If the design looks right, it probably is right. This principle should always be kept in mind when building applications. Consider some of the following design issues:
  - If the table design is so complicated that nobody can fully understand it, the table is probably designed badly.
  - If SQL statements are so long and involved that it would be impossible for any optimizer to effectively optimize it in real time, there is probably a bad statement, underlying transaction, or table design.
  - If there are many indexes on a table and the same columns are repeatedly indexed, there is probably a bad index design.
  - If queries are submitted without suitable qualification (the `WHERE` clause) for rapid response for online users, there is probably a bad user interface or transaction design.

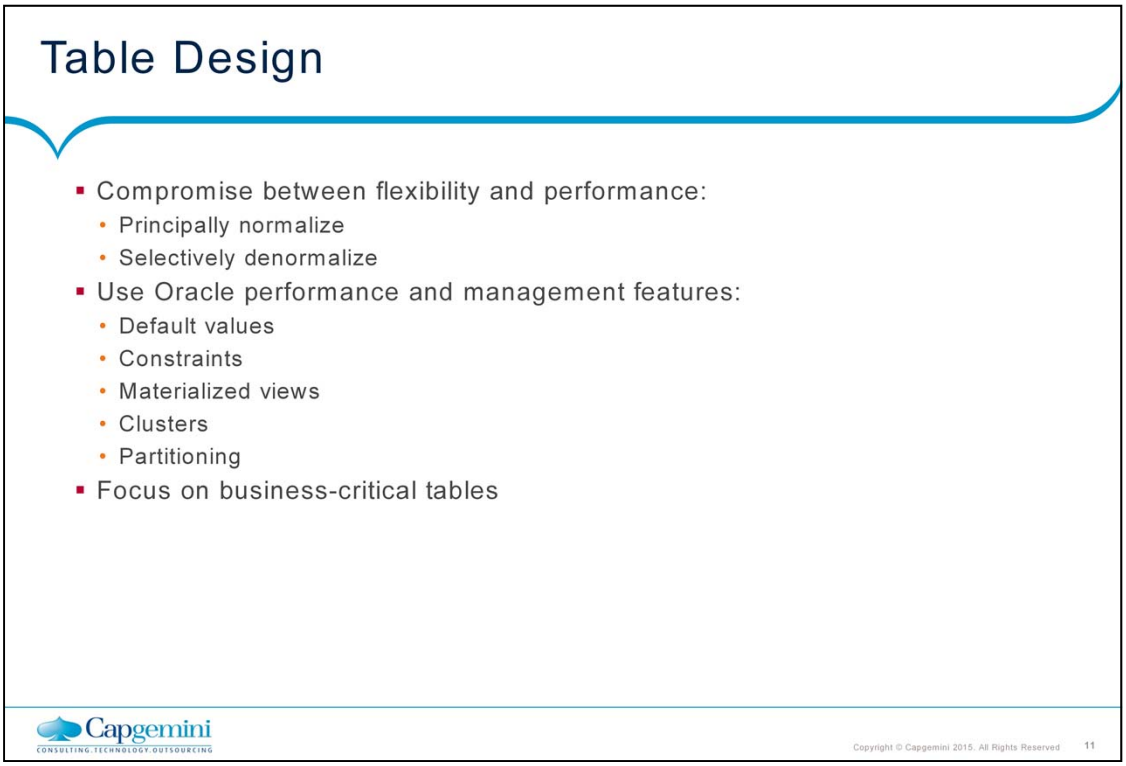
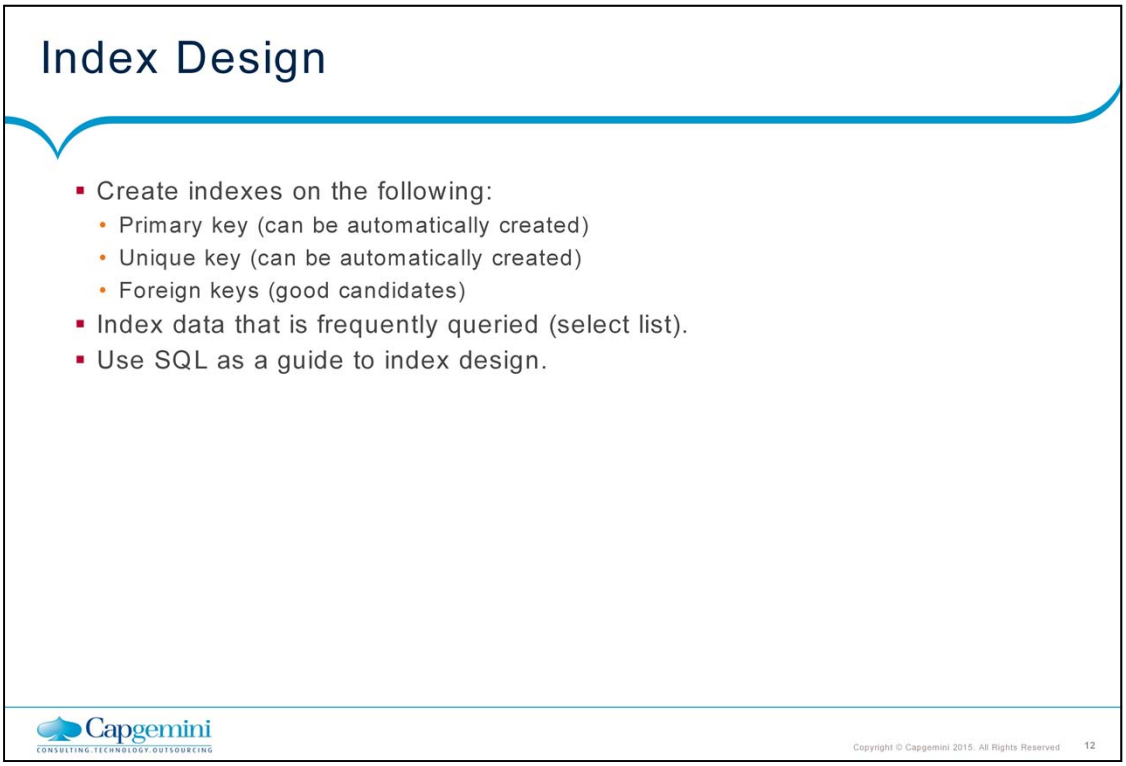


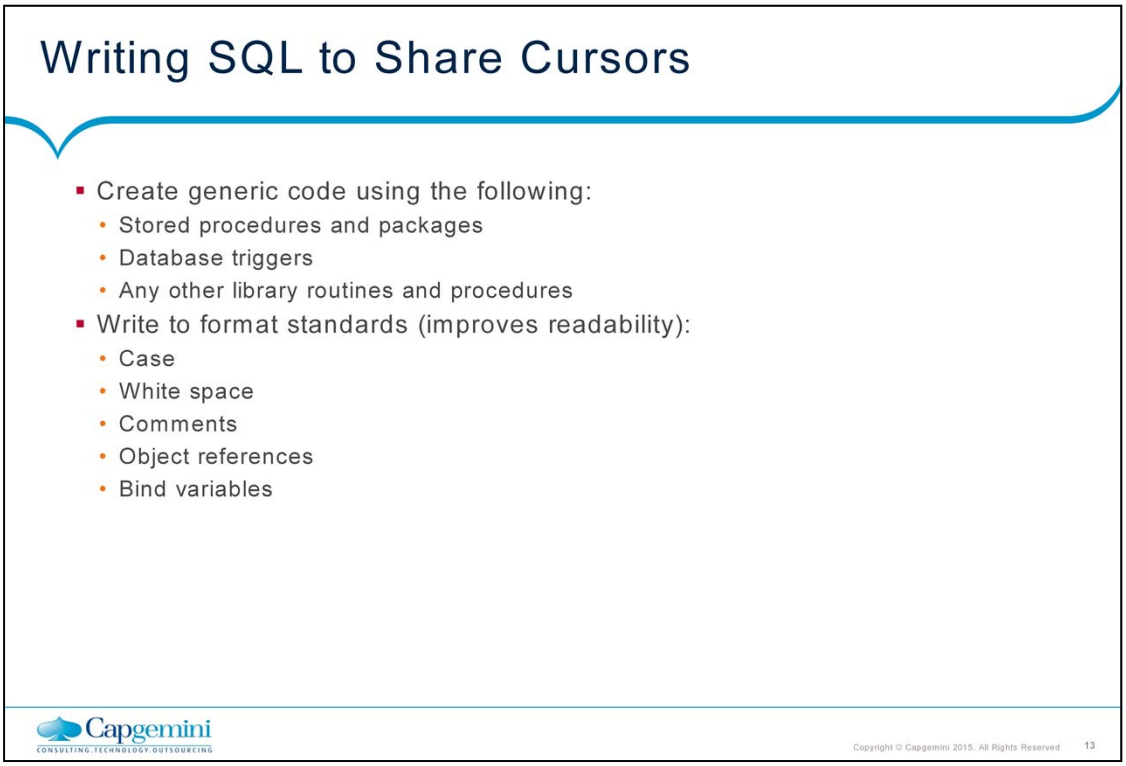
Table Design

- Table design is largely a compromise between flexibility and performance of core transactions. To keep the database flexible and able to accommodate unforeseen workloads, the table design should be very similar to the data model, and it should be normalized to at least third normal form. However, certain core transactions can require selective denormalization for performance purposes.
- Use the features supplied with Oracle Database to simplify table design for performance, such as storing tables prejoined in clusters, adding derived columns and aggregate values, and using materialized views or partitioned tables. Additionally, create check constraints and columns with default values to prevent bad data from getting into the tables.
- Design should be focused on business-critical tables so that good performance can be achieved in areas that are the most used. For noncritical tables, shortcuts in design can be adopted to enable a more rapid application development. If, however, a noncore table becomes a performance problem during prototyping and testing, remedial design efforts should be applied immediately.



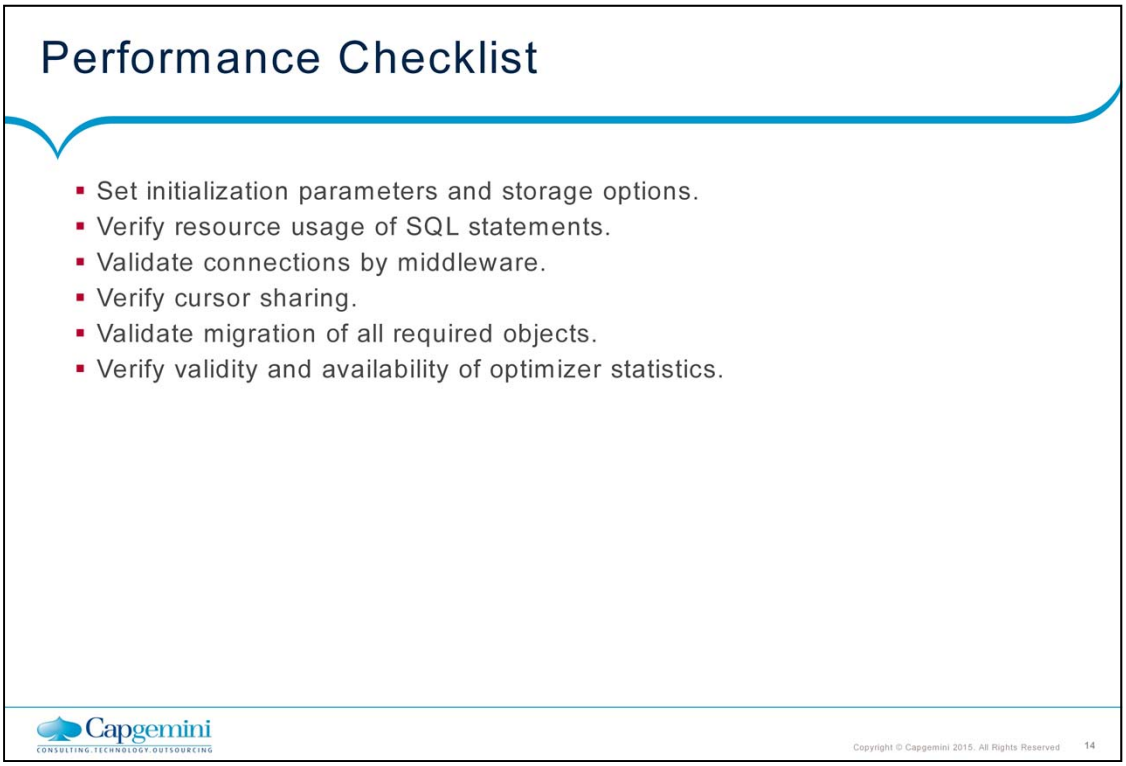
Index Design

- Index design is also a largely iterative process based on the SQL that is generated by application designers. However, it is possible to make a sensible start by building indexes that enforce foreign key constraints (to reduce response time on joins between primary key tables and foreign key tables) and creating indexes on frequently accessed data, such as a person's name. Primary keys and unique keys are automatically indexed except for the DISABLE VALIDATE and DISABLE NOVALIDATE RELY constraints. As the application evolves and testing is performed on realistic sizes of data, certain queries need performance improvements, for which building a better index is a good solution.
- The following indexing design ideas should be considered when building a new index.
- Appending Columns to an Index or Using Index-Organized Tables
- One of the easiest ways to speed up a query is to reduce the number of logical I/Os by eliminating a table scan from the execution plan. This can be done by creating an index over all the columns of the table referenced by the query. These columns are the select list columns, WHERE clause columns, and any required join or sort columns. This technique is particularly useful in speeding up an online application's response times when time-consuming I/Os are reduced. This is best applied when testing the application with
properly-sized data for the first time. The most aggressive form of this technique is to build an
index-organized table (IOT).



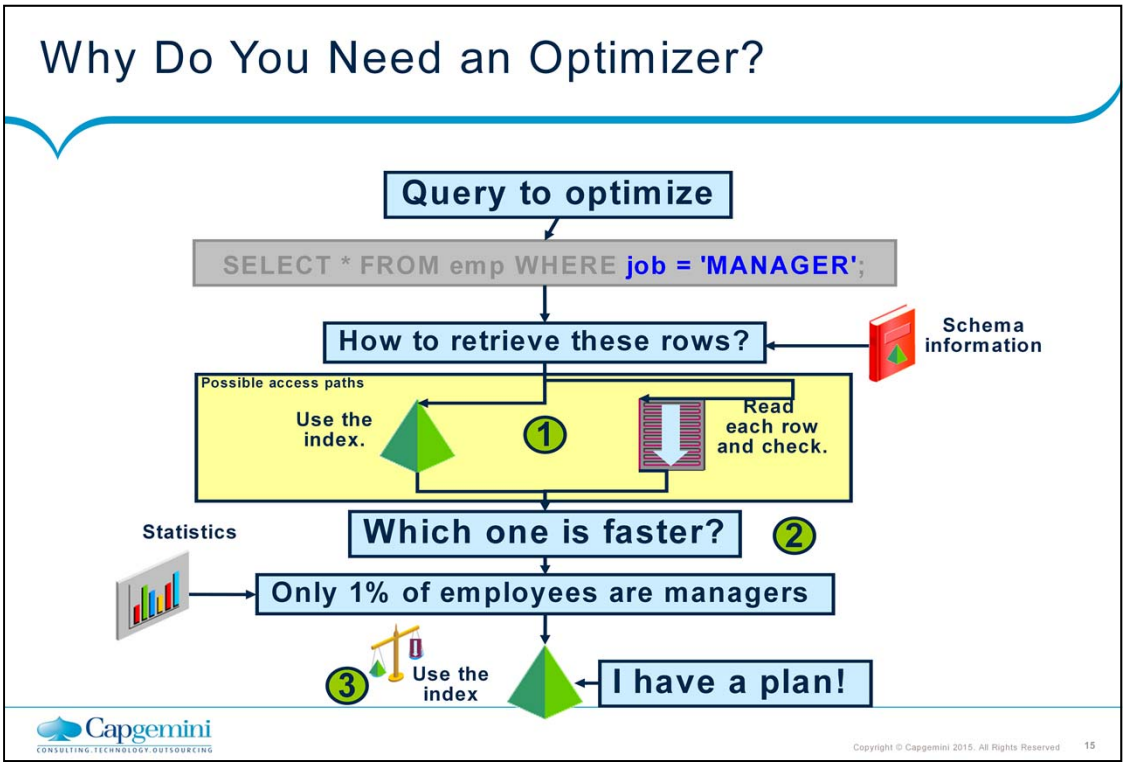
Writing SQL to Share Cursors

- Applications can share cursors when the code is written in the same way characterwise (which allows the system to recognize that two statements are the same and thus can be shared), even if you use some special initialization parameters, such as CURSOR_SHARING discussed later in the lesson titled “Using Bind Variables.” You should develop coding conventions for SQL statements in ad hoc queries, SQL scripts, and Oracle Call Interface (OCI) calls.
- Use generic shared code:
  - Write and store procedures that can be shared across applications.
  - Use database triggers.
  - Write referenced triggers and procedures when using application development tools.
  - Write library routines and procedures in other environments.
- Write to format standards:
  - Develop format standards for all statements, including those in PL/SQL code.
  - Develop rules for the use of uppercase and lowercase characters.
  - Develop rules for the use of white space (spaces, tabs, returns).
  - Develop rules for the use of comments (preferably keeping them out of the SQL statements themselves).
  - Use the same names to refer to identical database objects. If possible, prefix each object with a schema name.



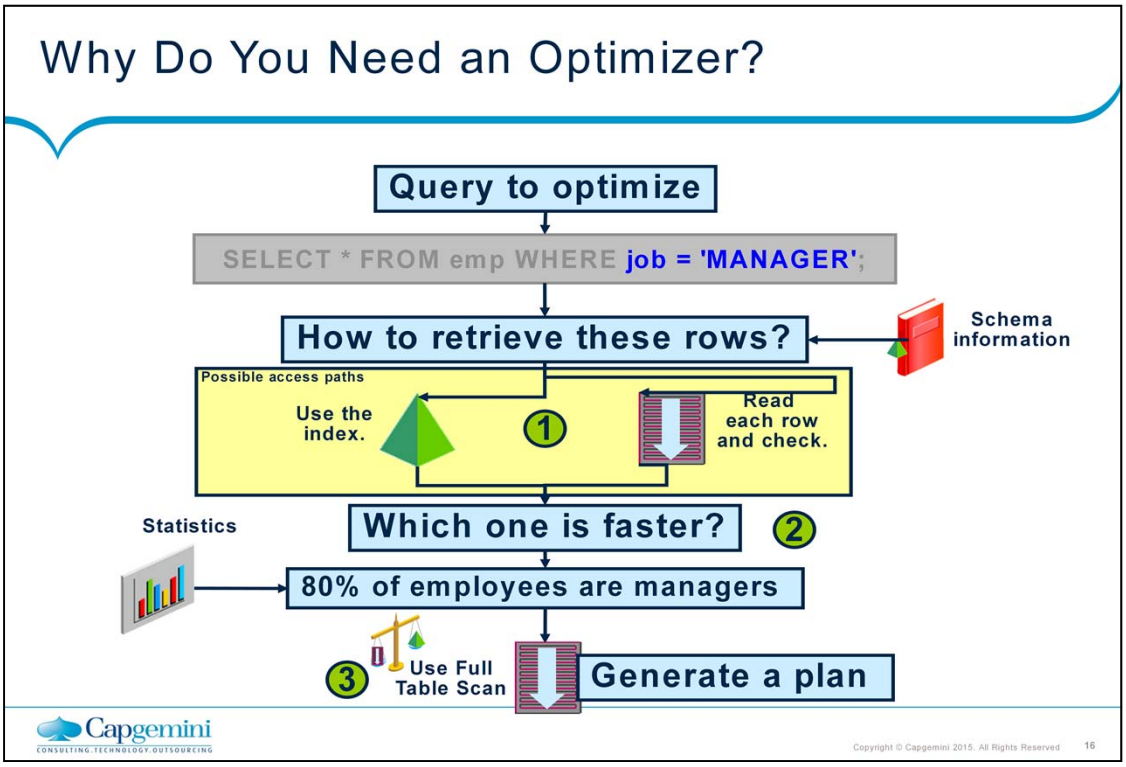
Performance Checklist

- Set the minimal number of initialization parameters. Ideally, most initialization parameters should be left at default. If there is more tuning to perform, this shows up when the system is under load. Set storage options for tables and indexes in appropriate tablespaces.
- Verify that all SQL statements are optimal and understand their resource usage.
- Validate that middleware and programs that connect to the database are efficient in their connection management and do not log on and log off repeatedly.
- Validate that the SQL statements use cursors efficiently. Each SQL statement should be parsed once and then executed multiple times. This happens mostly when bind variables are not used properly and the WHERE clause predicates are sent as string literals.
- Validate that all schema objects are correctly migrated from the development environment to the production database. This includes tables, indexes, sequences, triggers, packages, procedures, functions, Java objects, synonyms, grants, and views. Ensure that any modifications made in testing are made to the production system.
- As soon as the system is rolled out, establish a baseline set of statistics from the database and operating system. This first set of statistics validates or corrects any assumptions made in the design and rollout process.



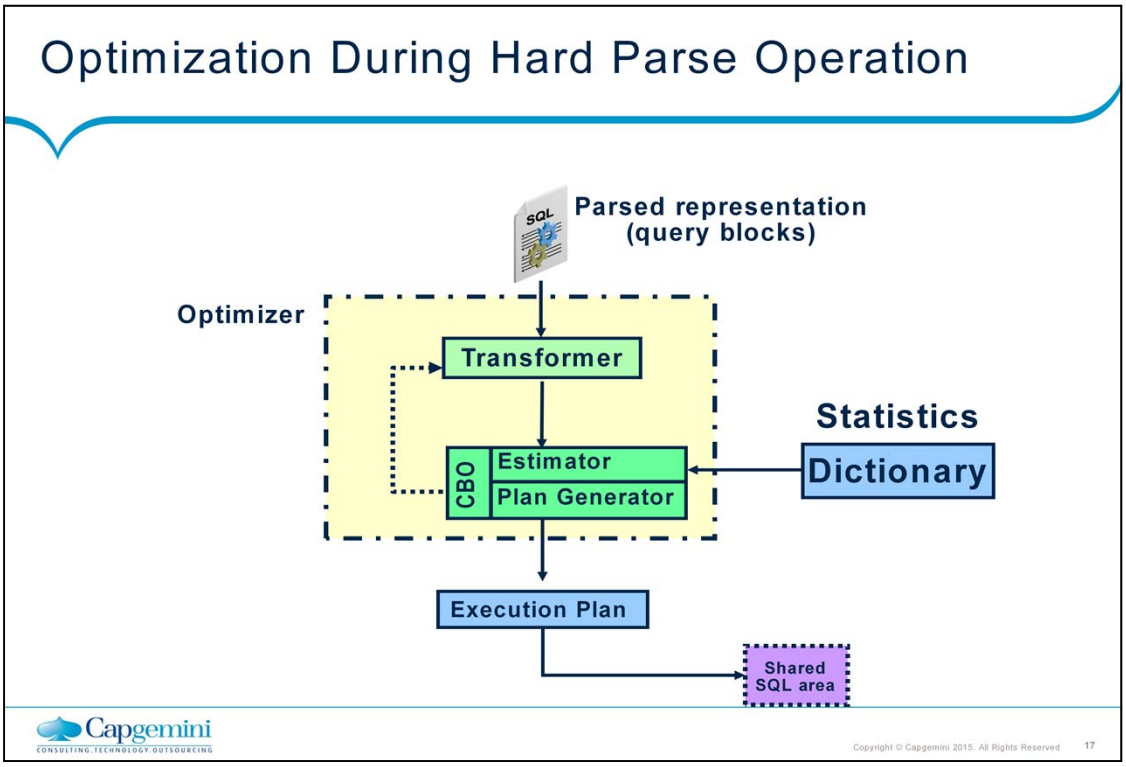
Why Do You Need an Optimizer?

- 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.



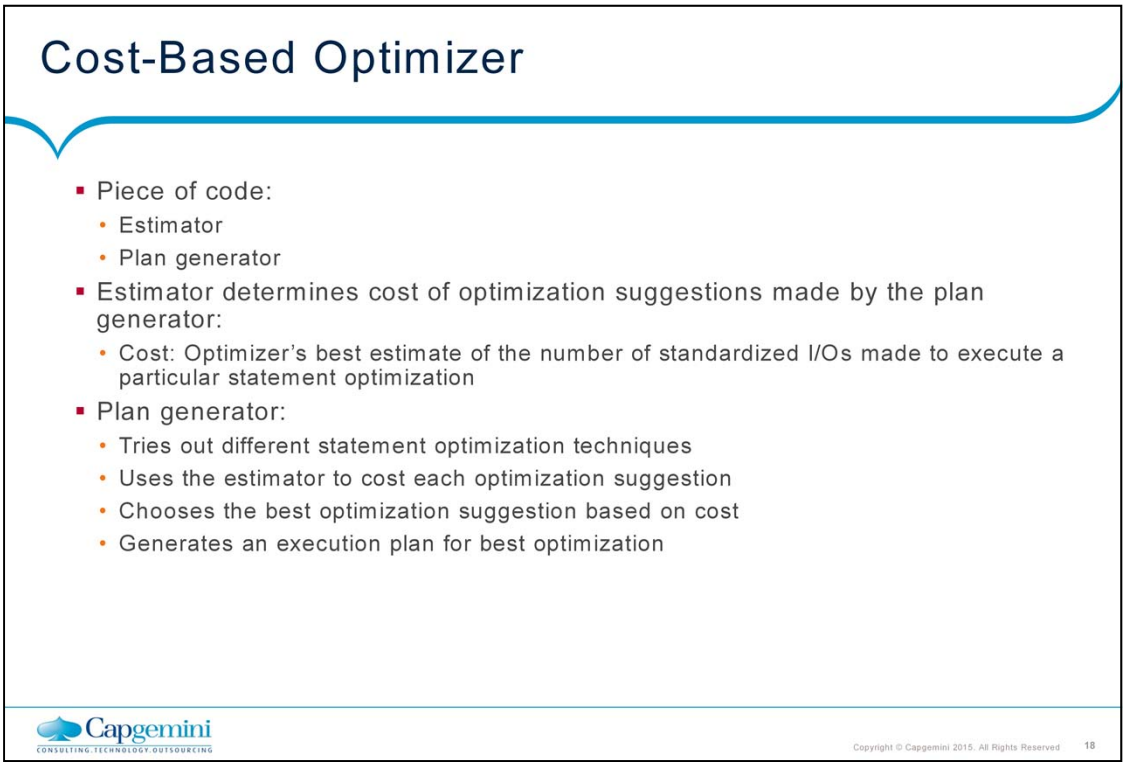
Why Do You Need an Optimizer? (continued)

- 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.



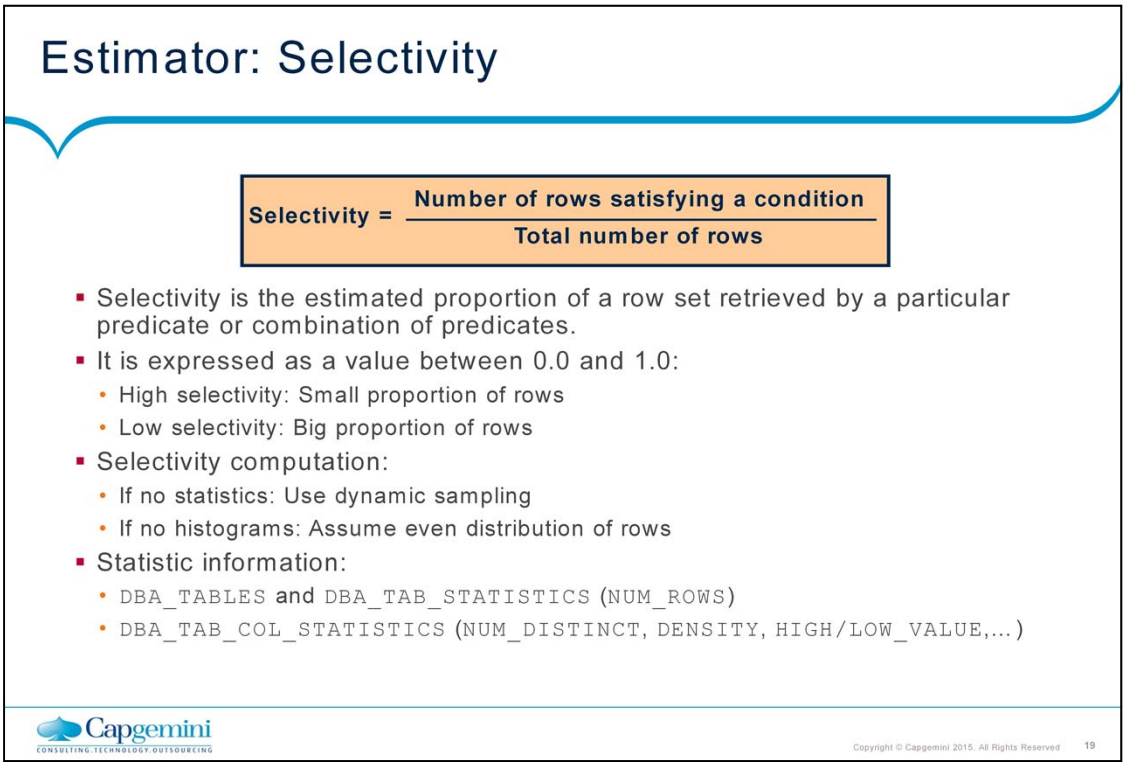
Optimization During Hard Parse Operation

- 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.



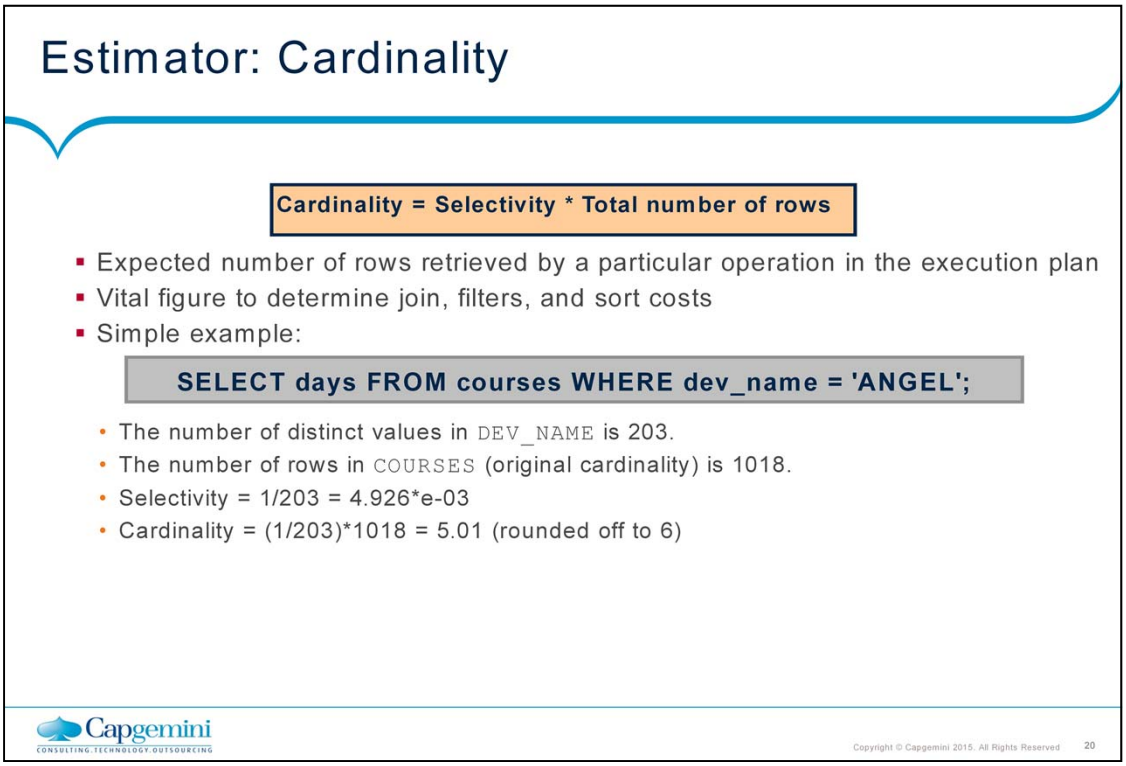
Cost-Based Optimizer

- 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.



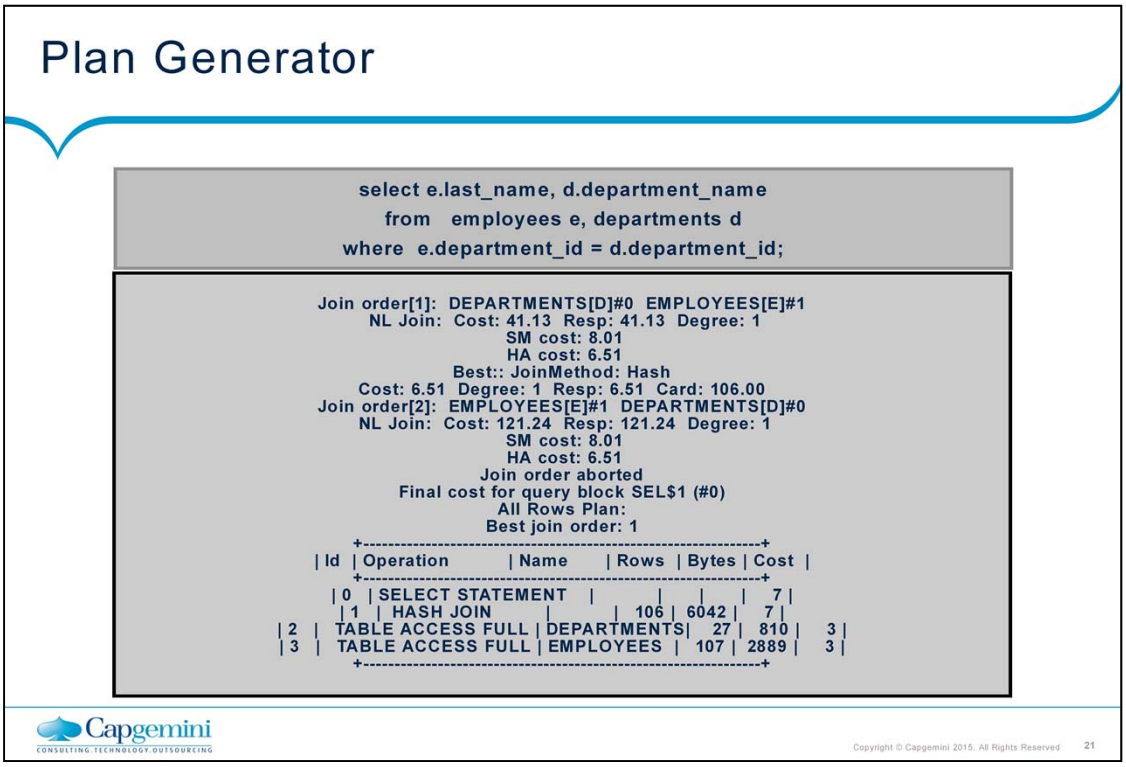
Estimator: Selectivity

- 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.



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.”

What Is an Execution Plan?

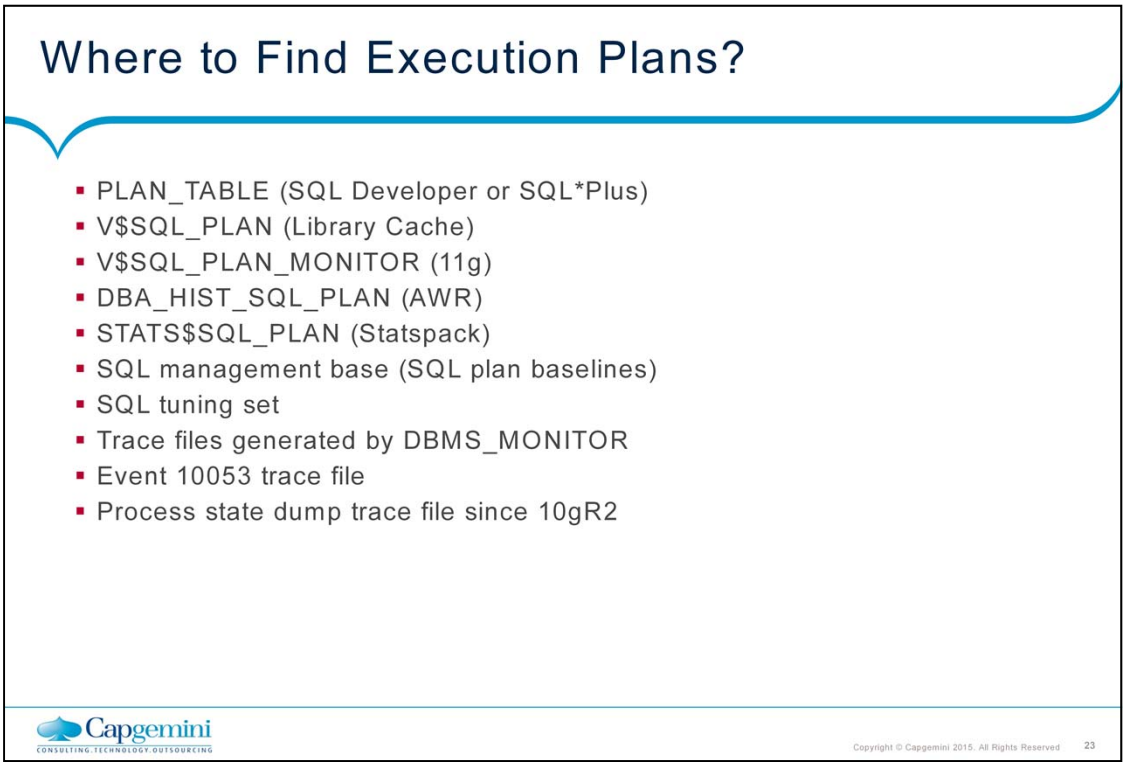
- The execution plan of a SQL statement is composed of small building blocks called row sources for serial execution plans.
- The combination of row sources for a statement is called the execution plan.
- By using parent-child relationships, the execution plan can be displayed in a tree-like structure (text or graphical).

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 22

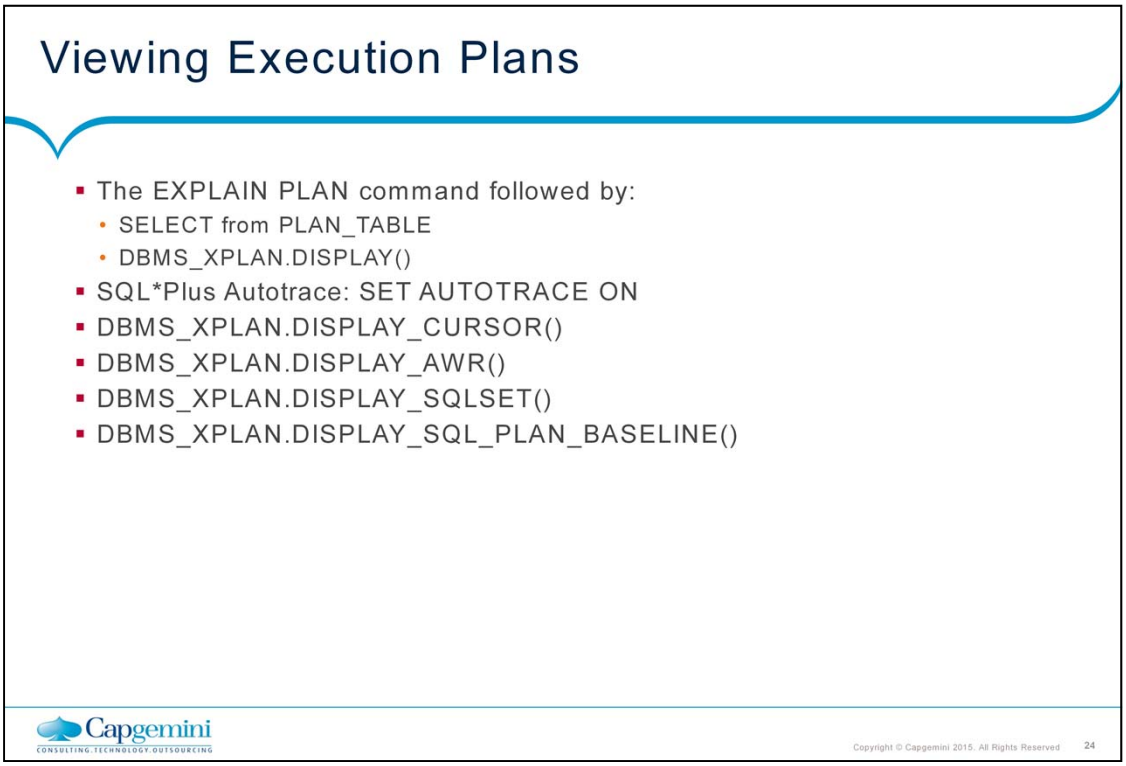
What Is an Execution Plan?

- An execution plan is the output of the optimizer and is presented to the execution engine for implementation. It instructs the execution engine about the operations it must perform for retrieving the data required by a query most efficiently.
- The EXPLAIN PLAN statement gathers execution plans chosen by the Oracle optimizer for the SELECT, UPDATE, INSERT, and DELETE statements. The steps of the execution plan are not performed in the order in which they are numbered. There is a parent-child relationship between steps. The row source tree is the core of the execution plan. It shows the following information:
  - An ordering of the tables referenced by the statement
  - An access method for each table mentioned in the statement
  - A join method for tables affected by join operations in the statement
  - Data operations, such as filter, sort, or aggregation
- In addition to the row source tree (or data flow tree for parallel operations), the plan table contains information about the following:
  - Optimization, such as the cost and cardinality of each operation
  - Partitioning, such as the set of accessed partitions
  - Parallel execution, such as the distribution method of join inputs
- The EXPLAIN PLAN results help you determine whether the optimizer selects a particular execution plan, such as nested loops join.



Where to Find Execution Plans?

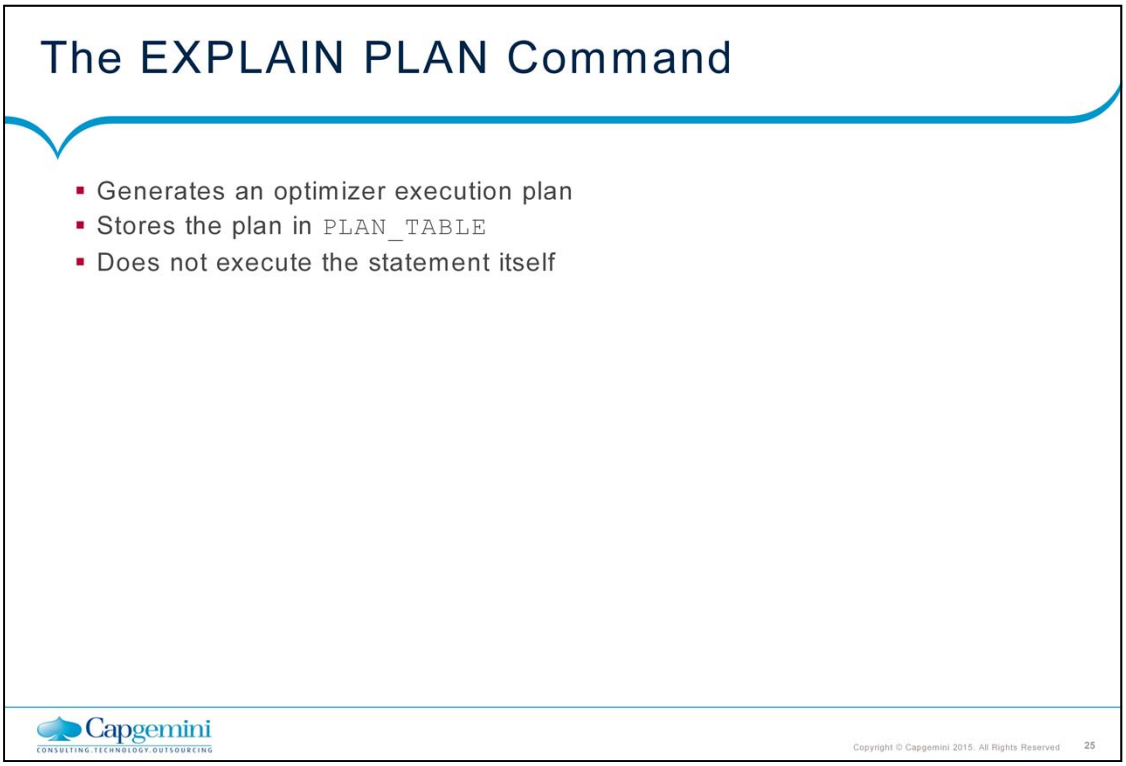
- There are many ways to retrieve execution plans inside the database. The most well-known ones are listed in the slide:
  - The EXPLAIN PLAN command enables you to view the execution plan that the optimizer might use to execute a SQL statement. This command is very useful because it outlines the plan that the optimizer may use and inserts it in a table called PLAN_TABLE without executing the SQL statement. This command is available from SQL*Plus or SQL Developer.
  - V$SQL_PLAN provides a way to examine the execution plan for cursors that were recently executed. Information in V$SQL_PLAN is very similar to the output of an EXPLAIN PLAN statement. However, while EXPLAIN PLAN shows a theoretical plan that can be used if this statement was executed, V$SQL_PLAN contains the actual plan used.
  - V$SQL_PLAN_MONITOR displays plan-level monitoring statistics for each SQL statement found in V$SQL_MONITOR. Each row in V$SQL_PLAN_MONITOR corresponds to an operation of the execution plan that is monitored.
  - The Automatic Workload Repository (AWR) infrastructure and Statspack store execution plans of top SQL statements. Plans are recorded into DBA_HIST_SQL_PLAN or STATS$SQL_PLAN.



Viewing Execution Plans

- If you execute the `EXPLAIN PLAN` SQL*Plus command, you can then `SELECT` from the `PLAN_TABLE` to view the execution plan. There are several SQL*Plus scripts available to format the plan table output. The easiest way to view an execution plan is to use the `DBMS_XPLAN` package. The `DBMS_XPLAN` package supplies five table functions:
  - `DISPLAY`: To format and display the contents of a plan table
  - `DISPLAY_AWR`: To format and display the contents of the execution plan of a stored SQL statement in the AWR
  - `DISPLAY_CURSOR`: To format and display the contents of the execution plan of any loaded cursor
  - `DISPLAY_SQL_PLAN_BASELINE`: To display one or more execution plans for the SQL statement identified by SQL handle
  - `DISPLAY_SQLSET`: To format and display the contents of the execution plan of statements stored in a SQL tuning set

An advantage of using the `DBMS_XPLAN` package table functions is that the output is formatted consistently without regard to the source.

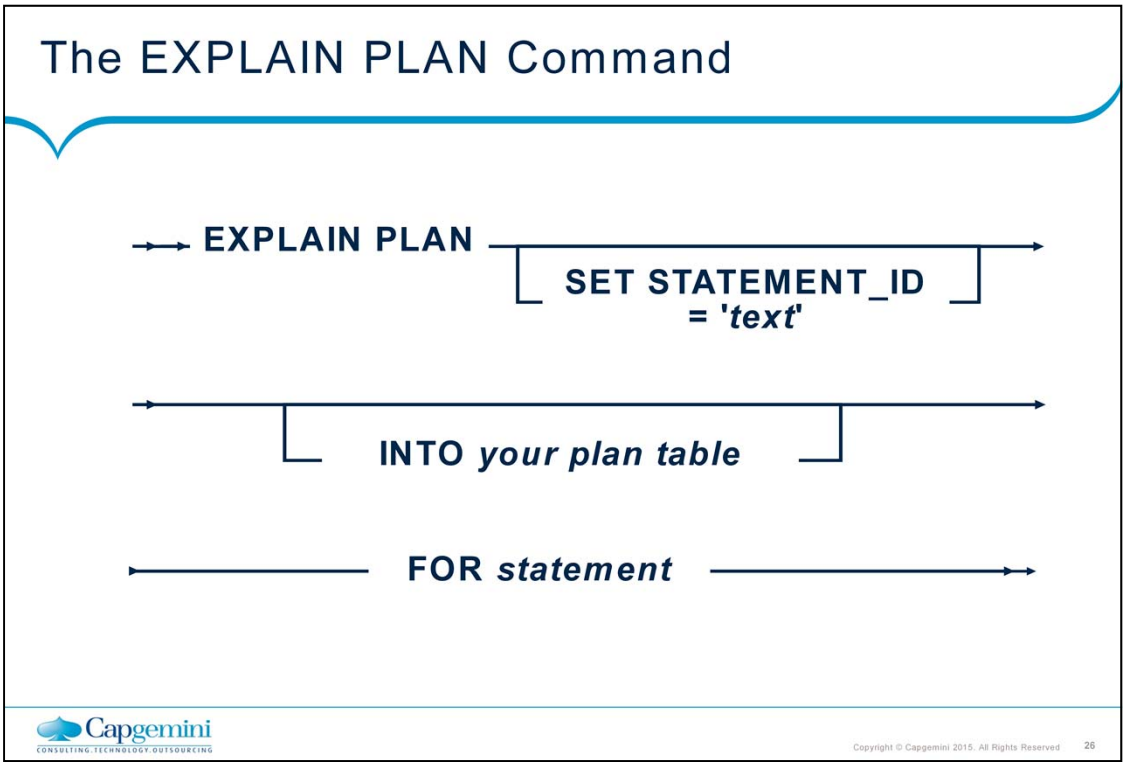


The EXPLAIN PLAN Command

- The EXPLAIN PLAN command is used to generate the execution plan that the optimizer uses to execute a SQL statement. It does not execute the statement, but simply produces the plan that may be used, and inserts this plan into a table. If you examine the plan, you can see how the Oracle Server executes the statement.
- **Using EXPLAIN PLAN**
  - First use the EXPLAIN PLAN command to explain a SQL statement.
  - Then retrieve the plan steps by querying PLAN_TABLE.

PLAN_TABLE is automatically created as a global temporary table to hold the output of an EXPLAIN PLAN statement for all users. PLAN_TABLE is the default sample output table into which the EXPLAIN PLAN statement inserts rows describing execution plans.

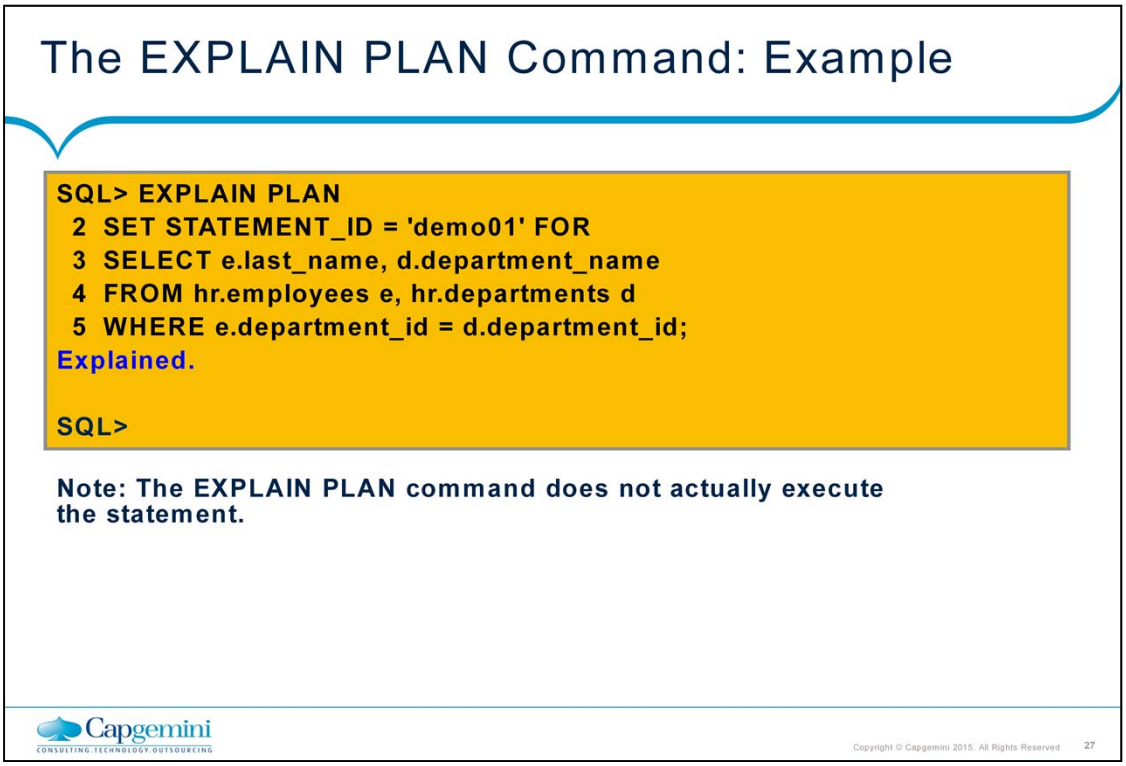
Note: You can create your own PLAN_TABLE using the $ORACLE_HOME/rdbms/admin/utlxplan.sql script if you want to keep the execution plan information for a long term.



The EXPLAIN PLAN Command (continued)

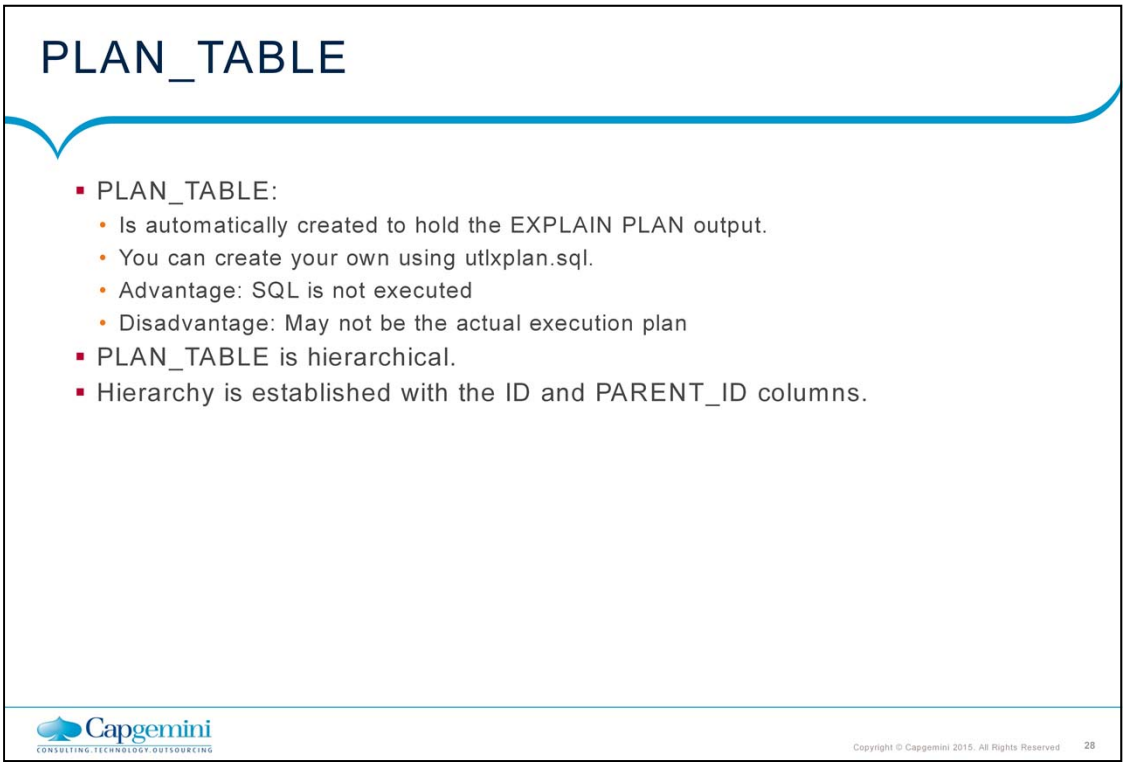
- This command inserts a row in the plan table for each step of the execution plan. In the syntax diagram in the slide, the fields in italics have the following meanings:

Field	Meaning
text	**This is an optional identifier for the statement. You should enter a value to identify each statement so that you can later specify the statement that you want explained. This is especially important when you share the plan table with others, or when you keep multiple execution plans in the same plan table.**
schema.table	**This is the optional name for the output table. The default is PLAN_TABLE.**
statement	**This is the text of the SQL statement.**



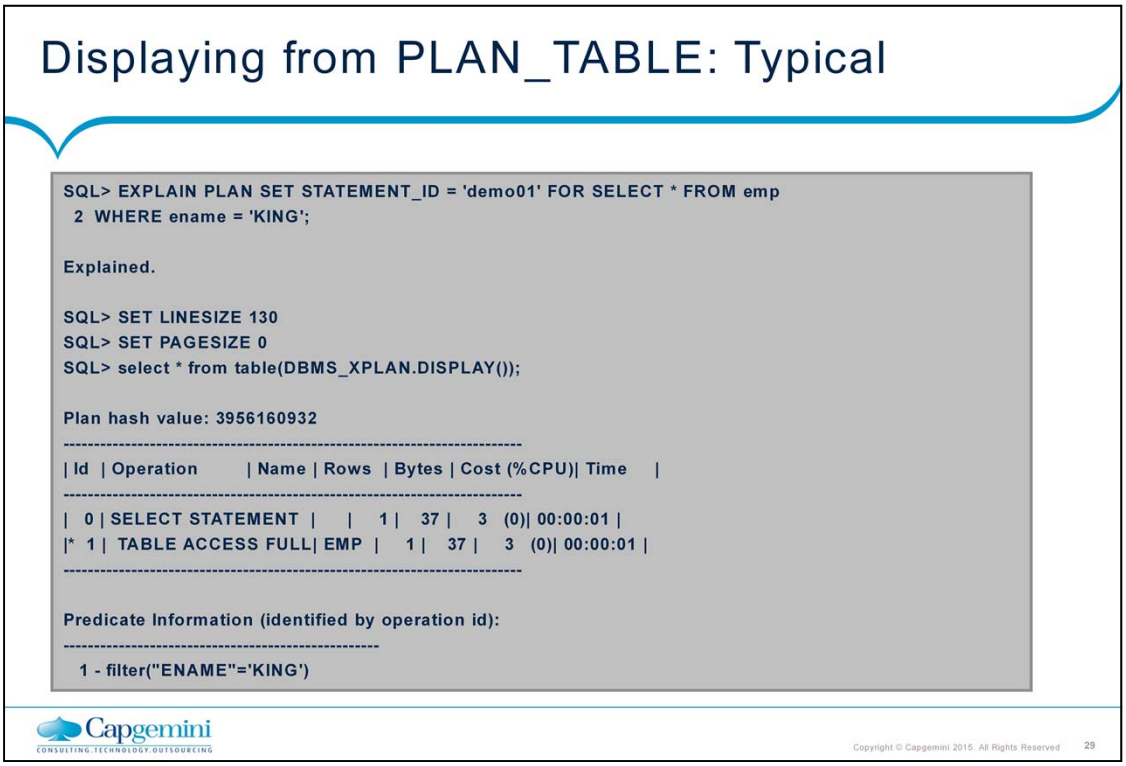
The EXPLAIN PLAN Command: Example

- This command inserts the execution plan of the SQL statement in the plan table and adds the optional demo01 name tag for future reference. You can also use the following syntax:
  - EXPLAIN PLAN
  - FOR
  - SELECT e.last_name, d.department_name
    FROM hr.employees e, hr.departments d
  - WHERE e.department_id =d.department_id;



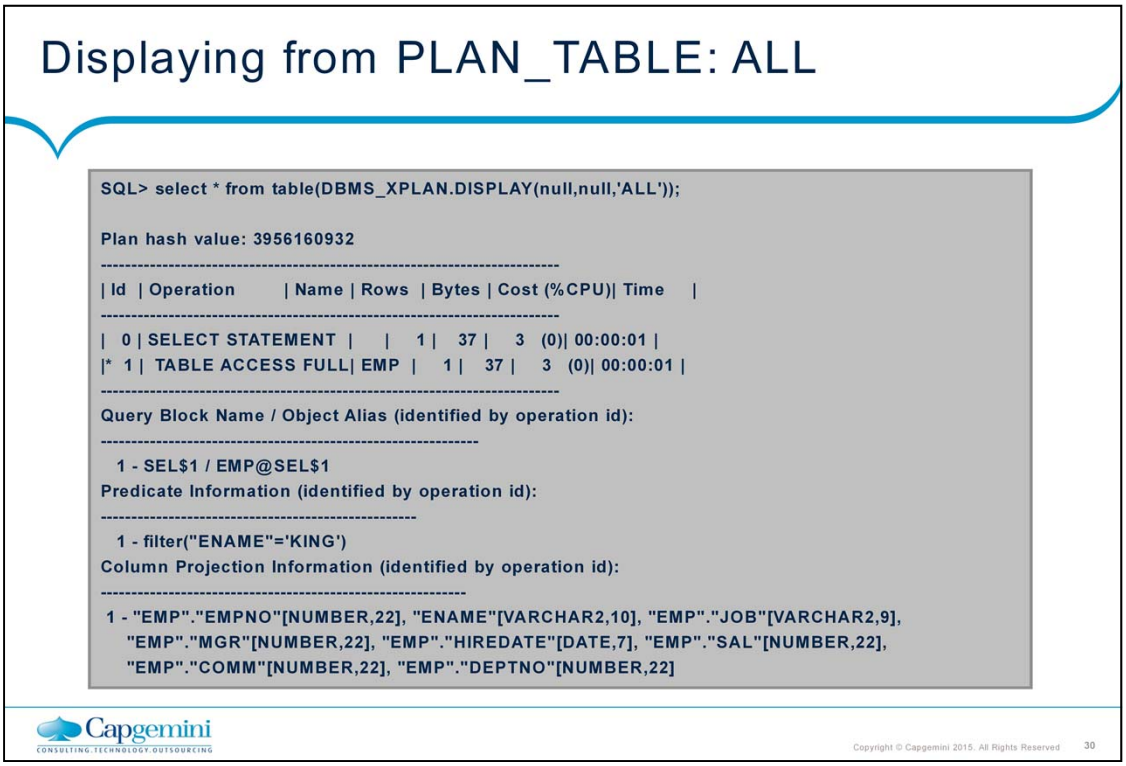
PLAN_TABLE

- There are various available methods to gather execution plans. Now, you are introduced only to the EXPLAIN PLAN statement. This SQL statement gathers the execution plan of a SQL statement without executing it, and outputs its result in the PLAN_TABLE table. Whatever the method to gather and display the explain plan, the basic format and goal are the same. However, PLAN_TABLE just shows you a plan that might not be the one chosen by the optimizer. PLAN_TABLE is automatically created as a global temporary table and is visible to all users. PLAN_TABLE is the default sample output table into which the EXPLAIN PLAN statement inserts rows describing execution plans. PLAN_TABLE is organized in a tree-like structure and you can retrieve that structure by using both the ID and PARENT_ID columns with a CONNECT BY clause in a SELECT statement. While a PLAN_TABLE table is automatically set up for each user, you can use the utlxplan.sql SQL script to manually create a local PLAN_TABLE in your schema and use it to store the results of EXPLAIN PLAN. The exact name and location of this script depends on your operating system. On UNIX, it is located in the $ORACLE_HOME/rdbms/admin directory. It is recommended that you drop and rebuild your local PLAN_TABLE table after upgrading the version of the database because the columns might change. This can cause scripts to fail or cause TKPROF to fail, if you are specifying the table.
- Note: If you want an output table with a different name, first create PLAN_TABLE manually with the utlxplan.sql script, and then rename the table with the RENAME SQL statement.



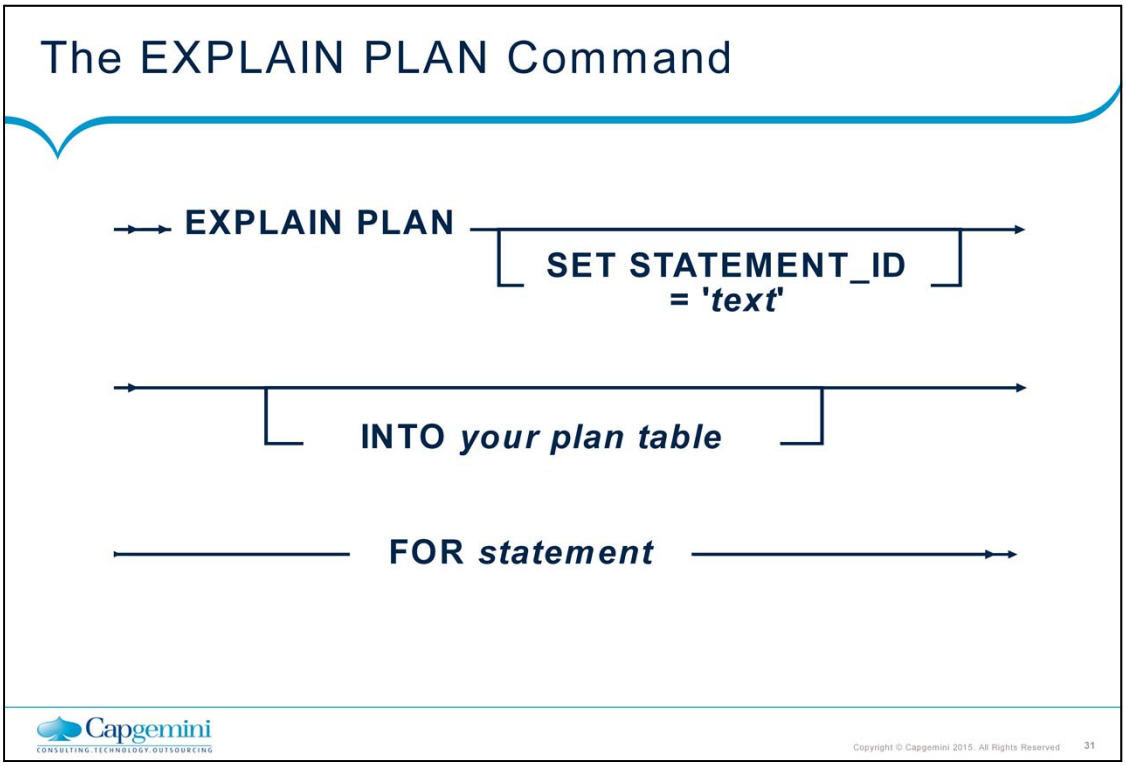
Displaying from PLAN_TABLE: Typical

- In the example in the slide, the EXPLAIN PLAN command inserts the execution plan of the SQL statement in PLAN_TABLE and adds the optional demo01 name tag for future reference. The DISPLAY function of the DBMS_XPLAN package can be used to format and display the last statement stored in PLAN_TABLE. You can also use the following syntax to retrieve the same result: SELECT * FROM TABLE(dbms_xplan.display('plan_table','demo01','typical',null));
- The output is the same as shown in the slide. In this example, you can substitute the name of another plan table instead of PLAN_TABLE and demo01 represents the statement ID. TYPICAL displays the most relevant information in the plan: operation ID, name and option, number of rows, bytes, and optimizer cost. The last parameter for the DISPLAY function is the one corresponding to filter_preds. This parameter represents a filter predicate or predicates to restrict the set of rows selected from the table where the plan is stored. When value is null (the default), the plan displayed corresponds to the last executed explain plan. This parameter can reference any column of the table where the plan is stored and can contain any SQL construct—for example, subquery or function calls.
- Note: Alternatively, you can run the utlxpls.sql (or utlxplp.sql for parallel queries) script (located in the ORACLE_HOME/rdbms/admin/ directory) to display the execution plan stored in PLAN_TABLE for the last statement explained. This script uses the DISPLAY table function from the DBMS_XPLAN package.



Displaying from PLAN_TABLE: ALL

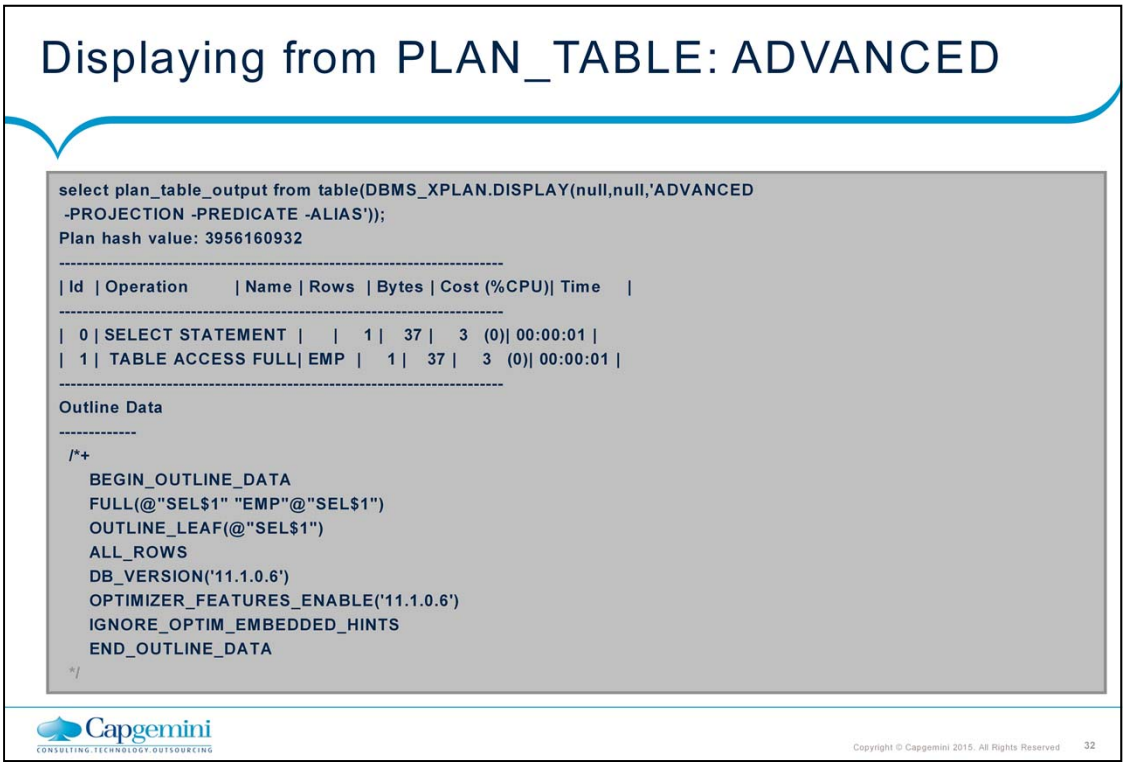
- Here you use the same EXPLAIN PLAN command example as in the previous slide. The ALL option used with the DISPLAY function allows you to output the maximum user level information. It includes information displayed with the TYPICAL level, with additional information such as PROJECTION, ALIAS, and information about REMOTE SQL, if the operation is distributed.
- For finer control on the display output, the following keywords can be added to the format parameter to customize its default behavior. Each keyword either represents a logical group of plan table columns (such as PARTITION) or logical additions to the base plan table output (such as PREDICATE). Format keywords must be separated by either a comma or a space:
  - ROWS: If relevant, shows the number of rows estimated by the optimizer
  - BYTES: If relevant, shows the number of bytes estimated by the optimizer
  - COST: If relevant, shows optimizer cost information
  - PARTITION: If relevant, shows partition pruning information
  - PARALLEL: If relevant, shows PX information (distribution method and table queue information)
  - PREDICATE: If relevant, shows the predicate section
  - PROJECTION: If relevant, shows the projection section



Displaying from PLAN_TABLE: ALL (continued)

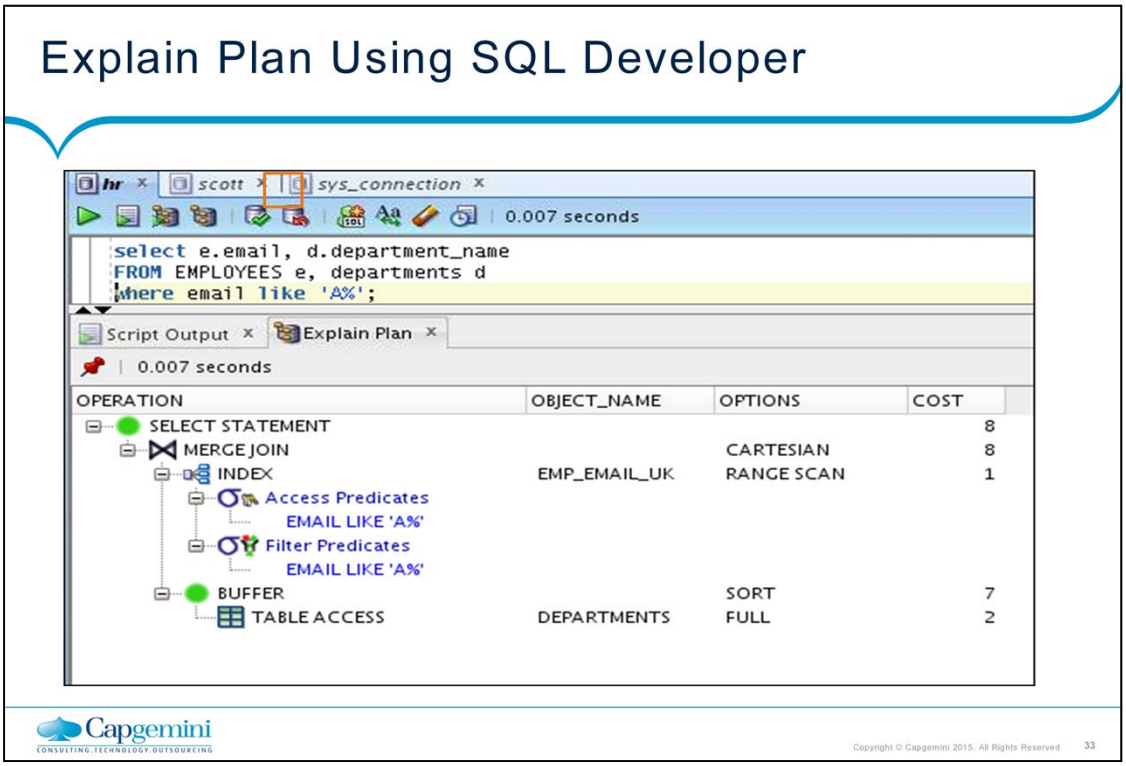
- ALIAS: If relevant, shows the “Query Block Name/Object Alias” section
- REMOTE: If relevant, shows the information for the distributed query (for example, remote from serial distribution and remote SQL)
- NOTE: If relevant, shows the note section of the explain plan

- If the target plan table also stores plan statistics columns (for example, it is a table used to capture the content of the fixed view V$SQL_PLAN_STATISTICS_ALL), additional format keywords can be used to specify which class of statistics to display when using the DISPLAY function. These additional format keywords are IOSTATS, MEMSTATS, ALLSTATS and LAST.
- Note: Format keywords can be prefixed with the “-” sign to exclude the specified information. For example, “-PROJECTION” excludes projection information.



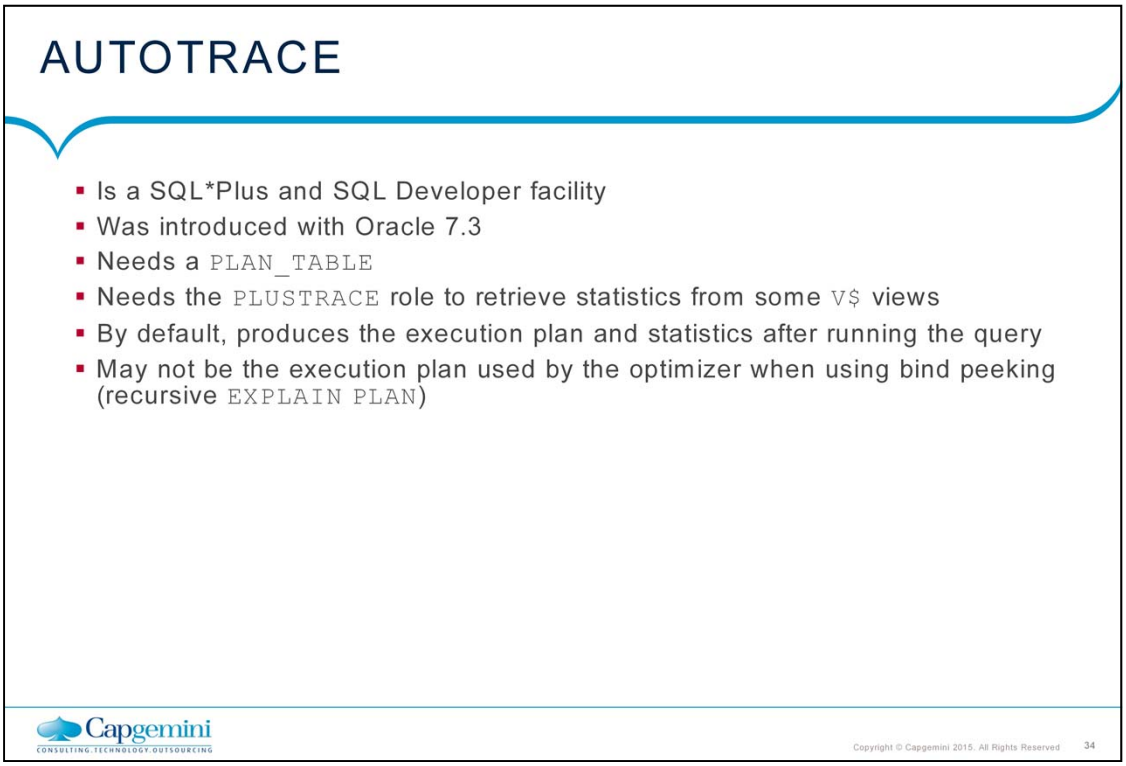
Displaying from PLAN_TABLE: ADVANCED

- The ADVANCED format is available only from Oracle Database 10g, Release 2 and later versions.
- This output format includes all sections from the ALL format plus the outline data that represents a set of hints to reproduce that particular plan.
- This section may be useful if you want to reproduce a particular execution plan in a different environment.
- This is the same section, which is displayed in the trace file for event 10053.
- Note: When the ADVANCED format is used with V$SQL_PLAN, there is one more section called Peeked Binds (identified by position).



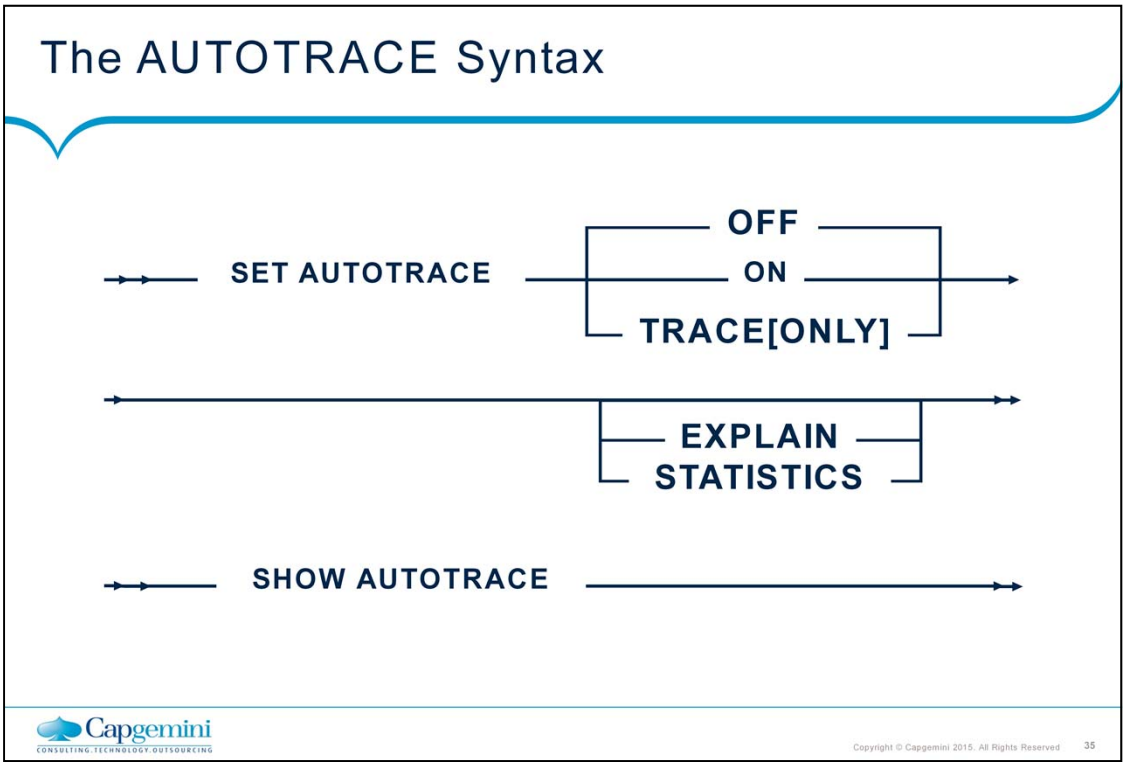
Explain Plan Using SQL Developer

- The Explain Plan icon generates the execution plan, which you can see in the Explain tab. An execution plan shows a row source tree with the hierarchy of operations that make up the statement. For each operation, it shows the ordering of the tables referenced by the statement, access method for each table mentioned in the statement, join method for tables affected by join operations in the statement, and data operations such as filter, sort, or aggregation. In addition to the row source tree, the plan table displays information about optimization (such as the cost and cardinality of each operation), partitioning (such as the set of accessed partitions), and parallel execution (such as the distribution method of join inputs).



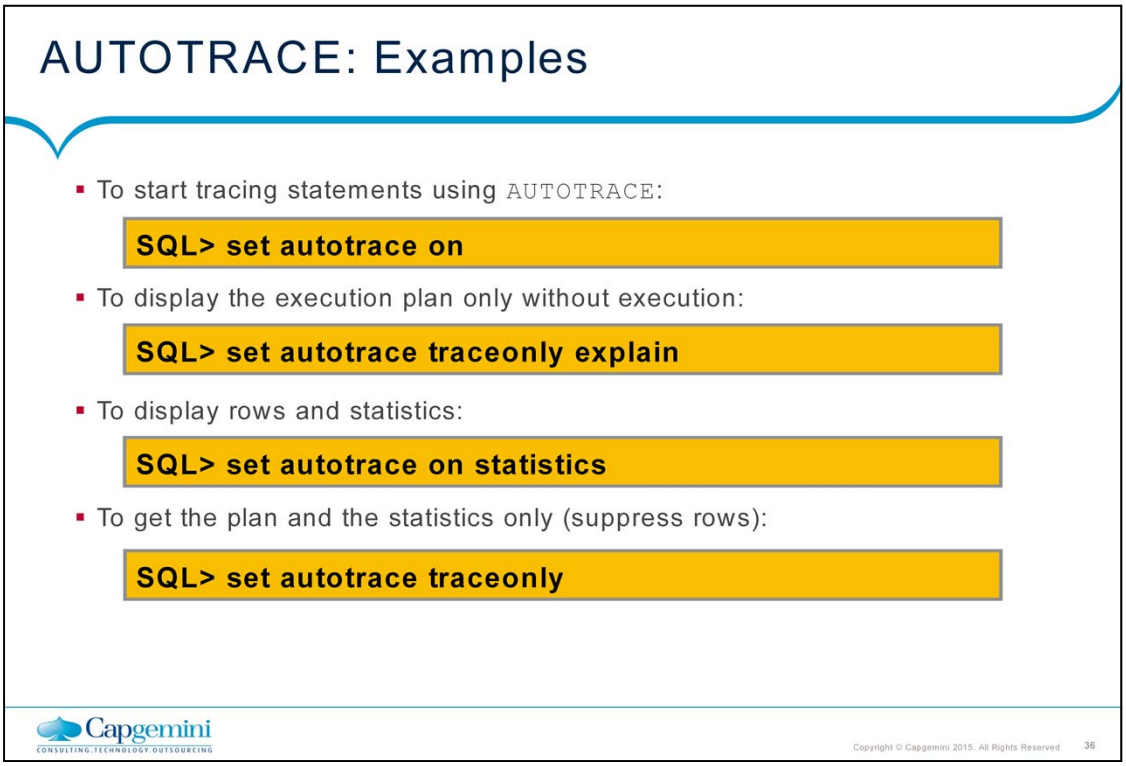
AUTOTRACE

- When running SQL statements under SQL*Plus or SQL Developer, you can automatically get a report on the execution plan and the statement execution statistics. The report is generated after successful SQL DML (that is, SELECT, DELETE, UPDATE, and INSERT) statements. It is useful for monitoring and tuning the performance of these statements.
- To use this feature, you must have a PLAN_TABLE available in your schema, and then have the PLUSTRACE role granted to you. The database administrator (DBA) privileges are required to grant the PLUSTRACE role. The PLUSTRACE role is created and granted to the DBA role by running the supplied $ORACLE_HOME/sqlplus/admin/plustrce.sql script.
- On some versions and platforms, this is run by the database creation scripts. If this is not the case on your platform, connect as SYSDBA and run the plustrce.sql script.
- The PLUSTRACE role contains the select privilege on three V$ views. These privileges are necessary to generate AUTOTRACE statistics.
- AUTOTRACE is an excellent diagnostic tool for SQL statement tuning. Because it is purely declarative, it is easier to use than EXPLAIN PLAN.
- Note: The system does not support EXPLAIN PLAN for statements performing implicit type conversion of date bind variables. With bind variables in general, the EXPLAIN PLAN output might not represent the real execution plan.



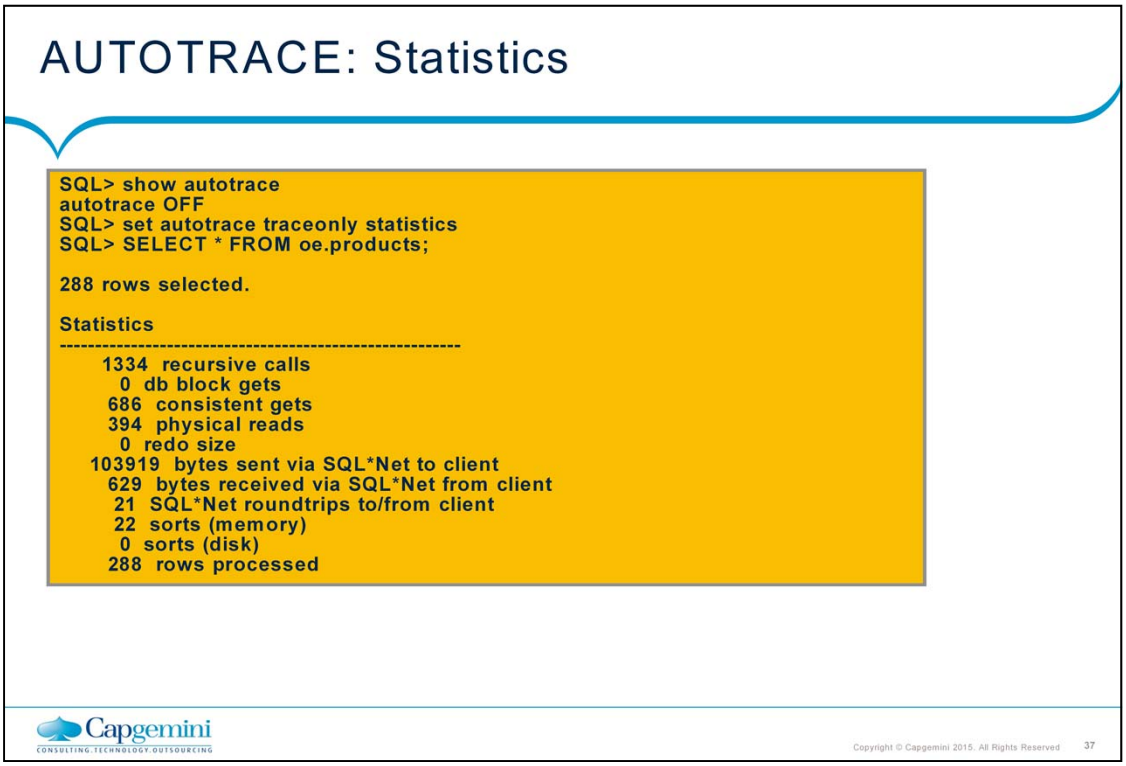
The AUTOTRACE Syntax

- You can enable AUTOTRACE in various ways using the syntax shown in the slide. The command options are as follows:
  - OFF: Disables autotracing SQL statements
  - ON: Enables autotracing SQL statements
  - TRACE or TRACE[ONLY]: Enables autotracing SQL statements and suppresses statement output
  - EXPLAIN: Displays execution plans, but does not display statistics
  - STATISTICS: Displays statistics, but does not display execution plans
- Note: If both the EXPLAIN and STATISTICS command options are omitted, execution plans and statistics are displayed by default.



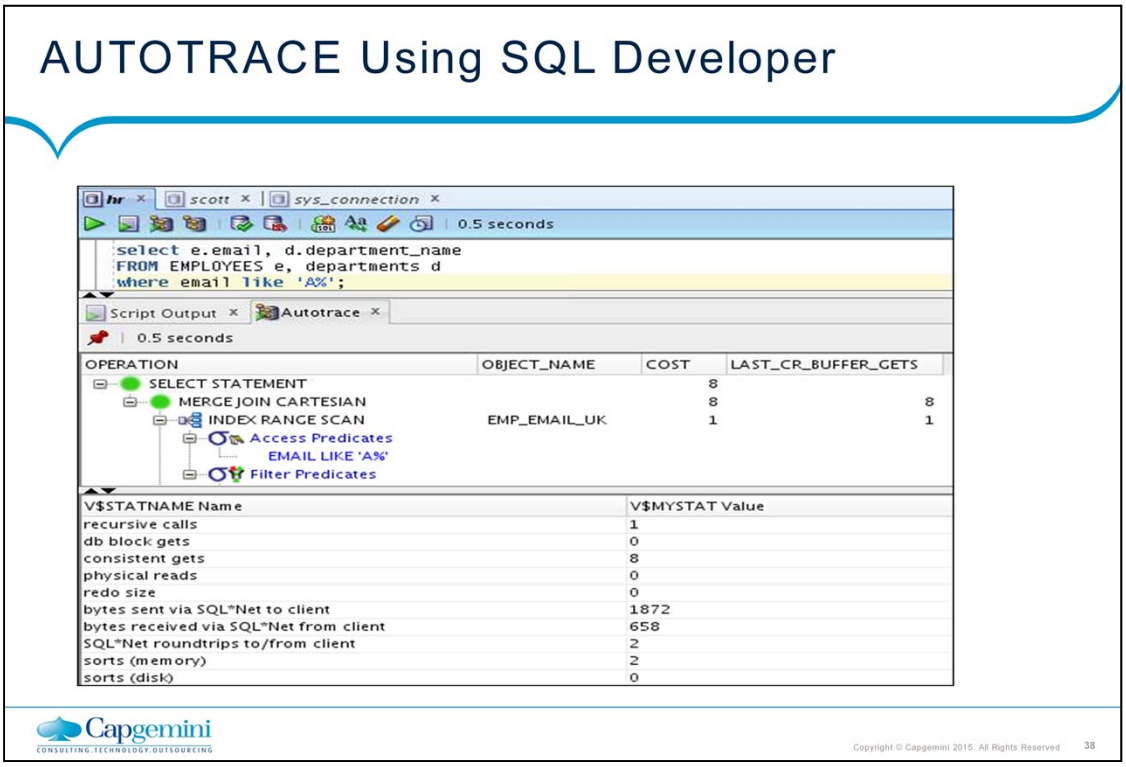
AUTOTRACE: Examples

- You can control the report by setting the AUTOTRACE system variable. The following are some examples:
  - SET AUTOTRACE ON: The AUTOTRACE report includes both the optimizer execution plan and the SQL statement execution statistics.
  - SET AUTOTRACE TRACEONLY EXPLAIN: The AUTOTRACE report shows only the optimizer execution path without executing the statement.
  - SET AUTOTRACE ON STATISTICS: The AUTOTRACE report shows the SQL statement execution statistics and rows.
  - SET AUTOTRACE TRACEONLY: This is similar to SET AUTOTRACE ON, but it suppresses the printing of the user's query output, if any. If STATISTICS is enabled, the query data is still fetched, but not printed.
  - SET AUTOTRACE OFF: No AUTOTRACE report is generated. This is the default.



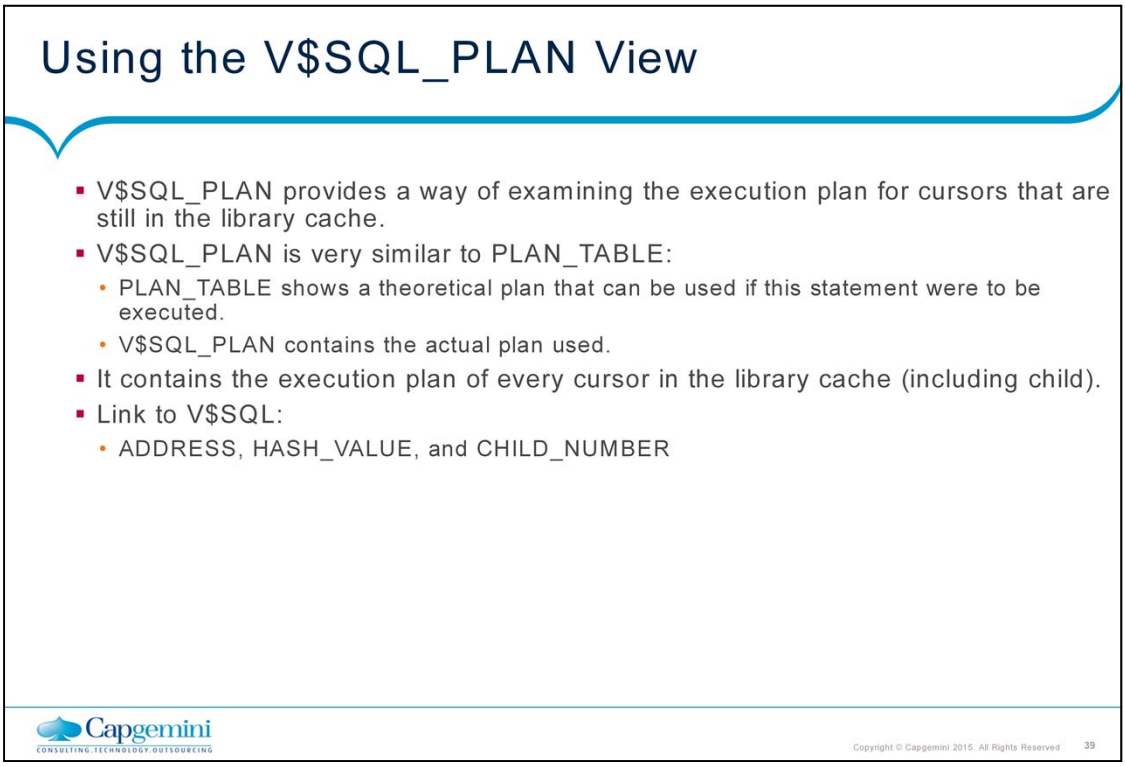
AUTOTRACE: Statistics

- The statistics are recorded by the server when your statement executes and indicate the system resources required to execute your statement. The results include the following statistics:
  - recursive calls is the number of recursive calls generated at both the user and system level. Oracle Database maintains tables used for internal processing. When Oracle Database needs to make a change to these tables, it internally generates an internal SQL statement, which in turn generates a recursive call.
  - db block gets is the number of times a CURRENT block was requested.
  - consistent gets is the number of times a consistent read was requested for a block.
  - physical reads is the total number of data blocks read from disk. This number equals the value of “physical reads direct” plus all reads into buffer cache.
  - redo size is the total amount of redo generated in bytes.
  - bytes sent via SQL*Net to client is the total number of bytes sent to the client from the foreground processes.
  - bytes received via SQL*Net from client is the total number of bytes received from the client over Oracle Net.



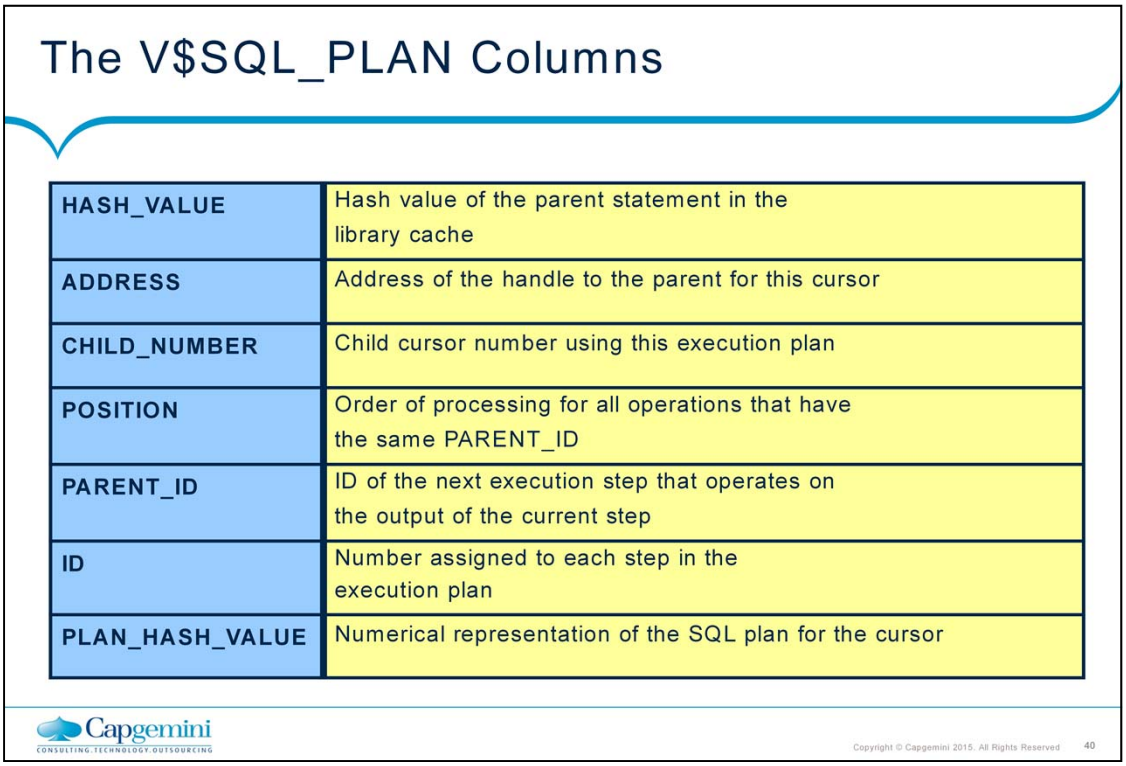
AUTOTRACE Using SQL Developer

- The Autotrace pane displays trace-related information when you execute the SQL statement by clicking the Autotrace icon. This information can help you to identify SQL statements that will benefit from tuning.



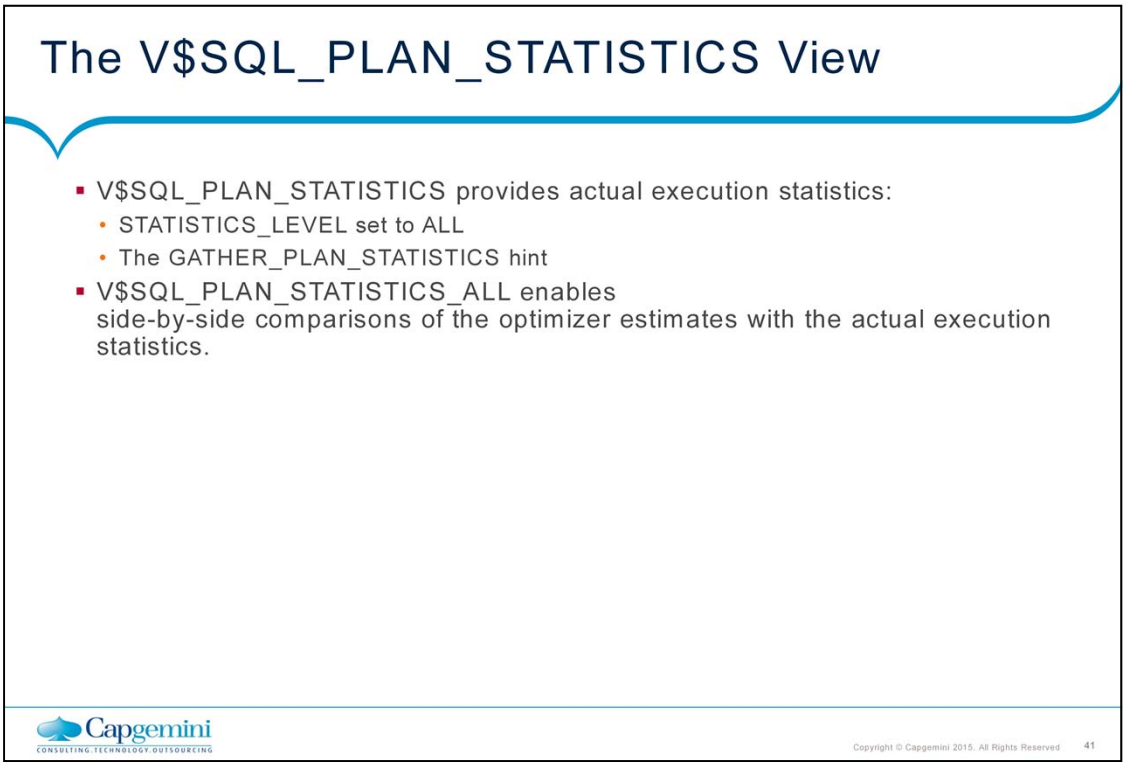
Using the V$SQL_PLAN View

- This view displays the execution plan for cursors that are still in the library cache. The information in this view is very similar to the information in PLAN_TABLE. However, V$SQL_PLAN contains the actual plan used. The execution plan obtained by the EXPLAIN PLAN statement can be different from the execution plan used to execute the cursor. This is because the cursor might have been compiled with different values of session parameters or bind variables..
- V$SQL_PLAN shows the plan for a cursor rather than for all cursors associated with a SQL statement. The difference is that a SQL statement can have more than one cursor associated with it, with each cursor further identified by a CHILD_NUMBER. For example, the same statement executed by different users has different cursors associated with it if the object that is referenced is in a different schema. Similarly, different hints can cause different cursors. The V$SQL_PLAN table can be used to see the different plans for different child cursors of the same statement.
- Note: Another useful view is V$SQL_PLAN_STATISTICS, which provides the execution statistics of each operation in the execution plan for each cached cursor. Also, the V$SQL_PLAN_STATISTICS_ALL view concatenates information from V$SQL_PLAN with execution statistics from V$SQL_PLAN_STATISTICS and V$SQL_WORKAREA.



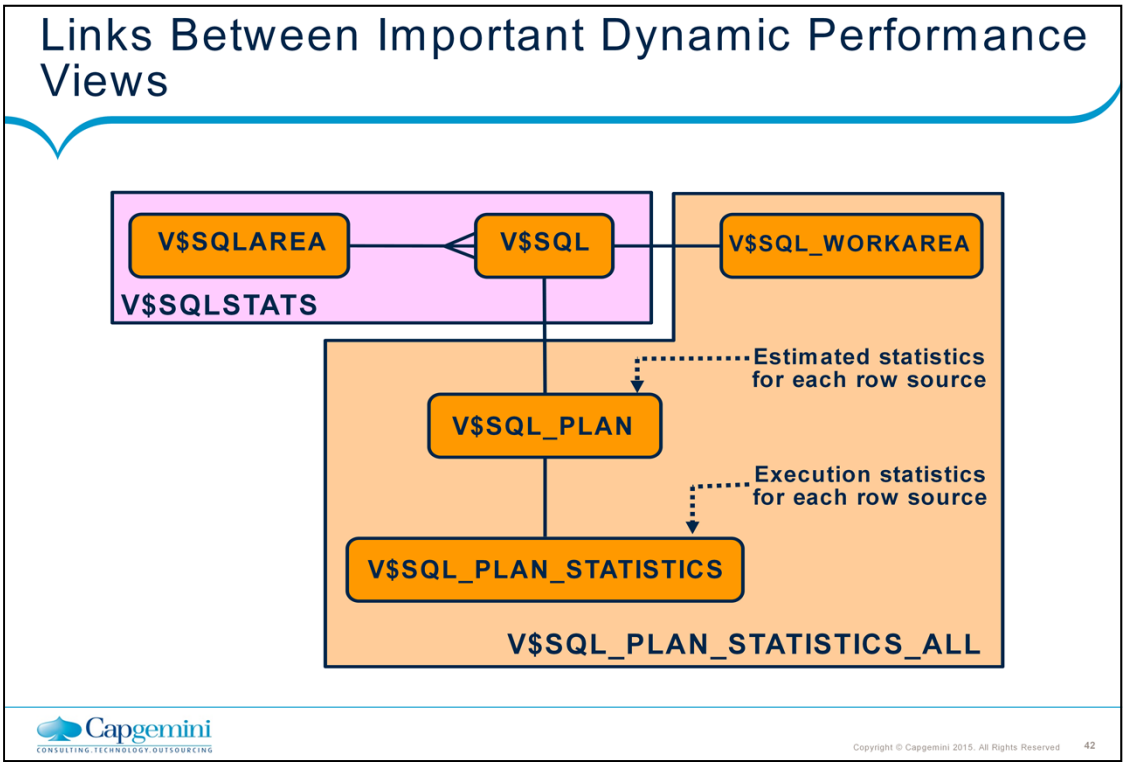
The V$SQL_PLAN Columns

- The view contains many of the PLAN_TABLE columns, plus several others. The columns that are also present in PLAN_TABLE have the same values:
  - ADDRESS
  - HASH_VALUE
- The ADDRESS and HASH_VALUE columns can be used to join with V$SQLAREA to add the cursor-specific information.
- The ADDRESS, HASH_VALUE, and CHILD_NUMBER columns can be used to join with V$SQL to add the child cursor–specific information.
- The PLAN_HASH VALUE column is a numerical representation of the SQL plan for the cursor. By comparing one PLAN_HASH_VALUE with another, you can easily identify whether the two plans are the same or not (rather than comparing the two plans line-by-line).
- Note: Since Oracle Database 10g, SQL_HASH_VALUE in V$SESSION has been complemented with SQL_ID, which you retrieve in many other V$ views. SQL_HASH_VALUE is a 32-bit value and is not unique enough for large repositories of AWR data. SQL_ID is a 64-bit hash value, which is more unique, the bottom 32 bits of which are SQL_HASH_VALUE. It is normally represented as a character string to make it more manageable.



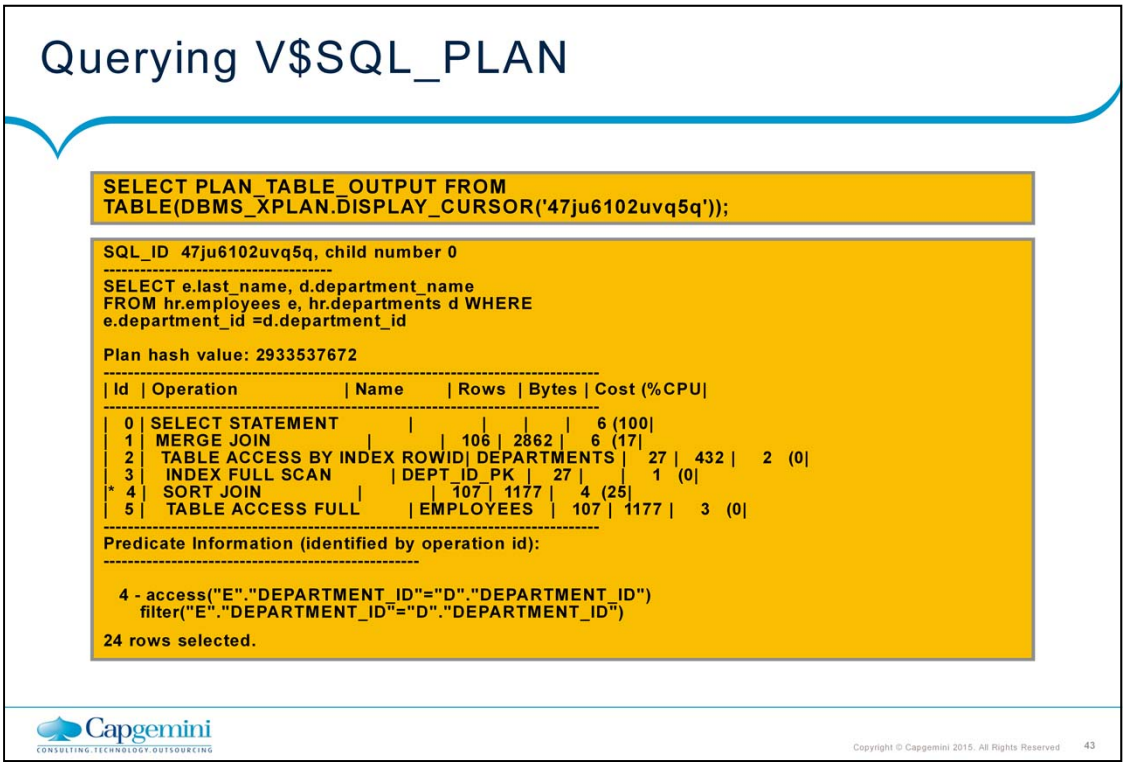
The V$SQL_PLAN_STATISTICS View

- The V$SQL_PLAN_STATISTICS view provides the actual execution statistics for every operation in the plan, such as the number of output rows, and elapsed time. All statistics, except the number of output rows, are cumulative. For example, the statistics for a join operation also include the statistics for its two inputs. The statistics in V$SQL_PLAN_STATISTICS are available for cursors that have been compiled with the STATISTICS_LEVEL initialization parameter set to ALL or using the GATHER_PLAN_STATISTICS hint.
- The V$SQL_PLAN_STATISTICS_ALL view contains memory-usage statistics for row sources that use SQL memory (sort or hash join). This view concatenates information in V$SQL_PLAN with execution statistics from V$SQL_PLAN_STATISTICS and V$SQL_WORKAREA.



Links Between Important Dynamic Performance Views

- V$SQLAREA displays statistics on shared SQL areas and contains one row per SQL string. It provides statistics on SQL statements that are in memory, parsed, and ready for execution:
  - SQL_ID is the SQL identifier of the parent cursor in the library cache.
  - VERSION_COUNT is the number of child cursors that are present in the cache under this parent.
- V$SQL lists statistics on shared SQL areas and contains one row for each child of the original SQL text entered:
  - ADDRESS represents the address of the handle to the parent for this cursor.
  - HASH_VALUE is the value of the parent statement in the library cache.
  - SQL_ID is the SQL identifier of the parent cursor in the library cache.
  - PLAN_HASH_VALUE is a numeric representation of the SQL plan for this cursor. By comparing one PLAN_HASH_VALUE with another, you can easily identify if the two plans are the same or not (rather than comparing the two plans line-by-line).
  - CHILD_NUMBER is the number of this child cursor.
- Statistics displayed in V$SQL are normally updated at the end of query execution. However, for long-running queries, they are updated every five seconds. This makes it easy to see the impact of long-running SQL statements while they are still in progress.



Querying V$SQL_PLAN

- You can query V$SQL_PLAN using the DBMS_XPLAN.DISPLAY_CURSOR() function to display the current or last executed statement (as shown in the example). You can pass the value of SQL_ID for the statement as a parameter to obtain the execution plan for a given statement. SQL_ID is the SQL_ID of the SQL statement in the cursor cache. You can retrieve the appropriate value by querying the SQL_ID column in V$SQL or V$SQLAREA. Alternatively, you could select the PREV_SQL_ID column for a specific session out of V$SESSION. This parameter defaults to null in which case the plan of the last cursor executed by the session is displayed. To obtain SQL_ID, execute the following query:

SELECT e.last_name, d.department_name

- FROM hr.employees e, hr.departments d
  WHERE e.department_id =d.department_id;

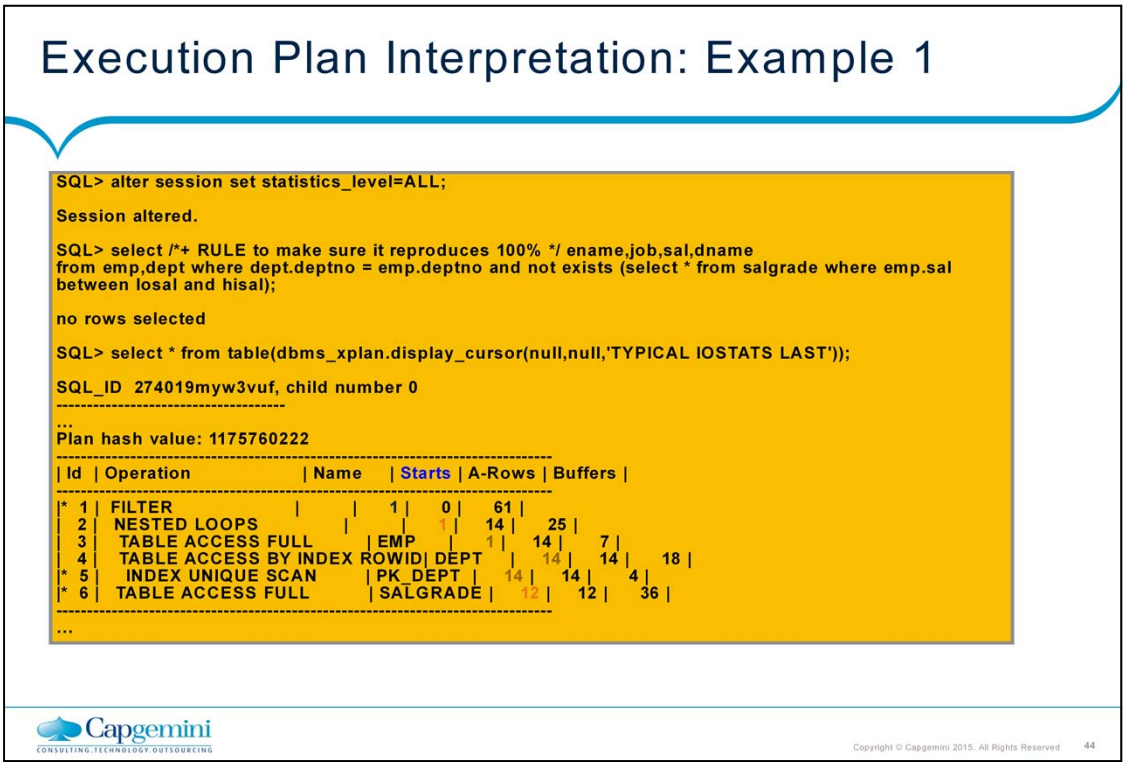
- SELECT SQL_ID, SQL_TEXT FROM V$SQL

WHERE SQL_TEXT LIKE '%SELECT e.last_name,%' ;

13saxr0mmz1s3 select SQL_id, sql_text from v$SQL …

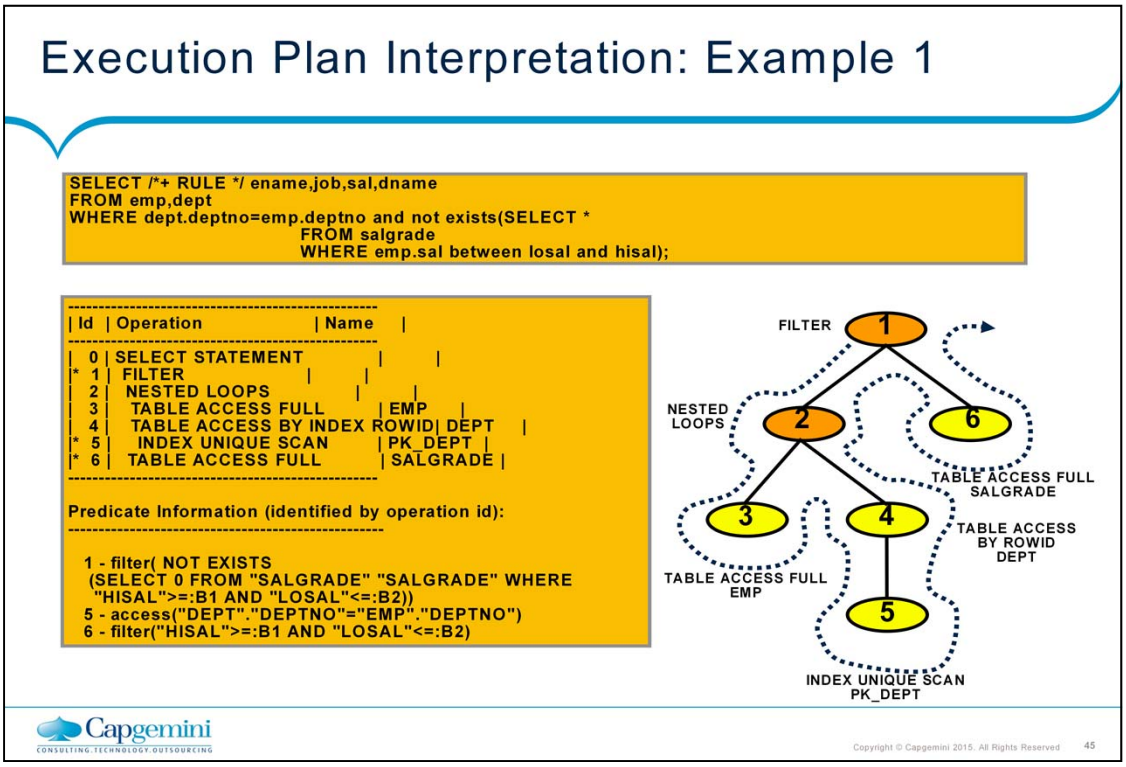
47ju6102uvq5q SELECT e.last_name, d.department_name

…



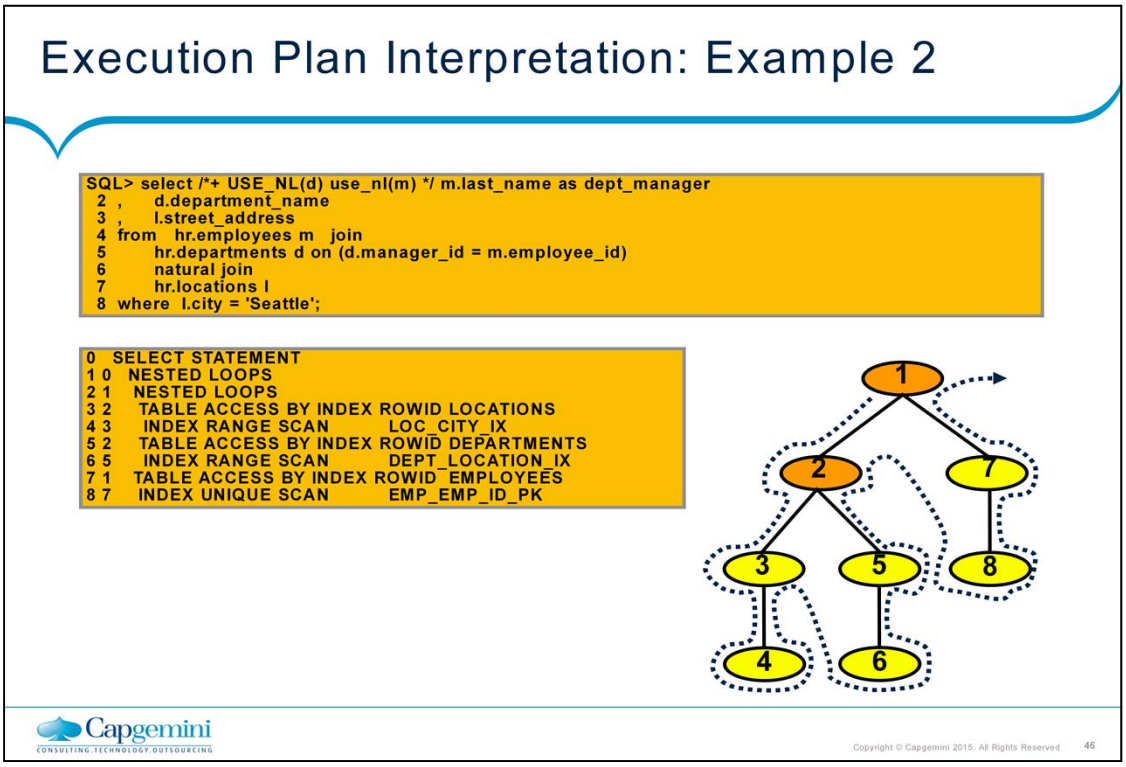
Execution Plan Interpretation: Example 1 (continued)

- The example in the slide is a plan dump from V$SQL_PLAN with STATISTICS_LEVEL set to ALL. This report shows you some important additional information compared to the output of the EXPLAIN PLAN command:
  - A-Rows corresponds to the number of rows produced by the corresponding row source.
  - Buffers corresponds to the number of consistent reads done by the row source.
  - Starts indicates how many times the corresponding operation was processed.
- For each row from the EMP table, the system gets its ENAME, SAL, JOB, and DEPTNO.
- Then the system accesses the DEPT table by its unique index (PK_DEPT) to get DNAME using DEPTNO from the previous result set.
- If you observe the statistics closely, the TABLE ACCESS FULL operation on the EMP table (ID=3) is started once. However, operations from ID 5 and 4 are started 14 times; once for each EMP rows. At this step (ID=2), the system gets all ENAME, SAL, JOB, and DNAME.
- The system now must filter out employees who have salaries outside the range of salaries in the salary grade table. To do that, for each row from ID=2, the system accesses the SALGRADE table using a FULL TABLE SCAN operation to check if the employee's salary is outside the salary range. This operation only needs to be done 12 times in this case because at run time the system does the check for each distinct salary, and there are 12 distinct salaries in the EMP table.



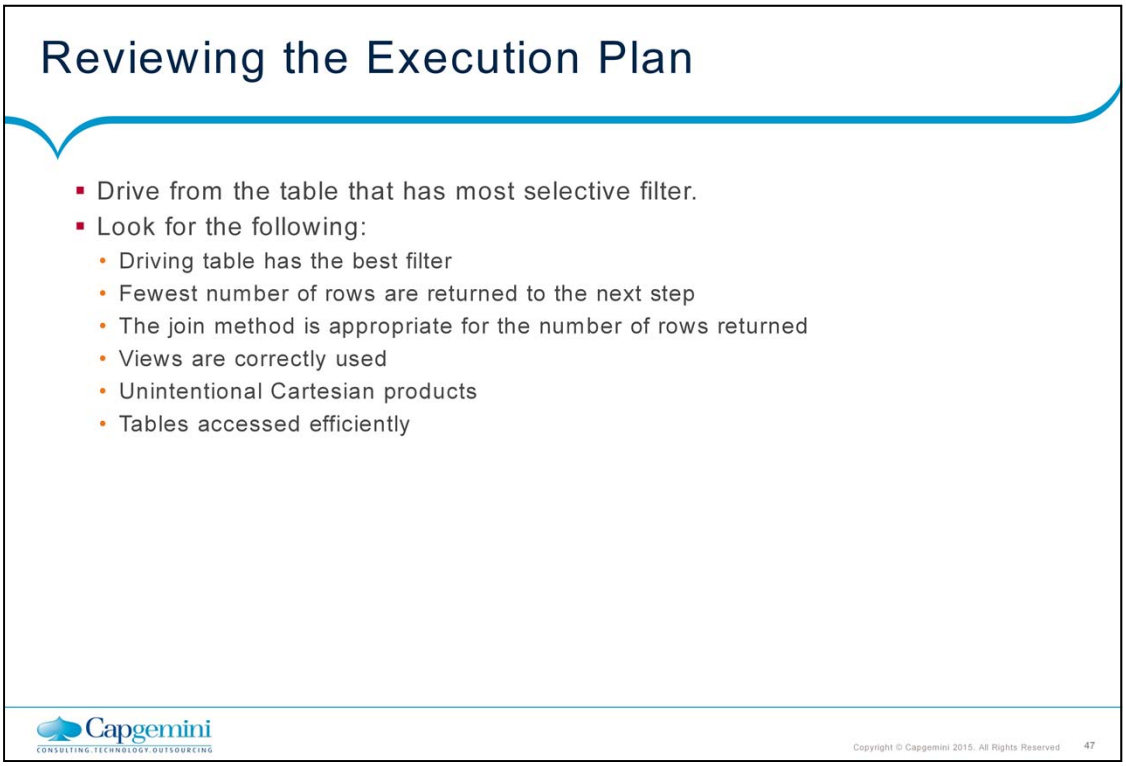
Execution Plan Interpretation: Example 1

- You start with an example query to illustrate how to interpret an execution plan. The slide shows a query with its associated execution plan and the same plan in the tree format.
- The query tries to find employees who have salaries outside the range of salaries in the salary grade table. The query is a SELECT statement from two tables with a subquery based on another table to check the salary grades.
- See the execution order for this query. Based on the example in the slide, and from the previous slide, the execution order is 3 – 5 – 4 – 2 – 6 – 1:
  - 3: The plan starts with a full table scan of EMP (ID=3).
  - 5: The rows are passed back to the controlling nested loops join step (ID=2), which uses them to execute the lookup of rows in the PK_DEPT index in ID=5.
  - 4: The ROWIDs from the index are used to lookup the other information from the DEPT table in ID=4.
  - 2: ID=2, the nested loops join step, is executed until completion.
  - 6: After ID=2 has exhausted its row sources, a full table scan of SALGRADE in ID=6 (at the same level in the tree as ID=2, therefore, its sibling) is executed.
  - 1: This is used to filter the rows from ID2 and ID6.



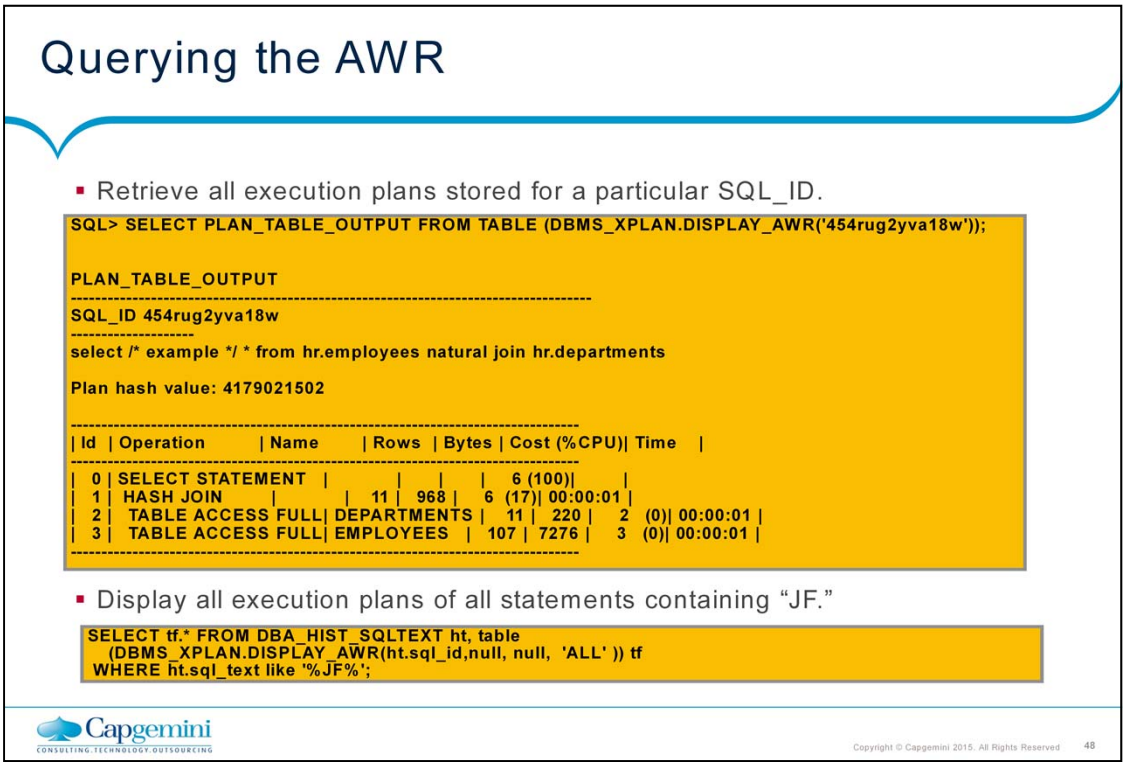
Execution Plan Interpretation: Example 2

- This query retrieves names, department names, and addresses for employees whose departments are located in Seattle and who have managers.
- For formatting reasons, the explain plan has the ID in the first column, and PID in the second column. The position is reflected by the indentation. The execution plan shows two nested loops join operations.
- You follow the steps from the previous example:
  - 1. Start at the top. ID=0
  - 2. Move down the row sources until you get to the one, which produces data, but does not consume any. In this case, ID 0, 1, 2, and 3 consume data. ID=4 is the first row source that does not consume any. This is the start row source. ID=4 is executed first. The index range scan produces ROWIDs, which are used to lookup in the LOCATIONS table in ID=3.
  - 3. Look at the siblings of this row source. These row sources are executed next. The sibling at the same level as ID=3 is ID=5. Node ID=5 has a child ID=6, which is executed before it. This is another index range scan producing ROWIDs, which are used to lookup in the DEPARTMENTS table in ID=5.



Reviewing the Execution Plan

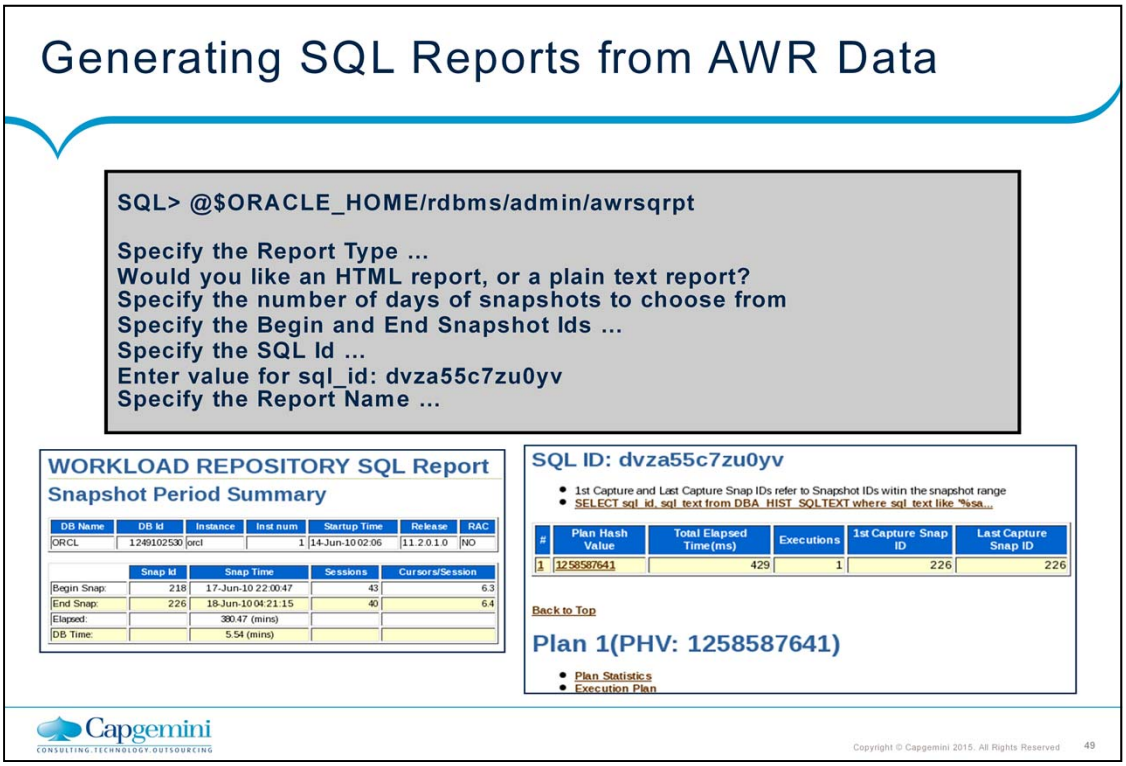
- When you tune a SQL statement in an online transaction processing (OLTP) environment, the goal is to drive from the table that has the most selective filter. This means that there are fewer rows passed to the next step. If the next step is a join, this means fewer rows are joined. Check to see whether the access paths are optimal. When you examine the optimizer execution plan, look for the following:
  - The plan is such that the driving table has the best filter.
  - The join order in each step means that the fewest number of rows are returned to the next step (that is, the join order should reflect going to the best not-yet-used filters).
  - The join method is appropriate for the number of rows being returned. For example, nested loop joins through indexes may not be optimal when many rows are returned.
  - Views are used efficiently. Look at the SELECT list to see whether access to the view is necessary.
  - There are any unintentional Cartesian products (even with small tables).
  - Each table is being accessed efficiently: Consider the predicates in the SQL statement and the number of rows in the table. Look for suspicious activity, such as a full table scans on tables with large number of rows, which have predicates in the WHERE clause. Also, a full table scan might be more efficient on a small table, or to leverage a better join method (for example, hash join) for the number of rows returned.
- If any of these conditions are not optimal, consider restructuring the SQL statement or the indexes available on the tables.



Querying the AWR

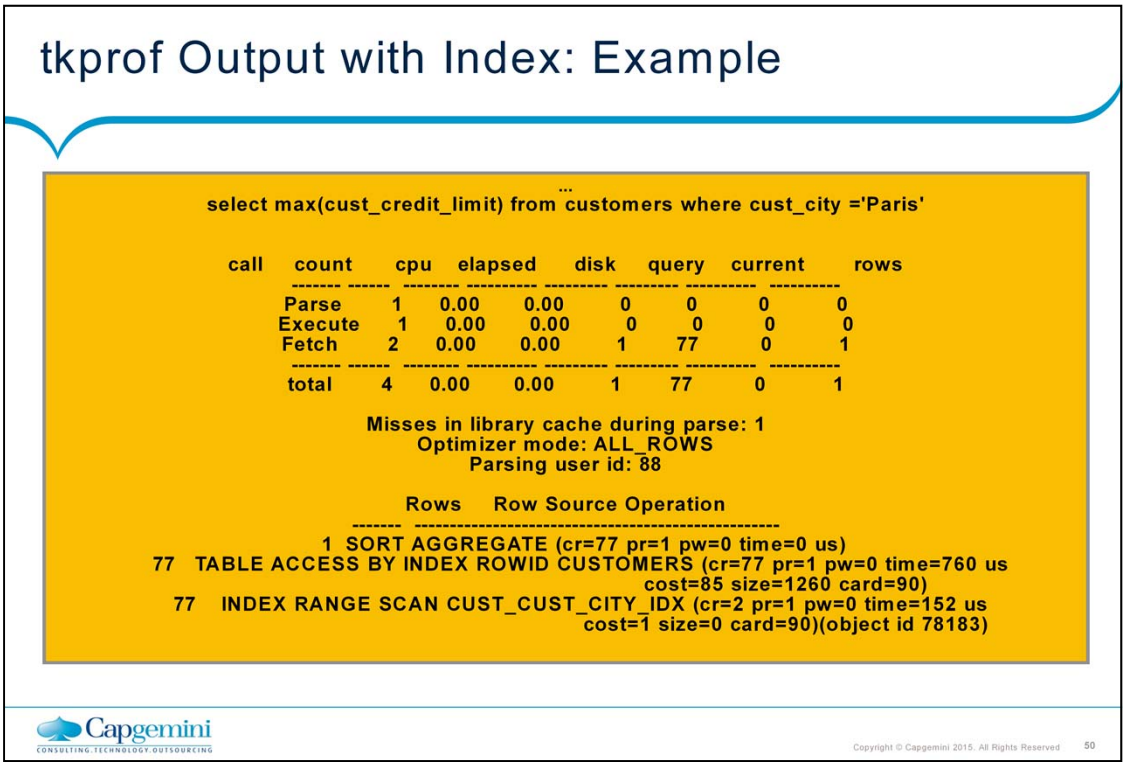
- You can use the DBMS_XPLAN.DISPLAY_AWR() function to display all stored plans in the AWR. In the example in the slide, you pass in a SQL_ID as an argument. SQL_ID is the SQL_ID of the SQL statement in the cursor cache. The DISPLAY_AWR() function also takes the PLAN_HASH_VALUE, DB_ID, and FORMAT parameters.
- The steps to complete this example are as follows:
  - 1. Execute the SQL statement:
  - SQL> select /* example */ * from hr.employees natural join hr.departments;
  - 2. Query V$SQL_TEXT to obtain the SQL_ID:

```
SQL> select sql_id, sql_text from v$SQL
     where sql_text
     like '%example%';

SQL_ID        SQL_TEXT
------------- ----------------------------------------
F8tc4anpz5cdb select sql_id, sql_text from v$SQL …
454rug2yva18w select /* example */ * from …
```



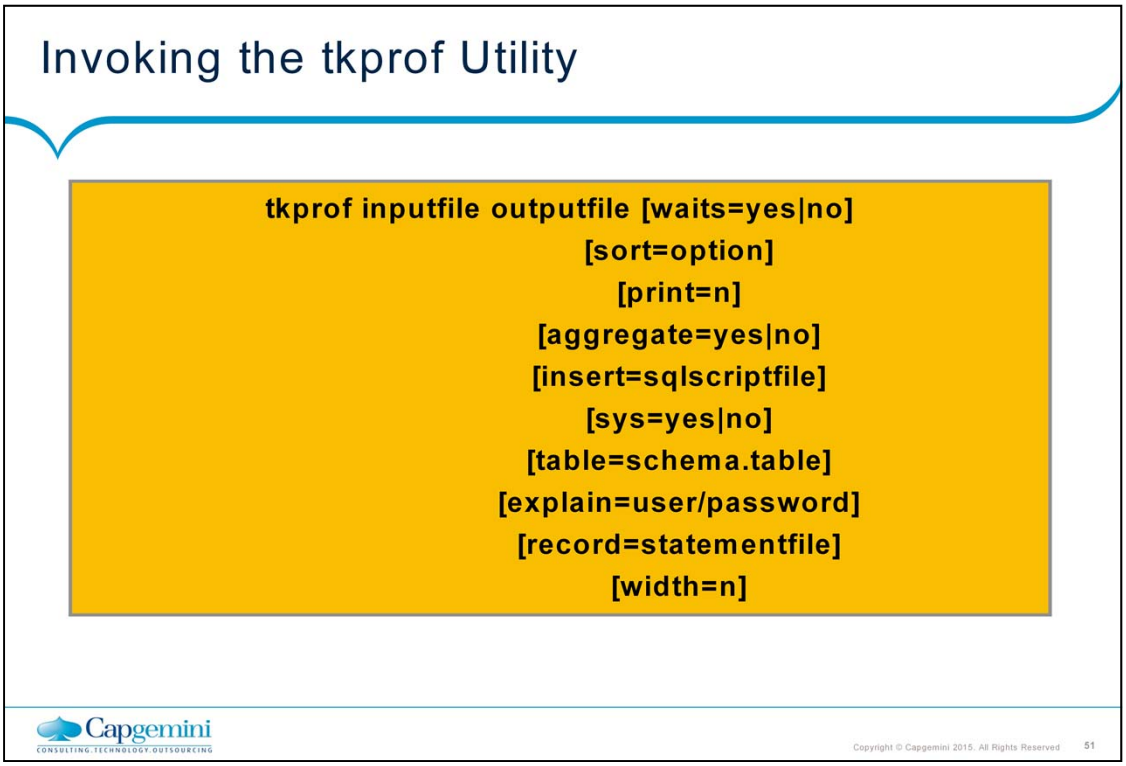
Generating SQL Reports from AWR Data

- Since Oracle Database 10g, Release 2, it is possible to generate SQL reports from AWR data, basically, the equivalent to sqrepsql.sql with Statspack. In 10.1.0.4.0, the equivalent to sprepsql.sql is not available in AWR. However, in 10gR2, the equivalent of sprepsql.sql is available. In 10gR2, the AWR SQL report can be generated by calling the $ORACLE_HOME/rdbms/admin/awrsqrpt.sql file.
- You can display the plan information in AWR by using the display_awr table function in the dbms_xplan PL/SQL package.
- For example, this displays the plan information for a SQL_ID in AWR:
  select * from table(dbms_xplan.display_awr('dvza55c7zu0yv'));
- You can retrieve the appropriate value for the SQL statement of interest by querying SQL_ID in the DBA_HIST_SQLTEXT column.



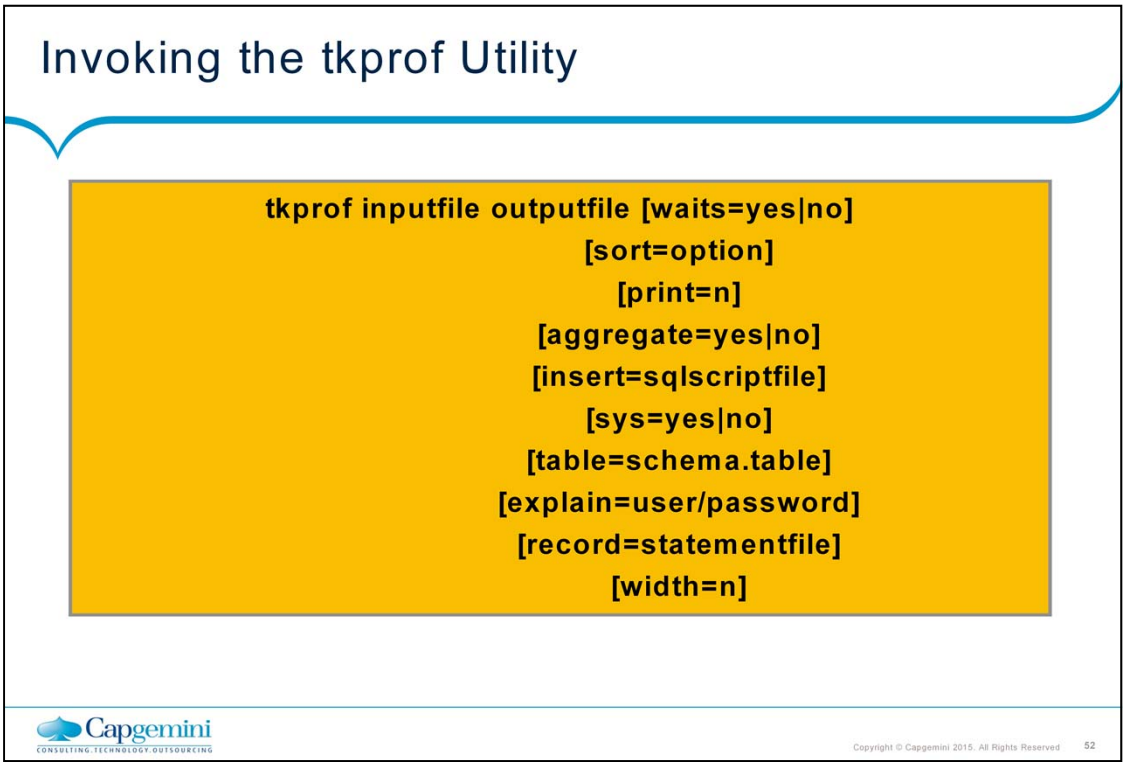
tkprof Output with Index: Example

- The results shown in the slide indicate that CPU time was reduced to 0.01 second when an index was created on the CUST_CITY column. These results may have been achieved because the statement uses the index to retrieve the data. Additionally, because this example reexecutes the same statement, most of the data blocks are already in memory. You can achieve significant improvements in performance by indexing sensibly. Identify areas for potential improvement using the SQL Trace facility.
- Note: Indexes should not be built unless required. Indexes do slow down the processing of the INSERT, UPDATE, and DELETE commands because references to rows must be added, changed, or removed. Unused indexes should be removed. However, instead of processing all the application SQL through EXPLAIN PLAN, you can use index monitoring to identify and remove any indexes that are not used.



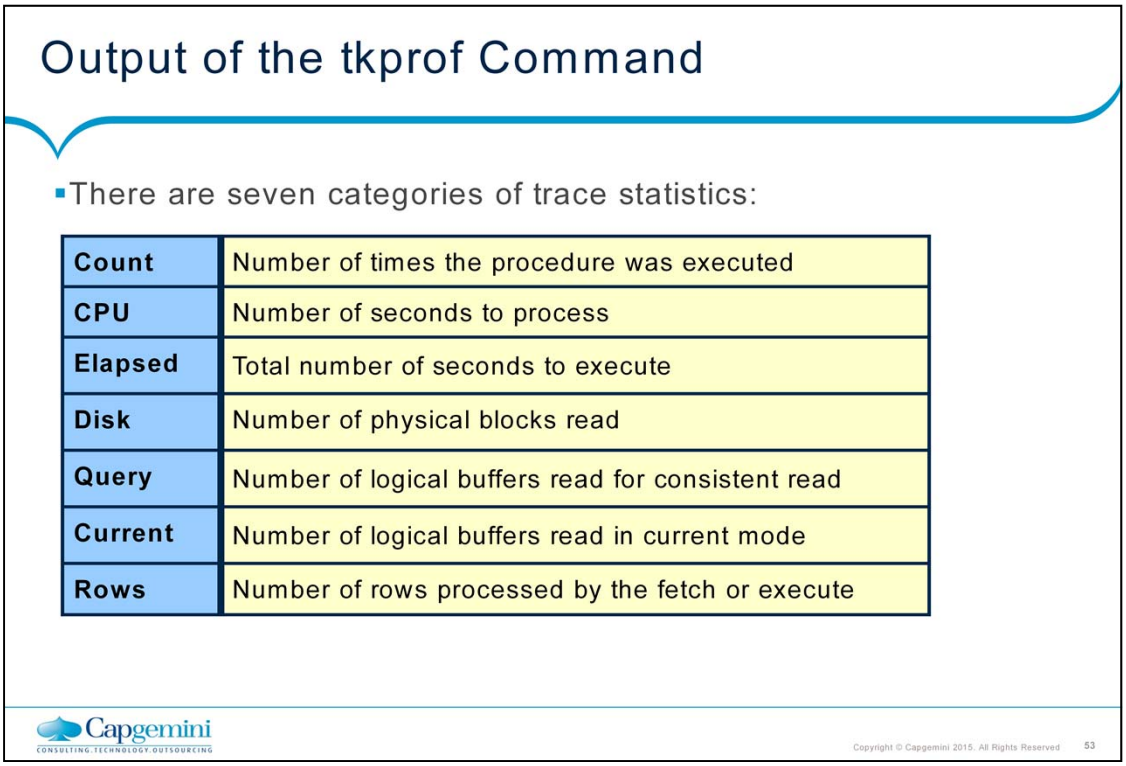
Invoking the tkprof Utility

- When you enter the tkprof command without any arguments, it generates a usage message together with a description of all tkprof options. The various arguments are shown in the slide:
  - inputfile: Specifies the SQL trace input file
  - outputfile: Specifies the file to which tkprof writes its formatted output
  - waits: Specifies whether to record the summary for any wait events found in the trace file. Values are YES or NO. The default is YES.
  - sorts: Sorts traced SQL statements in the descending order of specified sort option before listing them into the output file. If more than one option is specified, the output is sorted in the descending order by the sum of the values specified in the sort options. If you omit this parameter, tkprof lists statements into the output file in the order of first use.
  - print: Lists only the first integer sorted SQL statements from the output file. If you omit this parameter, tkprof lists all traced SQL statements. This parameter does not affect the optional SQL script. The SQL script always generates insert data for all traced SQL statements.
  - aggregate: If set to NO, tkprof does not aggregate multiple users of the same SQL text.



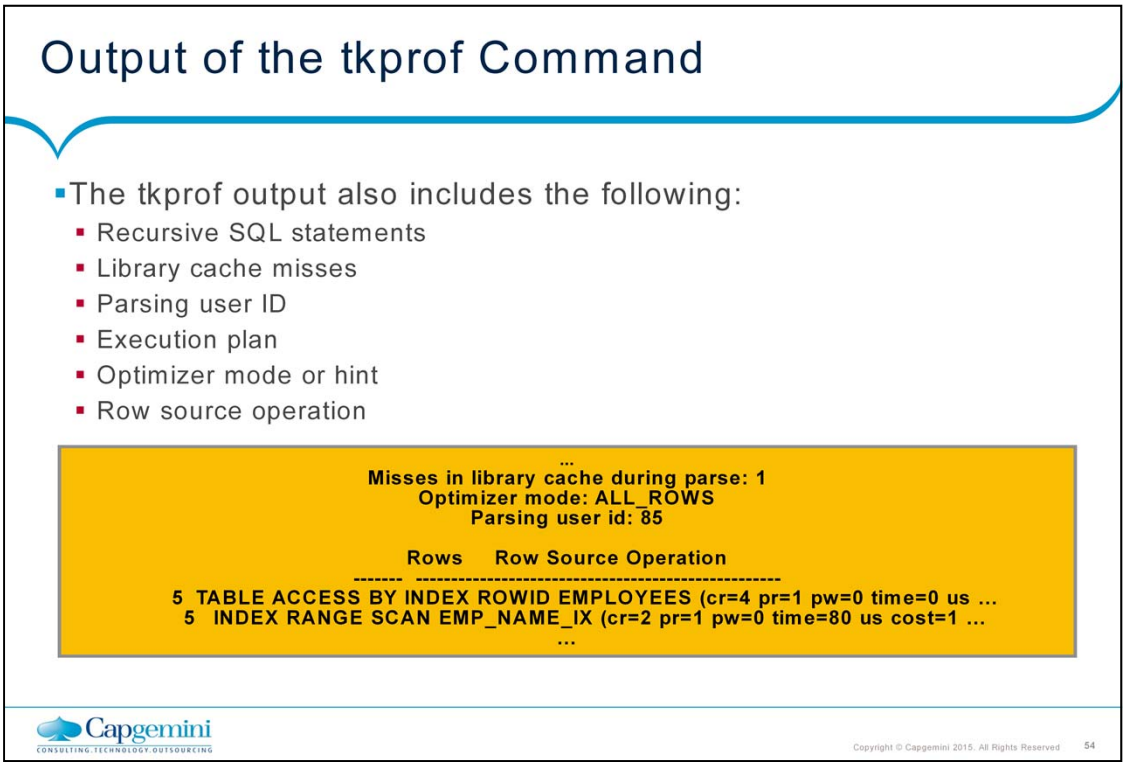
Invoking the tkprof Utility

- When you enter the tkprof command without any arguments, it generates a usage message together with a description of all tkprof options. The various arguments are shown in the slide:
  - inputfile: Specifies the SQL trace input file
  - outputfile: Specifies the file to which tkprof writes its formatted output
  - waits: Specifies whether to record the summary for any wait events found in the trace file. Values are YES or NO. The default is YES.
  - sorts: Sorts traced SQL statements in the descending order of specified sort option before listing them into the output file. If more than one option is specified, the output is sorted in the descending order by the sum of the values specified in the sort options. If you omit this parameter, tkprof lists statements into the output file in the order of first use.
  - print: Lists only the first integer sorted SQL statements from the output file. If you omit this parameter, tkprof lists all traced SQL statements. This parameter does not affect the optional SQL script. The SQL script always generates insert data for all traced SQL statements.
  - aggregate: If set to NO, tkprof does not aggregate multiple users of the same SQL text.



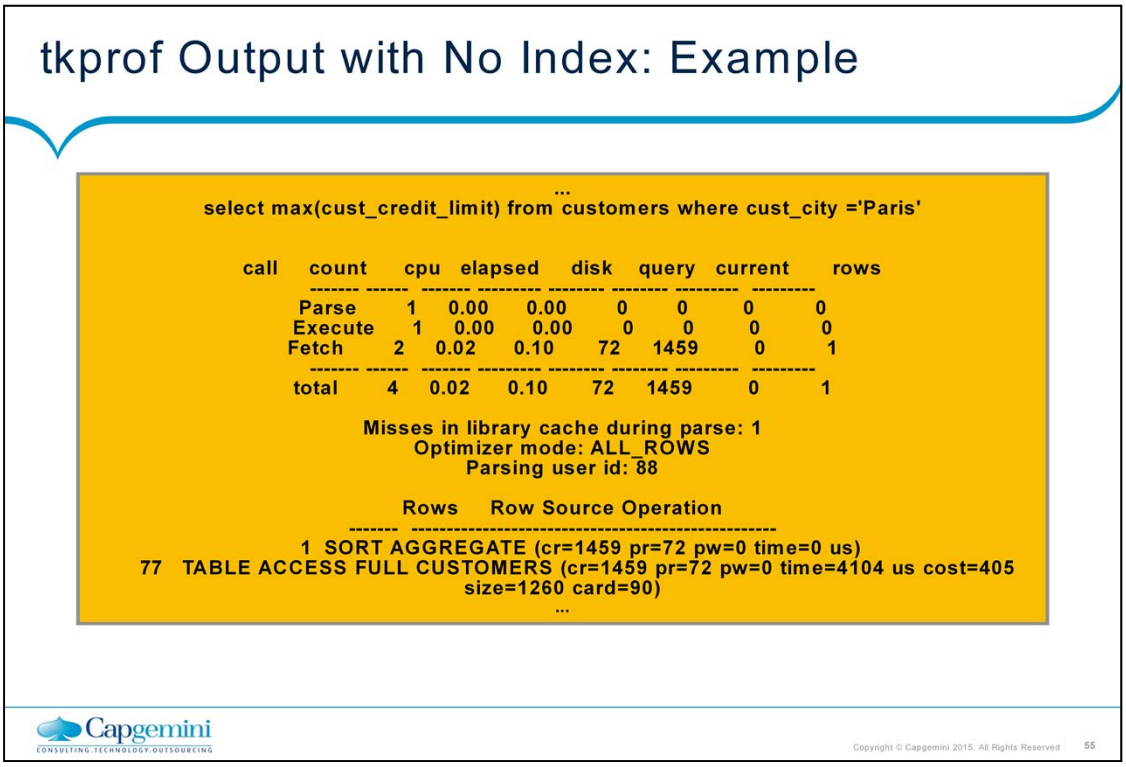
Output of the tkprof Command (continued)

- The output is explained on the following page.
- Sample output is as follows:
- call count cpu elapsed disk query current rows
  ------- ------ -------- ---------- ---------- ---------- ---------- ---
- Parse 1 0.03 0.06 0 0 0 0
  Execute 1 0.06 0.30 1 3 0 0
  Fetch 2 0.00 0.46 0 0 0 1
  ------- ------ -------- ---------- ---------- ---------- ---------- ---
- total 4 0.09 0.83 1 3 0 1



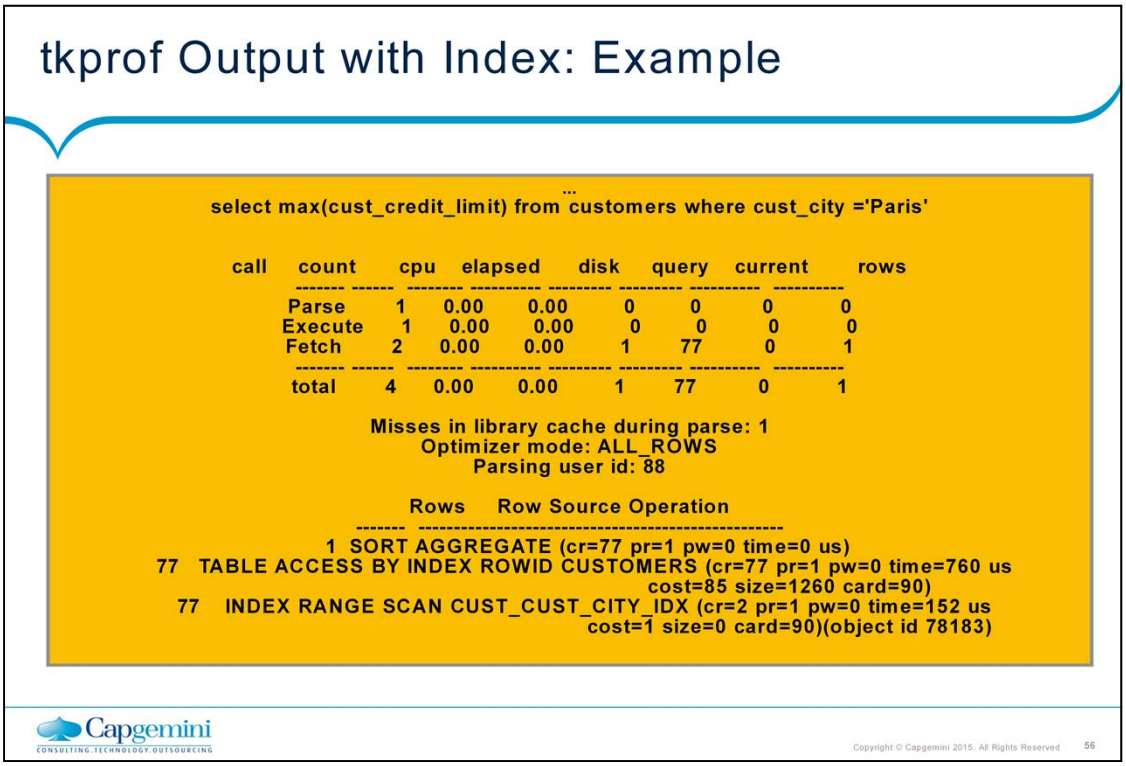
Output of the tkprof Command (continued)

- Recursive Calls
- To execute a SQL statement issued by a user, the Oracle server must occasionally issue additional statements. Such statements are called recursive SQL statements. For example, if you insert a row in a table that does not have enough space to hold that row, the Oracle server makes recursive calls to allocate the space dynamically. Recursive calls are also generated when data dictionary information is not available in the data dictionary cache and must be retrieved from disk.
- If recursive calls occur while the SQL Trace facility is enabled, tkprof marks them clearly as recursive SQL statements in the output file. You can suppress the listing of recursive calls in the output file by setting the SYS=NO command-line parameter. Note that the statistics for recursive SQL statements are always included in the listing for the SQL statement that caused the recursive call.
- Library Cache Misses
- tkprof also lists the number of library cache misses resulting from parse and execute steps for each SQL statement. These statistics appear on separate lines following the tabular statistics.



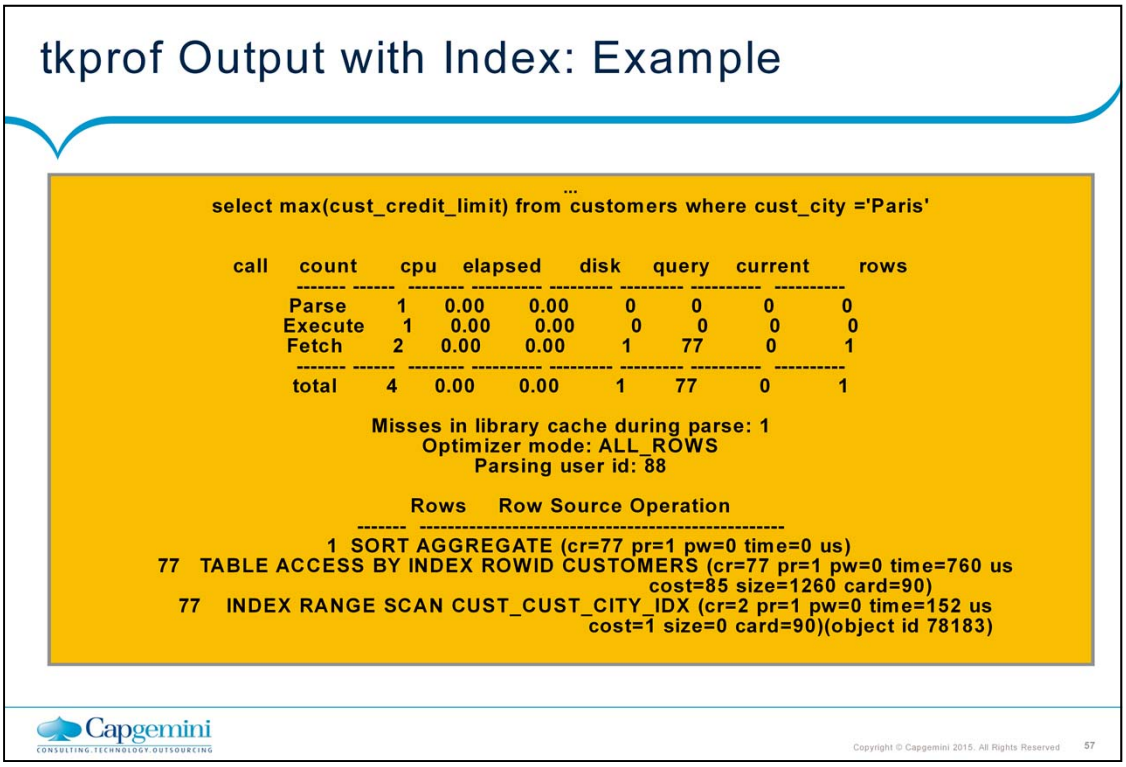
tkprof Output with No Index: Example

- The example in the slide shows that the aggregation of results across several executions (rows) is being fetched from the CUSTOMERS table. It requires 0.12 second of CPU fetch time. The statement is executed through a full table scan of the CUSTOMERS table, as you can see in the row source operation of the output.
- The statement must be optimized.
- Note: If CPU or elapsed values are 0, timed_statistics is not set.



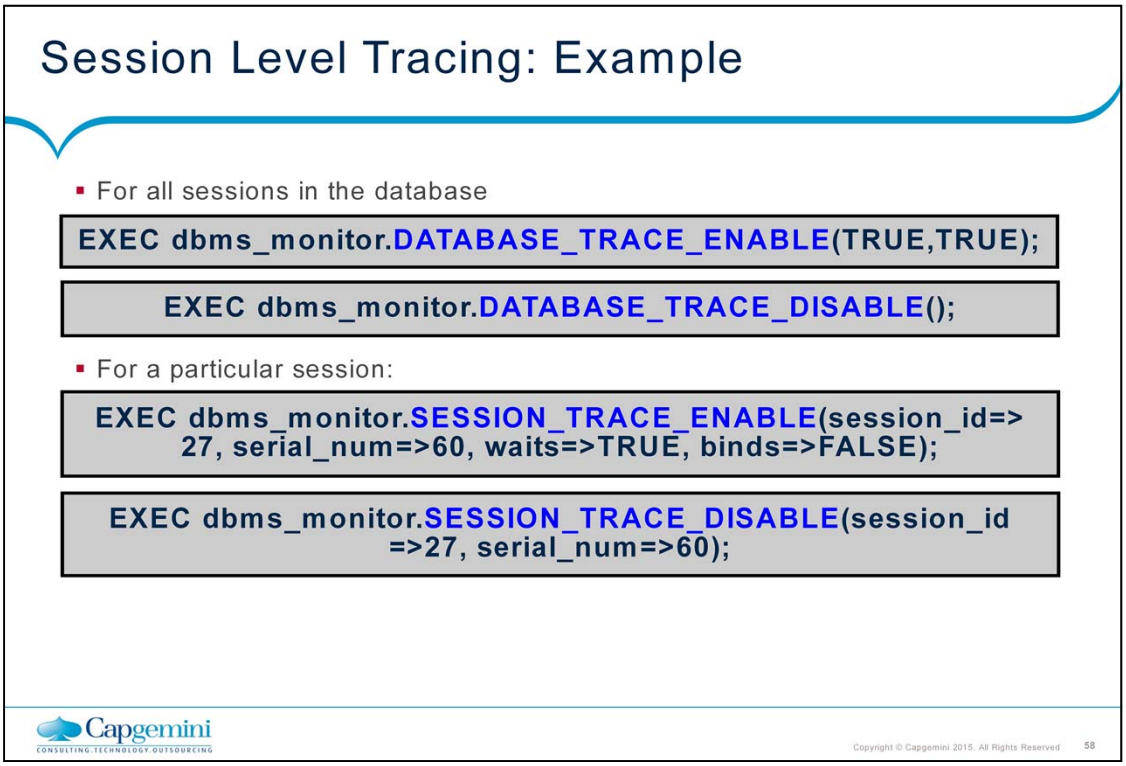
tkprof Output with Index: Example

- The results shown in the slide indicate that CPU time was reduced to 0.01 second when an index was created on the CUST_CITY column. These results may have been achieved because the statement uses the index to retrieve the data. Additionally, because this example reexecutes the same statement, most of the data blocks are already in memory. You can achieve significant improvements in performance by indexing sensibly. Identify areas for potential improvement using the SQL Trace facility.
- Note: Indexes should not be built unless required. Indexes do slow down the processing of the INSERT, UPDATE, and DELETE commands because references to rows must be added, changed, or removed. Unused indexes should be removed. However, instead of processing all the application SQL through EXPLAIN PLAN, you can use index monitoring to identify and remove any indexes that are not used.



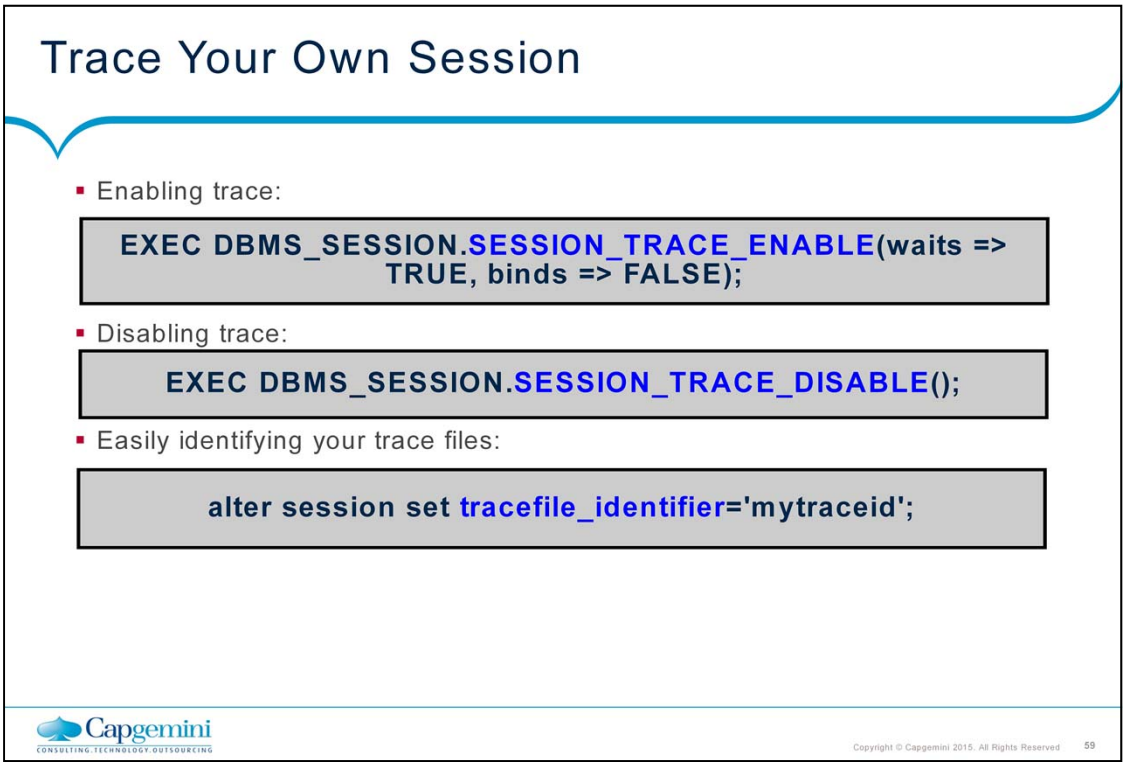
tkprof Output with Index: Example

- The results shown in the slide indicate that CPU time was reduced to 0.01 second when an index was created on the CUST_CITY column. These results may have been achieved because the statement uses the index to retrieve the data. Additionally, because this example reexecutes the same statement, most of the data blocks are already in memory. You can achieve significant improvements in performance by indexing sensibly. Identify areas for potential improvement using the SQL Trace facility.
- Note: Indexes should not be built unless required. Indexes do slow down the processing of the INSERT, UPDATE, and DELETE commands because references to rows must be added, changed, or removed. Unused indexes should be removed. However, instead of processing all the application SQL through EXPLAIN PLAN, you can use index monitoring to identify and remove any indexes that are not used.



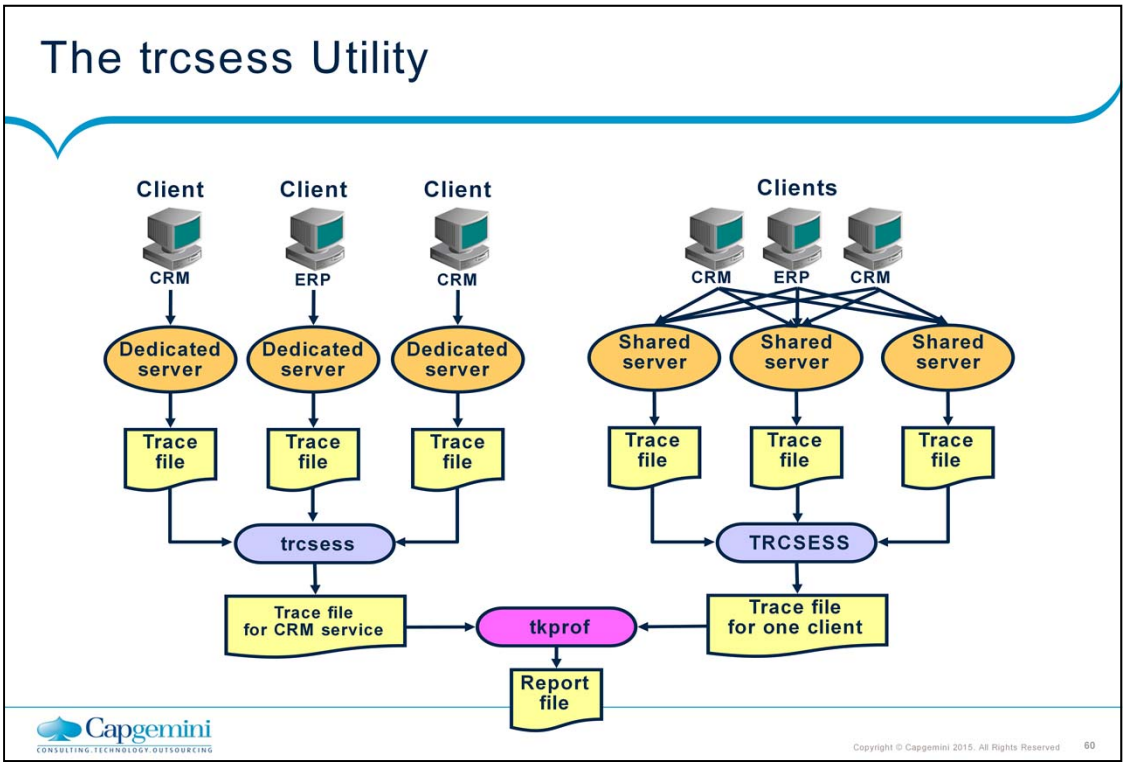
Session Level Tracing: Example

- You can use tracing to debug performance problems. Trace-enabling procedures have been implemented as part of the DBMS_MONITOR package. These procedures enable tracing globally for a database.
- You can use the DATABASE_TRACE_ENABLE procedure to enable session level SQL tracing instance-wide. The procedure has the following parameters:
  - WAITS: Specifies whether wait information is to be traced
  - BINDS: Specifies whether bind information is to be traced
  - INSTANCE_NAME: Specifies the instance for which tracing is to be enabled. Omitting INSTANCE_NAME means that the session-level tracing is enabled for the whole database.
- Use the DATABASE_TRACE_DISABLE procedure to disable SQL tracing for the whole database or a specific instance.
- Similarly, you can use the SESSION_TRACE_ENABLE procedure to enable tracing for a given database session identifier on the local instance. The SID and SERIAL# information can be found from V$SESSION.



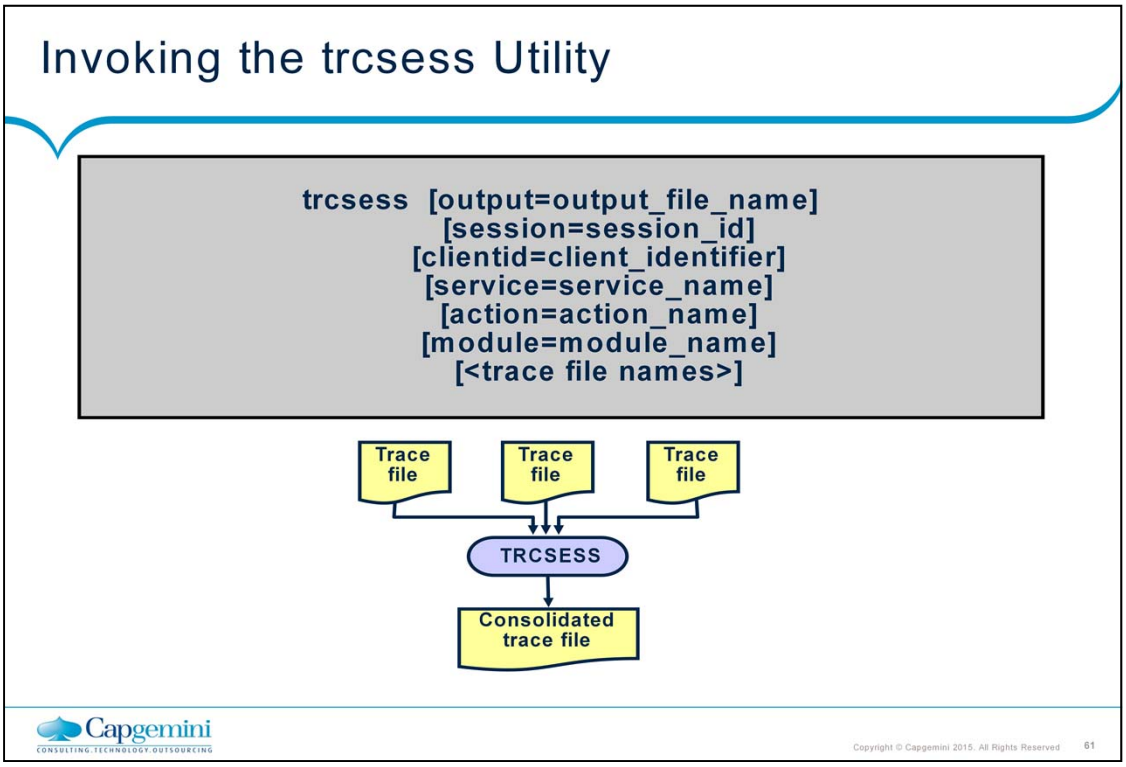
Trace Your Own Session

- Although the DBMS_MONITOR package can be invoked only by a user with the DBA role, any user can enable SQL tracing for his or her own session by using the DBMS_SESSION package. The SESSION_TRACE_ENABLE procedure can be invoked by any user to enable session level SQL tracing for his or her own session. An example is shown in the slide.
- You can then use the DBMS_SESSION.SESSION_TRACE_DISABLE procedure to stop dumping to your trace file.
- The TRACEFILE_IDENTIFIER initialization parameter specifies a custom identifier that becomes part of the Oracle trace file name. You can use such a custom identifier to identify a trace file simply from its name and without opening it or view its contents. Each time this parameter is dynamically modified at the session level, the next trace dump written to a trace file will have the new parameter value embedded in its name. This parameter can only be used to change the name of the foreground process trace file; the background processes continue to have their trace files named in the regular format. For foreground processes, the TRACEID column of the V$PROCESS view contains the current value of this parameter. When this parameter value is set, the trace file name has the following format: sid_ora_pid_traceid.trc.



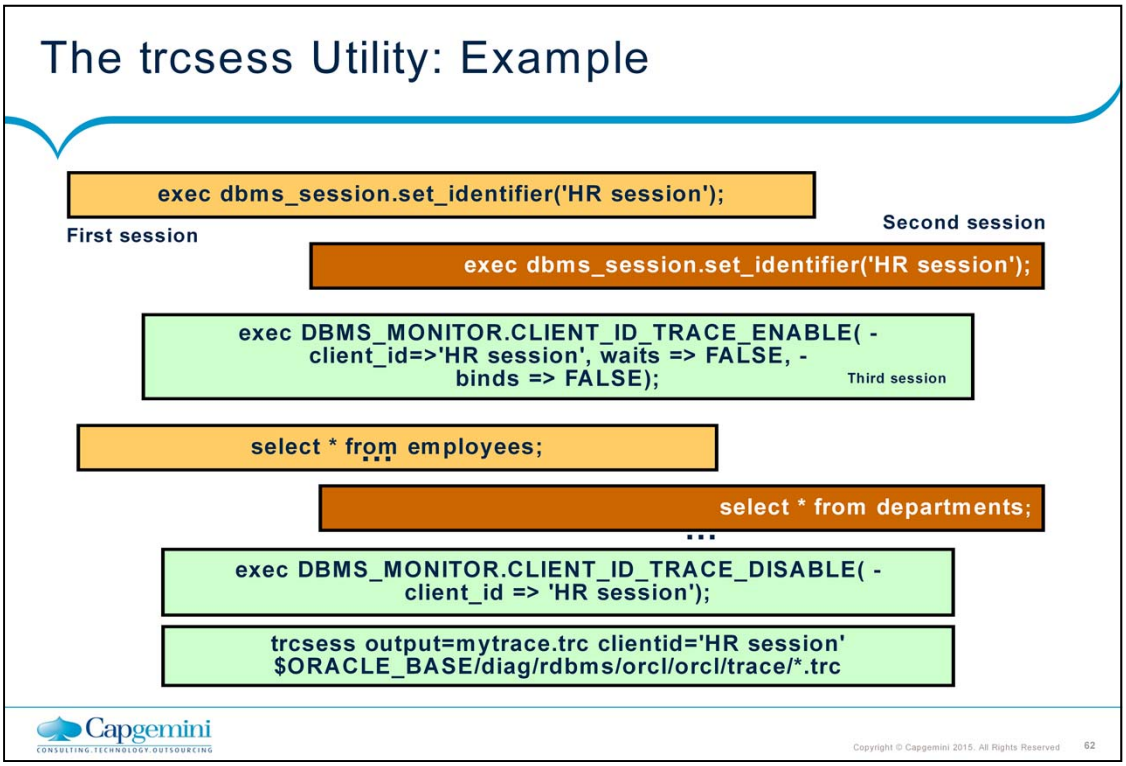
The trcsess Utility

- The trcsess utility consolidates trace output from selected trace files on the basis of several criteria: session ID, client identifier, service name, action name, and module name. After trcsess merges the trace information into a single output file, the output file can be processed by tkprof.
- When using the DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE procedure, tracing information is present in multiple trace files and you must use the trcsess tool to collect it into a single file.
- The trcsess utility is useful for consolidating the tracing of a particular session or service for performance or debugging purposes.
- Tracing a specific session is usually not a problem in the dedicated server model because a single dedicated process serves a session during its lifetime. All the trace information for the session can be seen from the trace file belonging to the dedicated server that serves it. However, tracing a service might become a complex task even in the dedicated server model.
- Moreover, in a shared-server configuration, a user session is serviced by different processes from time-to-time. The trace pertaining to the user session is scattered across different trace files belonging to different processes. This makes it difficult to get a complete picture of the life cycle of a session.



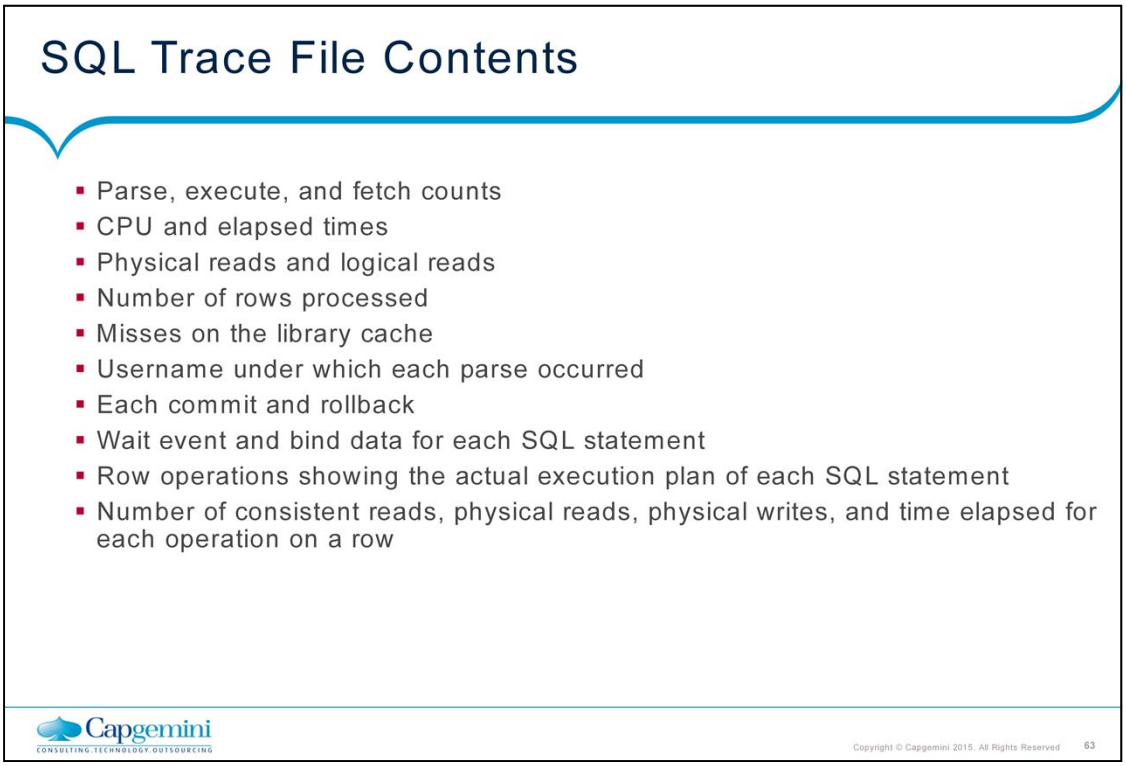
Invoking the trcsess Utility

- The syntax for the trcsess utility is shown in the slide, where:
  - output specifies the file where the output is generated. If this option is not specified, standard output is used for the output.
  - session consolidates the trace information for the session specified. The session identifier is a combination of session index and session serial number, such as 21.2371. You can locate these values in the V$SESSION view.
  - clientid consolidates the trace information for the given client identifier.
  - service consolidates the trace information for the given service name.
  - action consolidates the trace information for the given action name.
  - module consolidates the trace information for the given module name.
  - <trace file names> is a list of all the trace file names, separated by spaces, in which trcsess should look for trace information. The wildcard character “*” can be used to specify the trace file names. If trace files are not specified, all the files in the current directory are taken as input to trcsess. You can find trace files in ADR.
- Note: One of the session, clientid, service, action, or module options must be specified. If there is more than one option specified, the trace files, which satisfy all the criteria specified are consolidated into the output file.



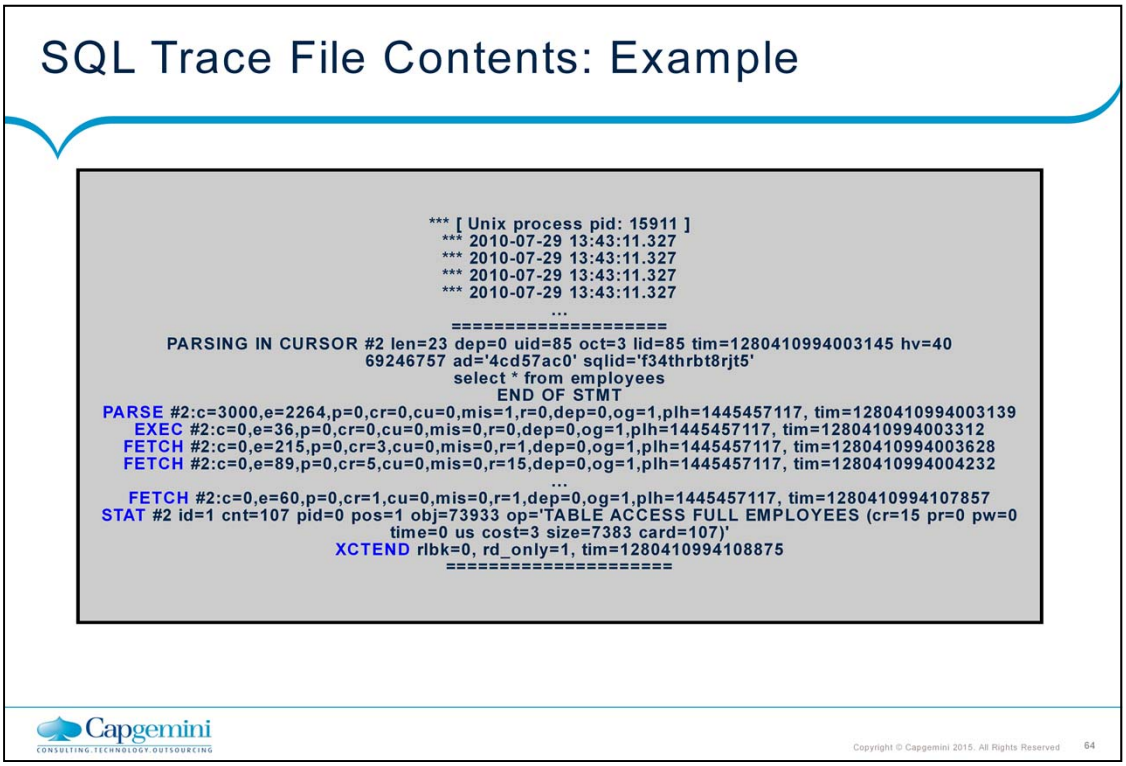
The trcsess Utility: Example

- The example in the slide illustrates a possible use of the trcsess utility. The example assumes that you have three different sessions: Two sessions that are traced (left and right), and one session (center) that enables or disables tracing and concatenates trace information from the previous two sessions.
- The first and second session set their client identifier to the ‘HR session’ value. This is done using the DBMS_SESSION package. Then, the third session enables tracing for these two sessions using the DBMS_MONITOR package.
- At that point, two new trace files are generated in ADR; one for each session that is identified with the ‘HR session’ client identifier.
- Each traced session now executes its SQL statements. Every statement generates trace information in its own trace file in ADR.
- Then, the third session stops trace generation using the DBMS_MONITOR package, and consolidates trace information for the ‘HR session’ client identifier in the mytrace.trc file. The example assumes that all trace files are generated in the $ORACLE_BASE/diag/rdbms/orcl/orcl/trace directory, which is the default in most cases.



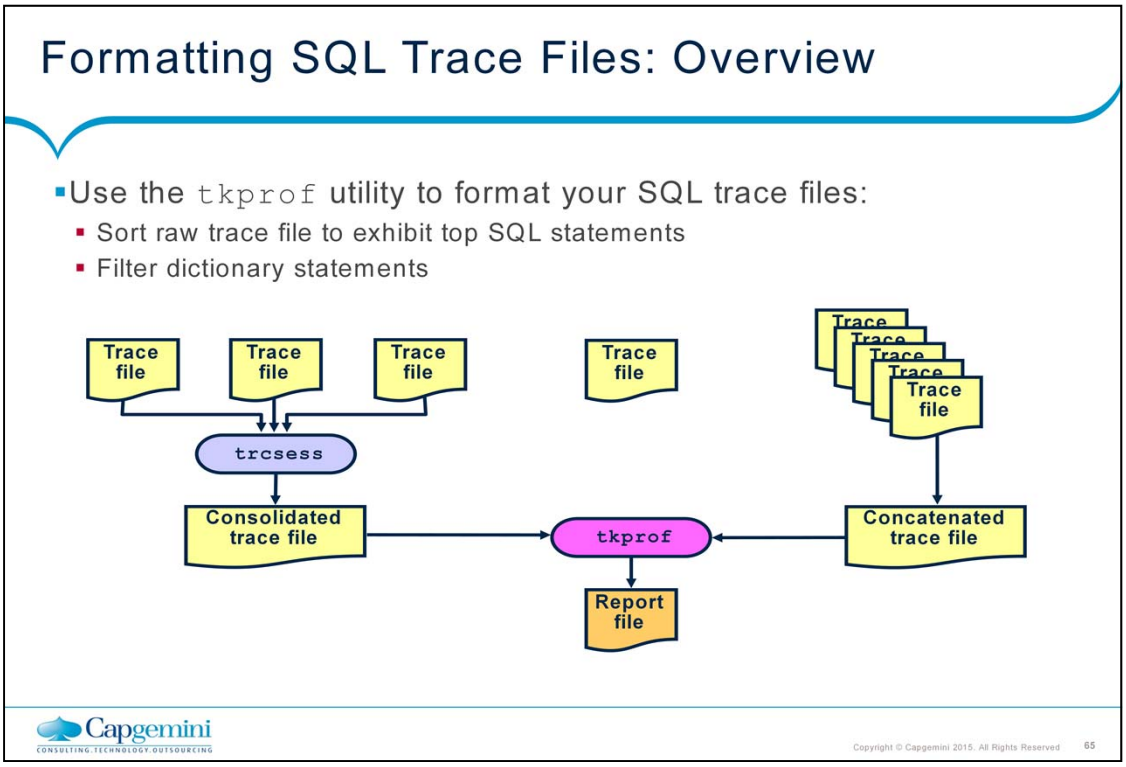
SQL Trace File Contents

- As seen already, a SQL trace file provides performance information on individual SQL statements. It generates the following statistics for each statement:
  - Parse, execute, and fetch counts
  - CPU and elapsed times
  - Physical reads and logical reads
  - Number of rows processed
  - Misses on the library cache
  - Username under which each parse occurred
  - Each commit and rollback
  - Wait event data for each SQL statement, and a summary for each trace file
- If the cursor for the SQL statement is closed, SQL Trace also provides row source information that includes:
  - Row operations showing the actual execution plan of each SQL statement
  - Number of rows, number of consistent reads, number of physical reads, number of physical writes, and time elapsed for each operation. This is possible only when the STATISTICS_LEVEL initialization parameter is set to ALL.
- Note: Using the SQL Trace facility can have a severe performance impact and may result in increased system overhead, excessive CPU usage, and inadequate disk space.



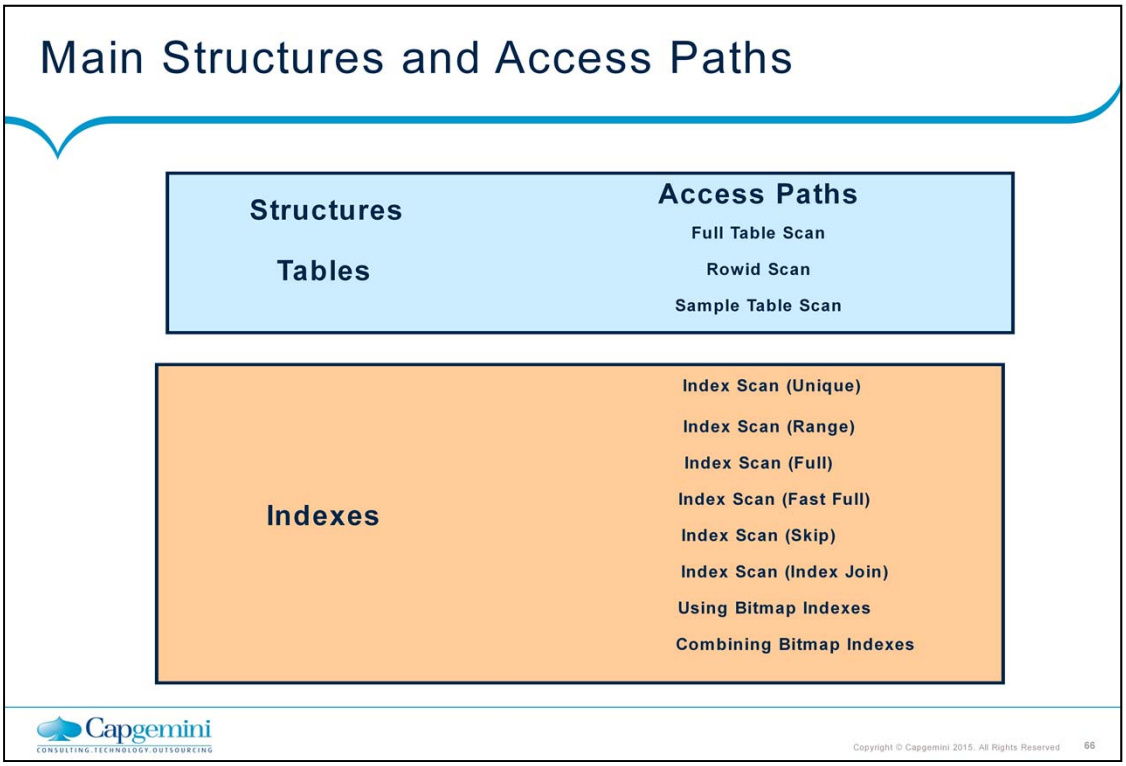
SQL Trace File Contents: Example

- There are multiple types of trace files that can be generated by the Oracle Database. The one that is referred to in this lesson is generally called a SQL trace file. The slide shows you a sample output from the mytrace.trc SQL trace file generated by the previous example.
- In this type of trace file, you can find (for each statement that was traced) the statement itself, with some corresponding cursor details. You can see statistic details for each phase of the statement's execution: PARSE, EXEC, and FETCH. As you can see, you can have multiple FETCH for one EXEC depending on the number of rows returned by your query.
- Last part of the trace is the execution plan with some cumulated statistics for each row source.
- Depending on the way you enabled tracing, you can also obtain information about wait events and bind variables in the generated trace files.
- Generally, you do not try to interpret the trace file itself. This is because you do not get an overall idea of what your sessions did. For example, one session could have executed the same statement multiple times at different moments. The corresponding traces are then scattered across the entire trace file, which makes them hard to find.
- Instead, you use another tool, such as tkprof to interpret the contents of the raw trace information.



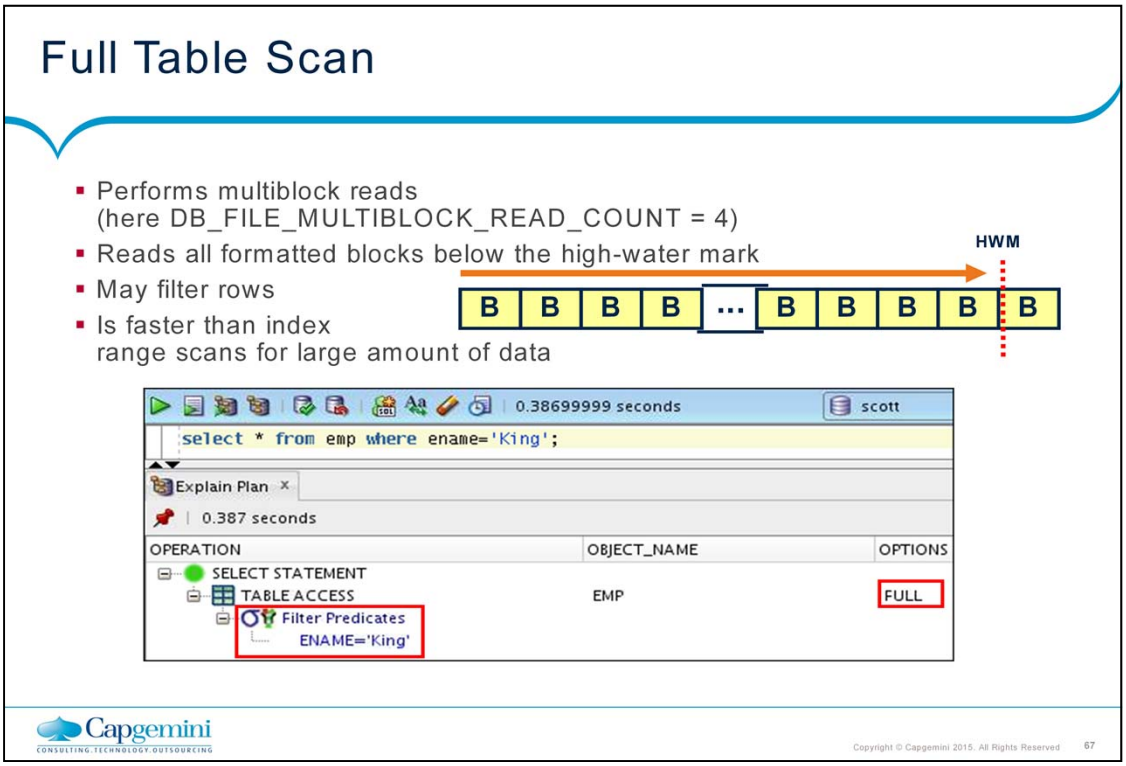
Formatting SQL Trace Files: Overview

- The tkprof utility parses SQL trace files to produce more readable output. Remember that all the information in tkprof is available from the raw trace file. There is a huge number of sort options that you can invoke with tkprof at the command prompt. A useful starting point is the fchela sort option, which orders the output by elapsed time fetching. The resultant file contains the most time-consuming SQL statement at the start of the file. Another useful parameter is SYS=NO. This can be used to prevent SQL statements run as the SYS user from being displayed. This can make the output file much shorter and easier to manage.
- After a number of SQL trace files have been generated, you can perform any of the following:
  - Run tkprof on each individual trace file, producing a number of formatted output files, one for each session.
  - Concatenate the trace files, and then run tkprof on the result to produce a formatted output file for the entire instance.
  - Run the trcsess command-line utility to consolidate tracing information from several trace files, then run tkprof on the result.
- tkprof does not report COMMITs and ROLLBACKs that are recorded in the trace file.
- Note: Set the TIMED_STATISTICS parameter to TRUE when tracing sessions because no
time-based comparisons can be made without this. TRUE is the default value with Oracle Database 11g.



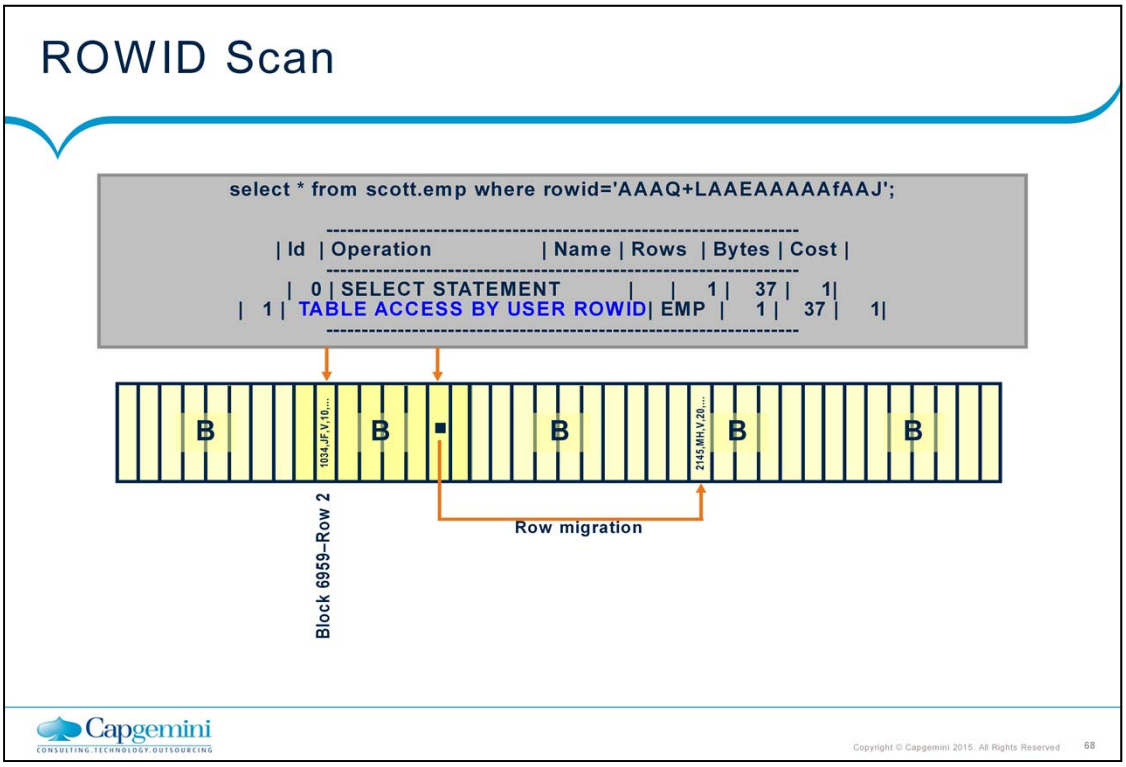
Main Structures and Access Paths

- Any row can be located and retrieved with one of the methods mentioned in the slide.
- In general, index access paths should be used for statements that retrieve a small subset of table rows, while full scans are more efficient when accessing a large portion of the table. To decide on the alternative, the optimizer gives each alternative (execution plan) a cost. The one with the lower cost is elected.
- There are special types of table access paths including clusters, index-organized tables, and partitions, which have not been mentioned in the slide.
- Clusters are an optional method of storing table data. A cluster is a group of tables that share the same data blocks because they share common columns and are often used together. For example, the EMP and DEPT table share the DEPTNO column. When you cluster the EMP and DEPT tables, Oracle physically stores all rows for each department from both the EMP and DEPT tables in the same data blocks.
- Hash clusters are single-table clusters in which rows with the same hash-key values are stored together. A mathematical hash function is used to select the location of a row within the cluster. All rows with the same key value are stored together on disk.
- The special types of access paths are discussed later in this course.



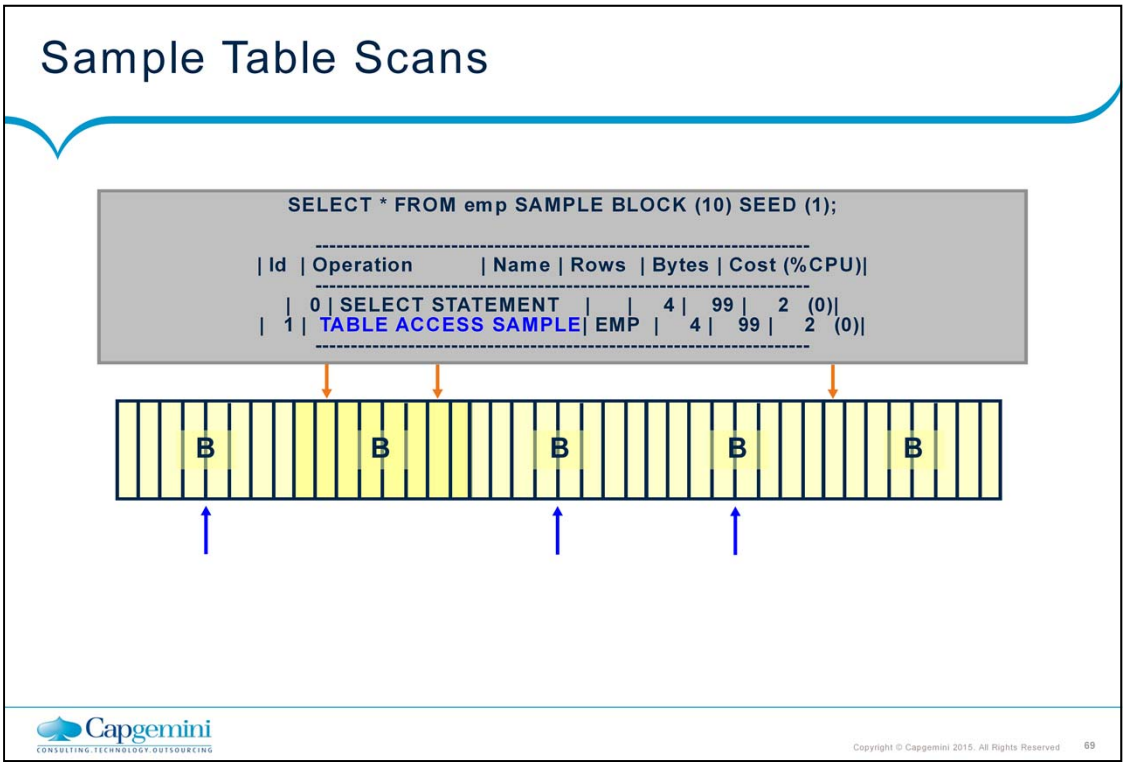
Full Table Scan

- A full table scan sequentially reads all rows from a table and filters out those that do not meet the selection criteria. During a full table scan, all formatted blocks in the table that are under the high-water mark are scanned even if all the rows have been deleted from the table. Each block is read only once. The high-water mark indicates the amount of used space, or space that was formatted to receive data. Each row is examined to determine whether it satisfies the statement's WHERE clause using the applicable filter conditions specified in the query.
- You can see the filter conditions in the "Predicate Information" section of the explain plan. The filter to be applied returns only rows where EMP.ENAME='King'.
  Because a full table scan reads all the formatted blocks in a table, it reads blocks that are physically adjacent to each other. This means that performance benefits can be reaped by utilizing input/output (I/O) calls that read multiple blocks at the same time. The size of the read call can range from a single block to any number of blocks up to the DB_FILE_MULTIBLOCK_READ_COUNT init parameter.
- Note: In Oracle 6, a full table scan (FTS) could flood the buffer cache because there was no difference in the way blocks were handled between FTS and other reads. Since Oracle V7, blocks read by FTS are allowed to occupy only a small percentage of the buffer cache. Currently, FTS are read into the PGA with direct reads bypassing the buffer cache in most cases.



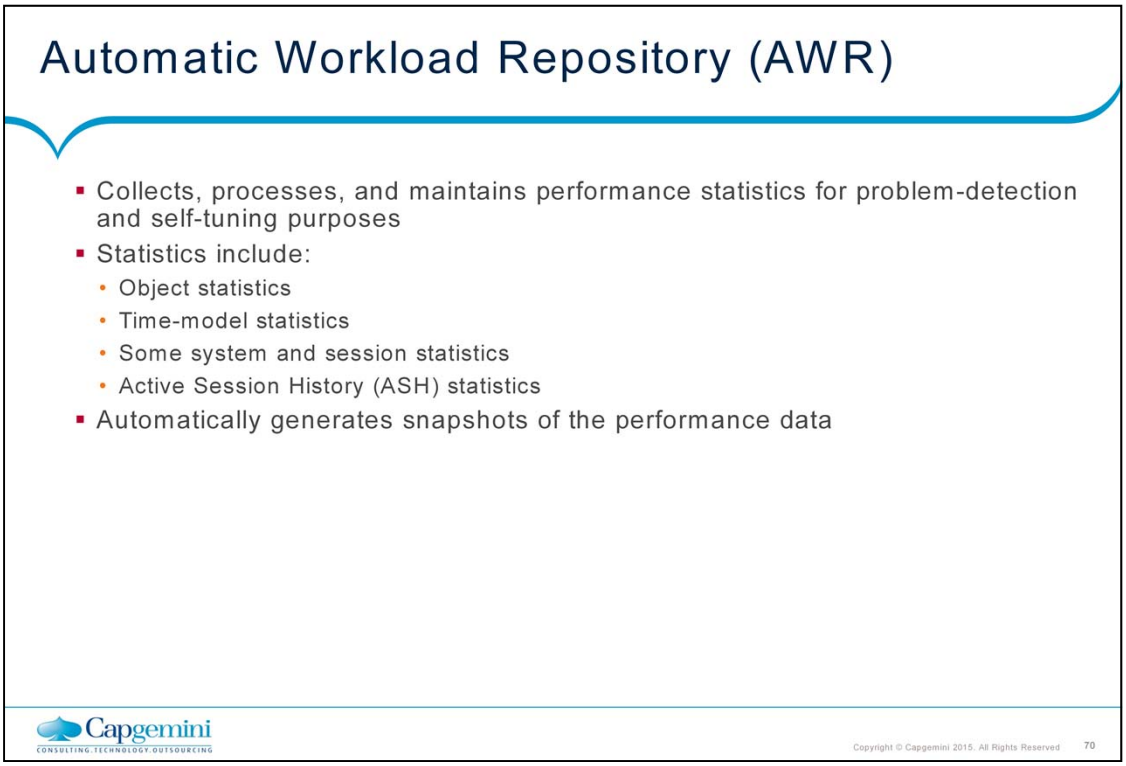
ROWID Scan

- The rowid of a row specifies the data file and data block containing the row and the location of the row in that block. Locating a row by specifying its rowid is the fastest way to retrieve a single row because the exact location of the row in the database is specified.
- To access a table by rowid, the system first obtains the rowids of the selected rows, either from the statement's WHERE clause or through an index scan of one or more of the table's indexes. The system then locates each selected row in the table based on its rowid.
- Mostly, the optimizer uses rowids after retrieving them from an index (See the "Index Scans" slides.). The table access might be required for columns in the statement that are not present in the index. A table access by rowid does not need to follow every index scan. If the index contains all the columns needed for the statement, table access by rowid might not occur.
- Rowids are the system's internal representation of where data is stored. Accessing data based on position is not recommended because rows can move around due to row migration and chaining, and also after export and import.
- Note: Due to row migration, a rowid can sometimes point to an address different from the actual row location, resulting in more than one block being accessed to locate a row. For example, an update to a row may cause the row to be placed in another block with a pointer in the original block. The rowid, however, still has only the address of the original block.



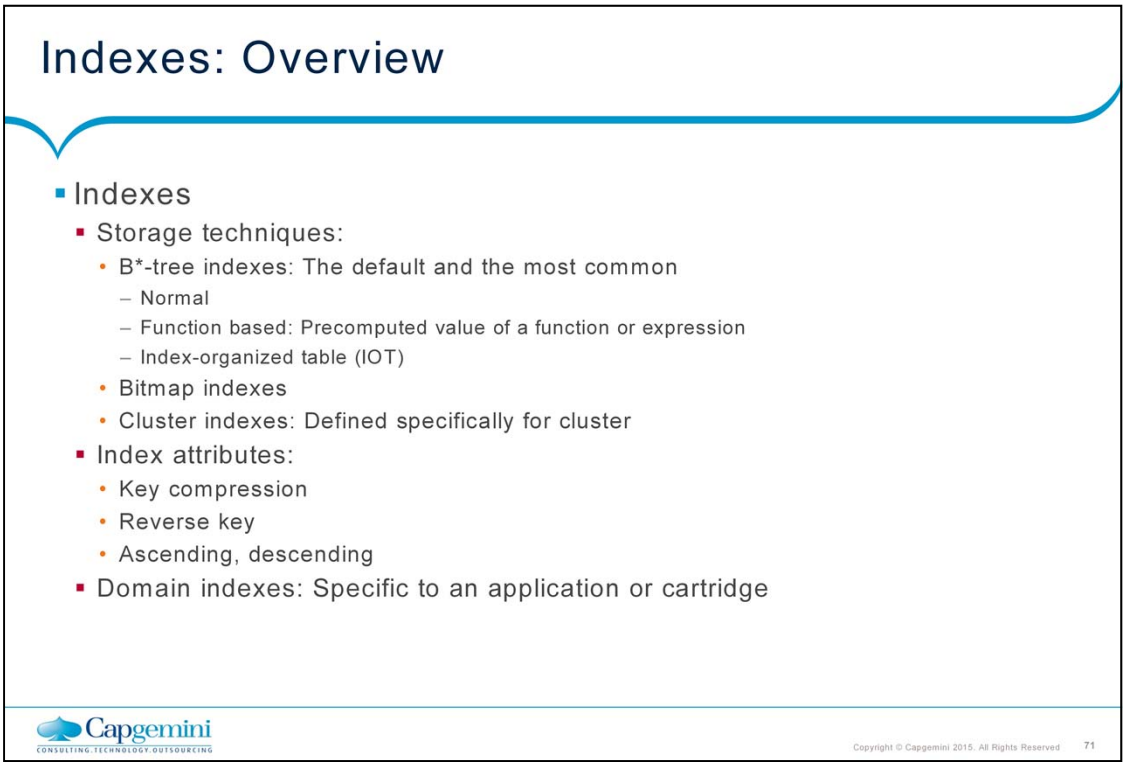
Sample Table Scans

- A sample table scan retrieves a random sample of data from a simple table or a complex SELECT statement, such as a statement involving joins and views. This access path is used when a statement's FROM clause includes the SAMPLE clause or the SAMPLE BLOCK clause. To perform a sample table scan when sampling by rows with the SAMPLE clause, the system reads a specified percentage of rows in the table. To perform a sample table scan when sampling by blocks with the SAMPLE BLOCK clause, the system reads a specified percentage of table blocks.
  - SAMPLE option: To perform a sample table scan when sampling by rows, the system reads a specified percentage of rows in the table and examines each of these rows to determine whether it satisfies the statement's WHERE clause.
  - SAMPLE BLOCK option: To perform a sample table scan when sampling by blocks, the system reads a specified percentage of the table's blocks and examines each row in the sampled blocks to determine whether it satisfies the statement's WHERE clause.
- The sample percent is a number specifying the percentage of the total row or block count to be included in the sample. The sample value must be in the [0.000001 , 99.999999] range.
- This percentage indicates the probability of each row, or each cluster of rows in the case of block sampling, being selected as part of the sample. It does not mean that the database retrieves exactly sample_percent of the rows of table.



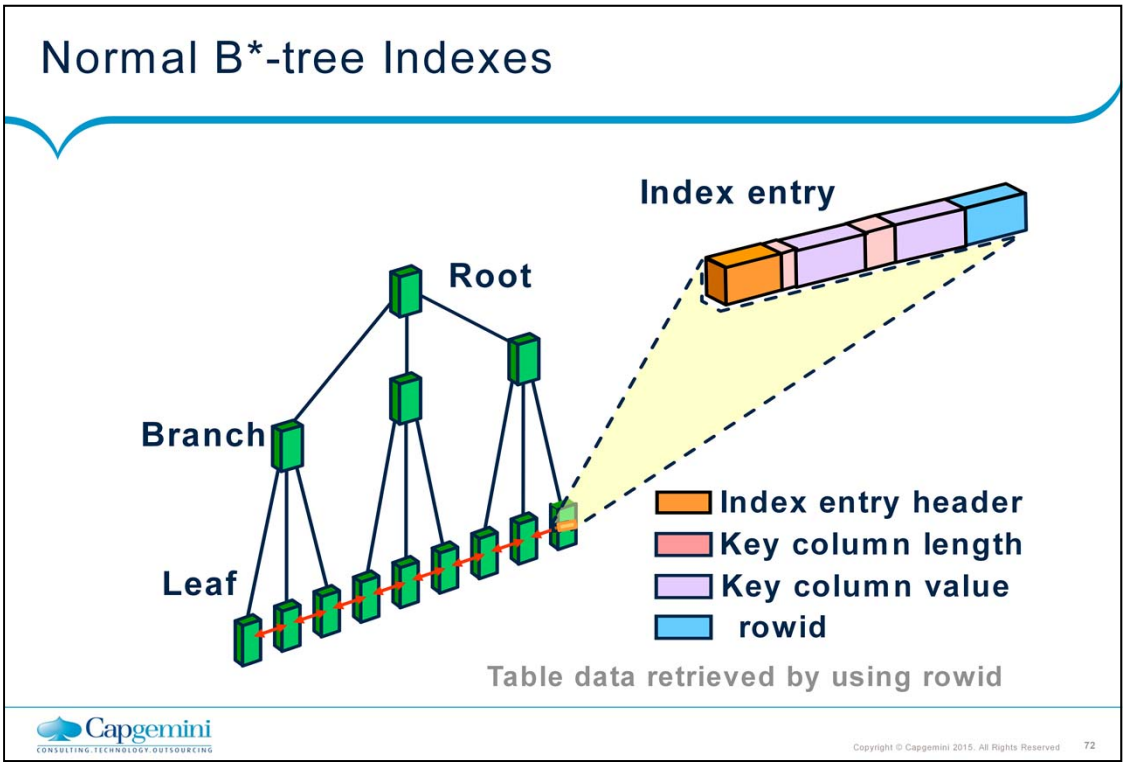
Automatic Workload Repository (AWR)

- The AWR is part of the intelligent infrastructure introduced with Oracle Database 10g. This infrastructure is used by many components, such as Automatic Database Diagnostic Monitor (ADDM) for analysis. The AWR automatically collects, processes, and maintains system-performance statistics for problem-detection and self-tuning purposes and stores the statistics persistently in the database.
- The statistics collected and processed by the AWR include:
  - Object statistics that determine both access and usage statistics of database segments
  - Time-model statistics based on time usage for activities, displayed in the V$SYS_TIME_MODEL and V$SESS_TIME_MODEL views
  - Some of the system and session statistics collected in the V$SYSSTAT and V$SESSTAT views
  - SQL statements that produce the highest load on the system, based on criteria, such as elapsed time, CPU time, buffer gets, and so on
  - ASH statistics, representing the history of recent sessions



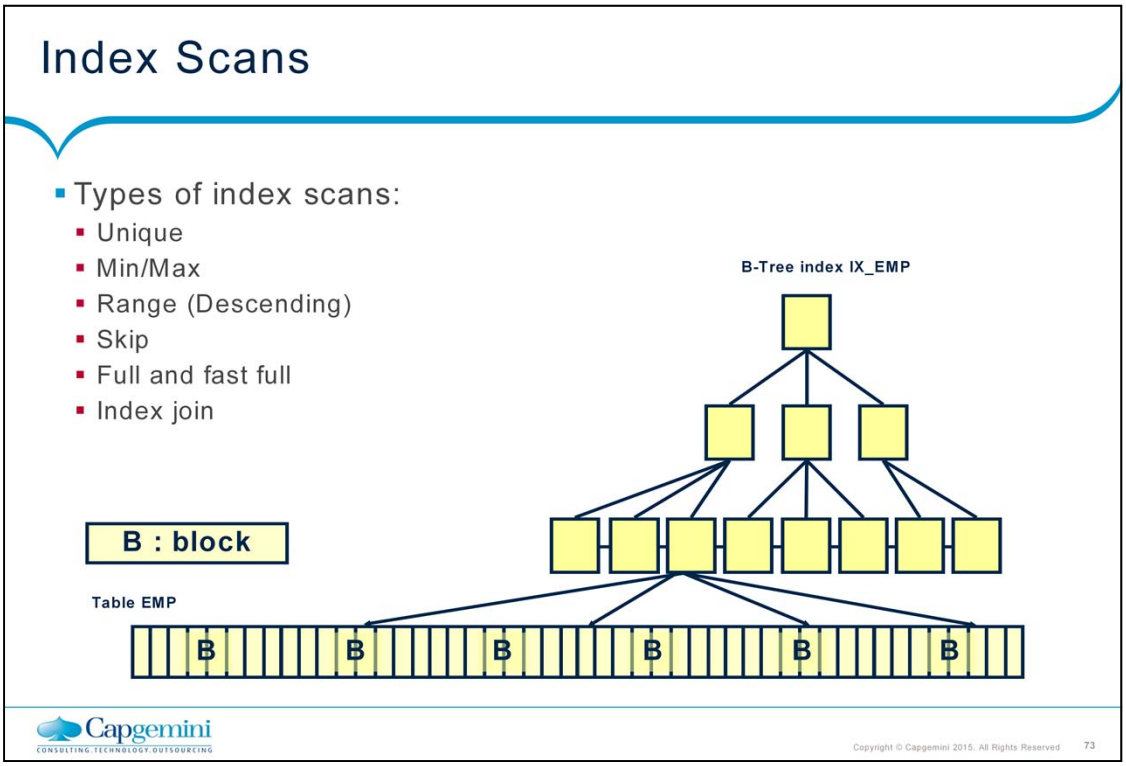
Indexes: Overview

- An index is an optional database object that is logically and physically independent of the table data. Being independent structures, they require storage space. Just as the index of a book helps you locate information fast, an Oracle Database index provides a faster access path to table data. The Oracle Database may use an index to access data that is required by a SQL statement, or it may use indexes to enforce integrity constraints. The system automatically maintains indexes when the related data changes. You can create and drop indexes at any time. If you drop an index, all applications continue to work. However, access to previously indexed data might be slower. Indexes can be unique or nonunique.
- A composite index, also called a concatenated index, is an index that you create on multiple columns (up to 32) in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.
  For standard indexes, the database uses B*-tree indexes that are balanced to equalize access times. B*-tree indexes can be normal, reverse key, descending, or function based.
  - B*-tree indexes: They are by far the most common indexes. Similar in construct to a binary tree, B*-tree indexes provide fast access, by key, to an individual row or range of rows, normally requiring few reads to find the correct row. However, the “B” in “B*-tree” does not stand for “binary,” but rather for “balanced.”



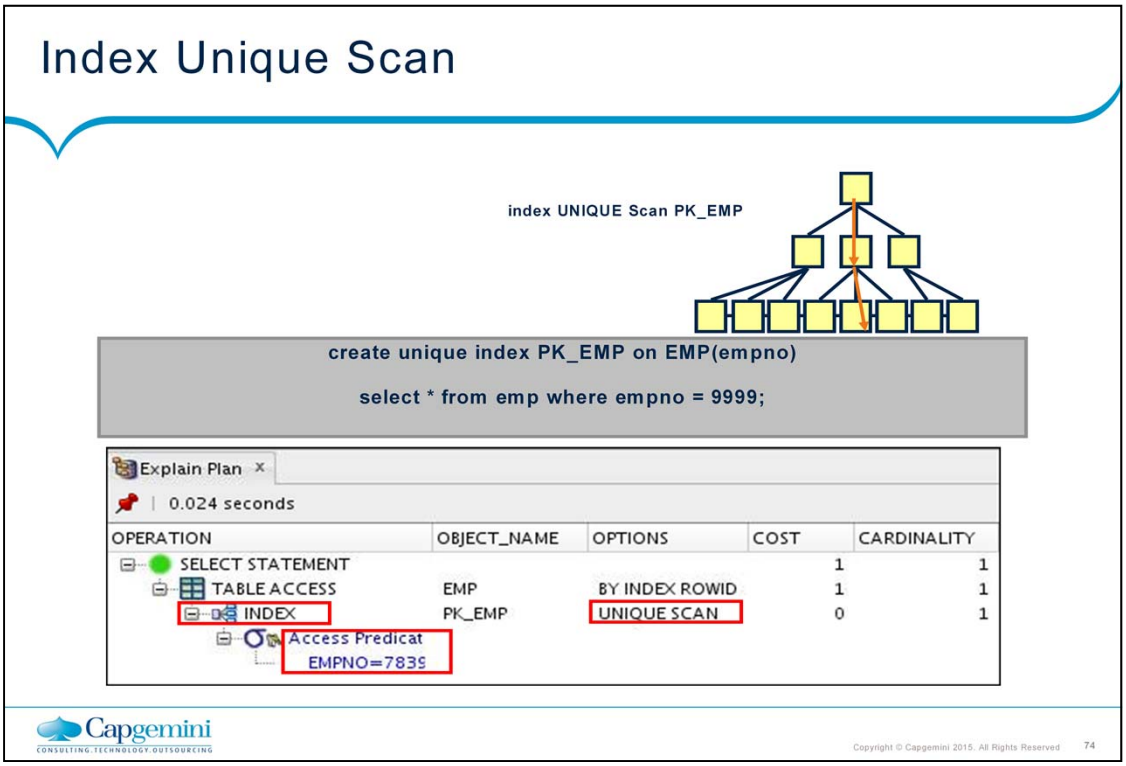
Normal B*-tree Indexes

- Each B*-tree index has a root block as a starting point. Depending on the number of entries, there are multiple branch blocks that can have multiple leaf blocks. The leaf blocks contain all values of the index plus ROWIDs that point to the rows in the associated data segment.
- Previous and next block pointers connect the leaf blocks so that they can be traversed from left to right (and vice versa).
  Indexes are always balanced, and they grow from the top down. In certain situations, the balancing algorithm can cause the B*-tree height to increase unnecessarily. It is possible to reorganize indexes. This is done by the ALTER INDEX … REBUILD | COALESCE command.
- The internal structure of a B*-tree index allows rapid access to the indexed values. The system can directly access rows after it has retrieved the address (the ROWID) from the index leaf blocks.
- Note: The maximum size of a single index entry is approximately one-half of the data block size.



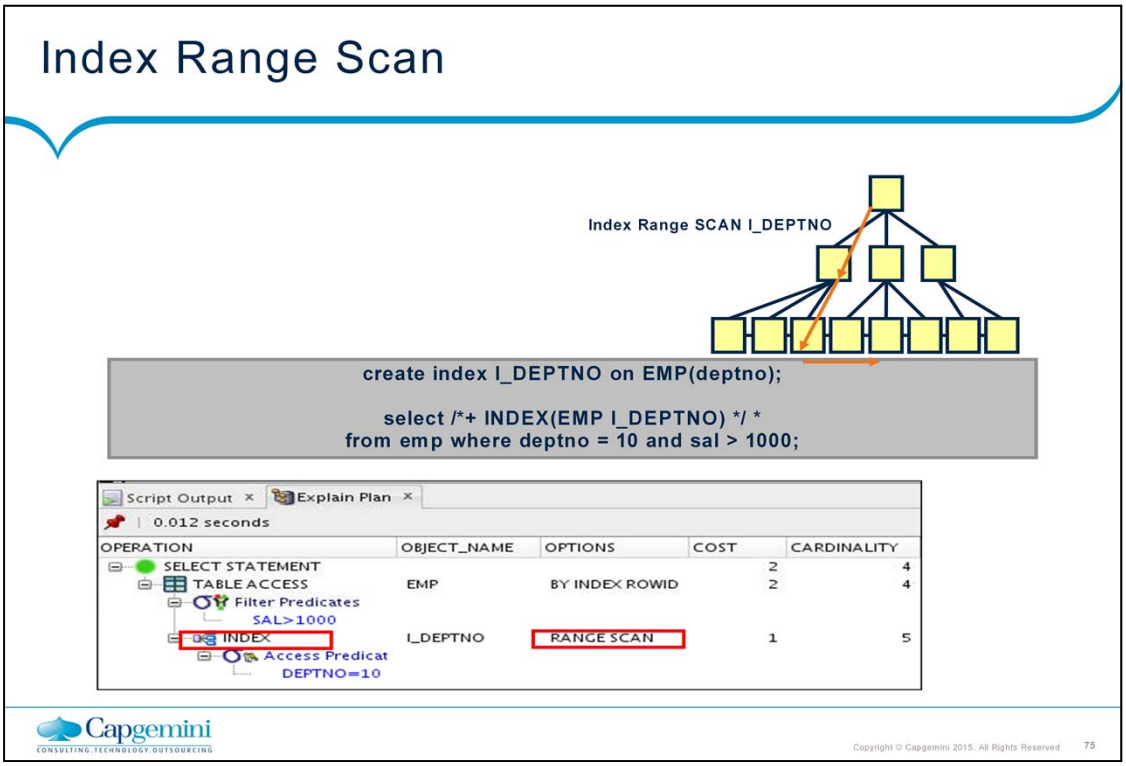
Index Scans

- An index scan can be one of the following types:
- A row is retrieved by traversing the index, using the indexed column values specified by the statement's WHERE clause. An index scan retrieves data from an index based on the value of one or more columns in the index. To perform an index scan, the system searches the index for the indexed column values accessed by the statement. If the statement accesses only columns of the index, the system reads the indexed column values directly from the index, rather than from the table.
- The index contains not only the indexed value, but also the rowids of rows in the table that have the value. Therefore, if the statement accesses other columns in addition to the indexed columns, the system can find the rows in the table by using either a table access by rowid or a cluster scan.
- Note: The graphic shows a case where four rows are retrieved from the table using their rowids obtained by the index scan.



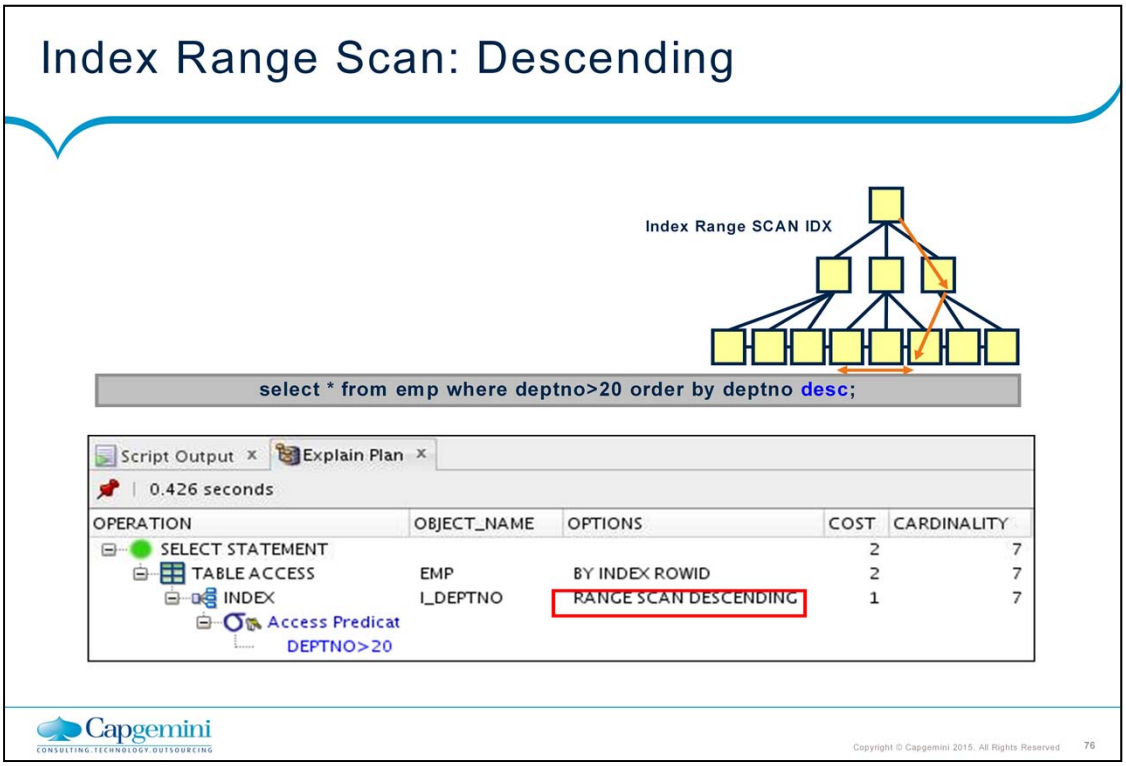
Index Unique Scan

- An index unique scan returns, at most, a single ROWID. The system performs a unique scan if a statement contains a UNIQUE or a PRIMARY KEY constraint that guarantees that only a single row is accessed. This access path is used when all the columns of a unique (B*-tree) index are specified with equality conditions.
- Key values and ROWIDs are obtained from the index, and table rows are obtained using ROWIDs.
- You can look for access conditions in the “Predicate Information” section of the execution plan (The execution plan is dealt with in detail in the lesson titled “Interpreting Execution Plans.”). Here the system accesses only matching rows for which EMPNO=9999.
- Note: Filter conditions filter rows after the fetch operation and output the filtered rows.



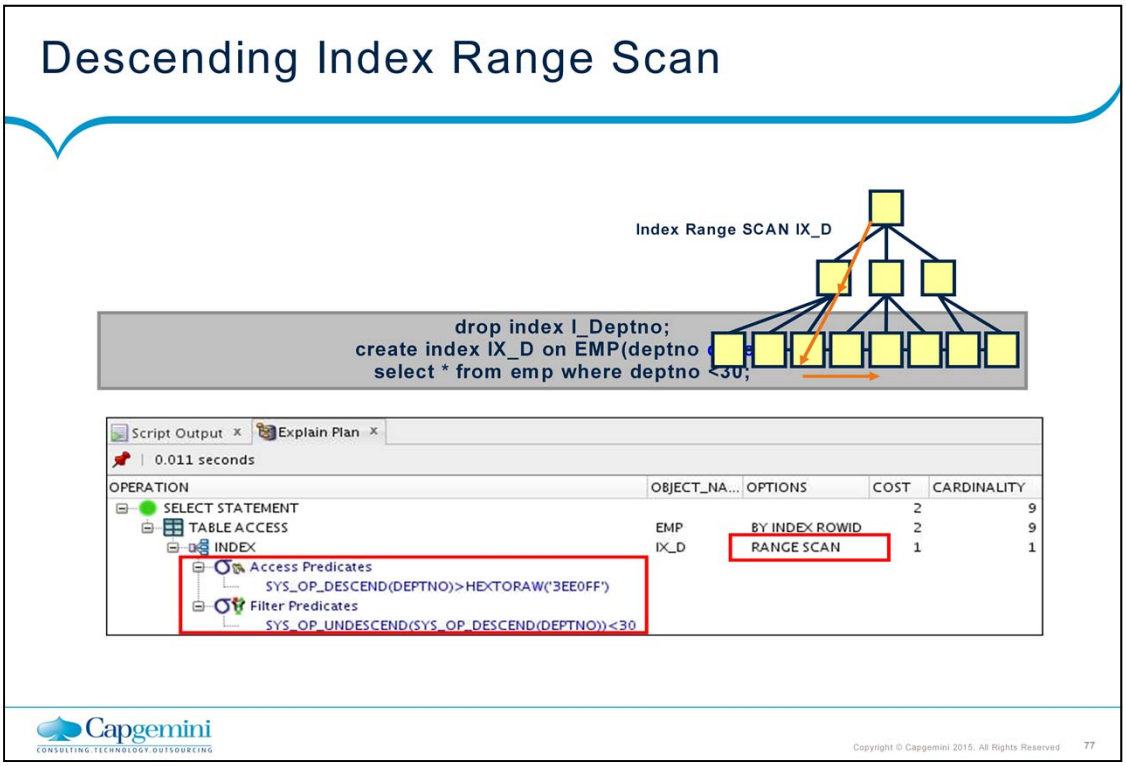
Index Range Scan

- An index range scan is a common operation for accessing selective data. It can be bounded (on both sides) or unbounded (on one or both sides). Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted in the ascending order by ROWID.
- The optimizer uses a range scan when it finds one or more leading columns of an index specified in conditions (the WHERE clause), such as col1 = :b1, col1 < :b1, col1 > :b1, and any combination of the preceding conditions.
- Wildcard searches (col1 like '%ASD') should not be in a leading position, as this does not result in a range scan.
- Range scans can use unique or nonunique indexes. Range scans can avoid sorting when index columns constitute the ORDER BY/GROUP BY clause and the indexed columns are NOT NULL as otherwise they are not considered.
- An index range scan descending is identical to an index range scan, except that the data is returned in the descending order. The optimizer uses index range scan descending when an order by descending clause can be satisfied by an index.
- In the example in the slide, using index I_DEPTNO, the system accesses rows for which EMP.DEPTNO=10. It gets their ROWIDs, fetches other columns from the EMP table, and finally, applies the EMP.SAL >1000 filter from these fetched rows to output the final result.



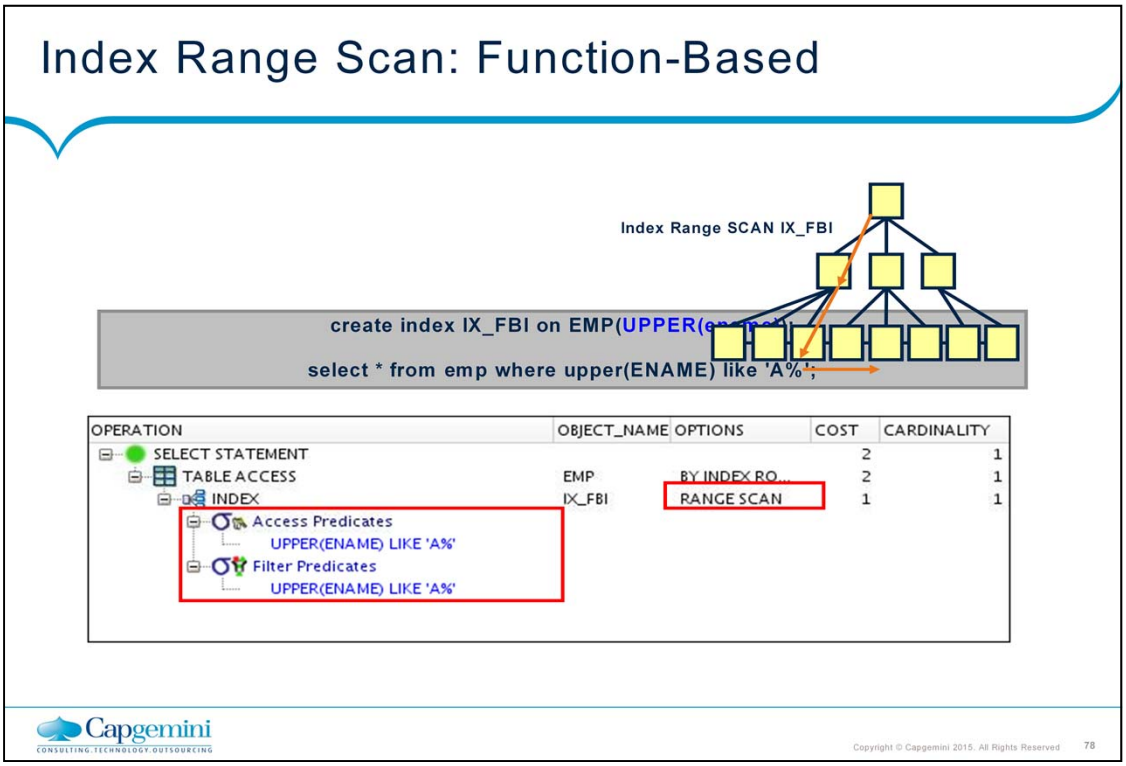
Index Range Scan: Descending

- In addition to index range scans in ascending order, which are described in the previous slide, the system is also able to scan indexes in the reverse order as illustrated by the graphic in the slide.
- The example retrieves rows from the EMP table by descending order on the DEPTNO column. You can see the DESCENDING operation row source for ID 2 in the execution plan that materialized this type of index scans.
- Note: By default an index range scan is done in the ascending order.



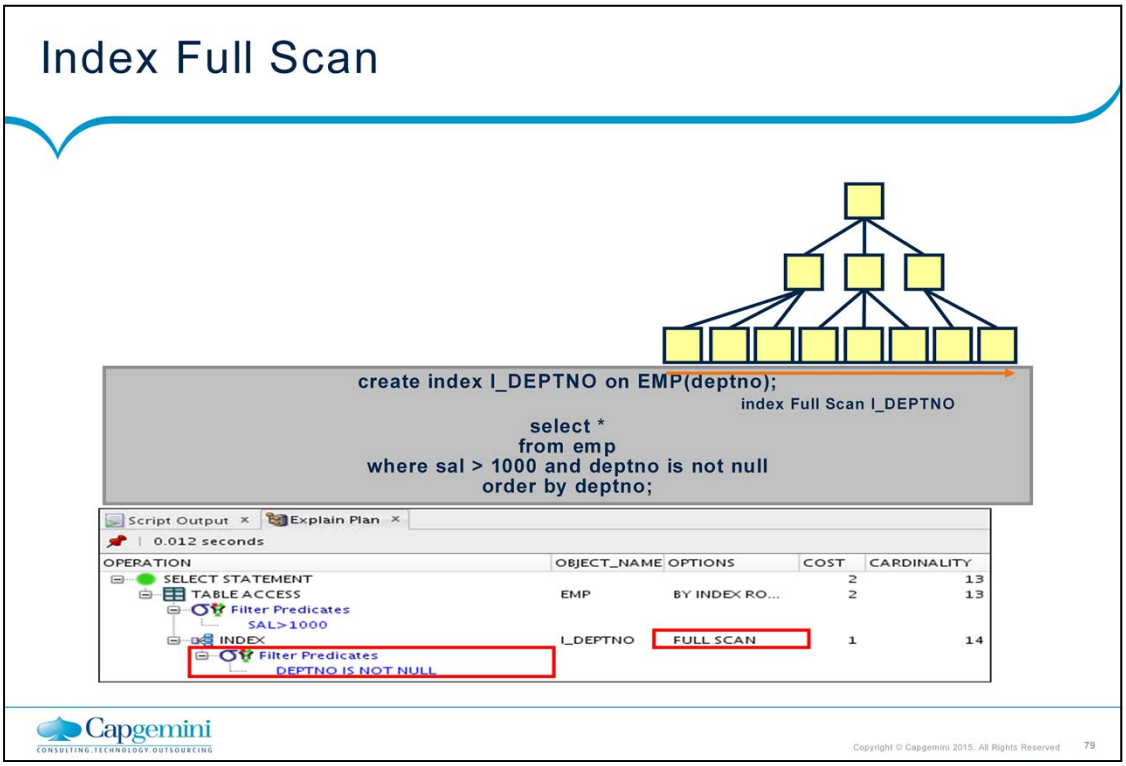
Descending Index Range Scan

- A descending index range scan is identical to an index range scan, except that the data is returned in descending order. Descending indexes allow for data to be sorted from “big to small” (descending) instead of “small to big” (ascending) in the index structure. Usually, this scan is used when ordering data in a descending order to return the most recent data first, or when seeking a value less than a specified value as in the example in the slide.
- The optimizer uses descending index range scan when an order by descending clause can be satisfied by a descending index.
- The INDEX_DESC(table_alias index_name) hint can be used to force this access path if possible.
- Note: The system treats descending indexes as function-based indexes. The columns marked DESC are stored in a special descending order in the index structure that is reversed again using the SYS_OP_UNDESCEND function.



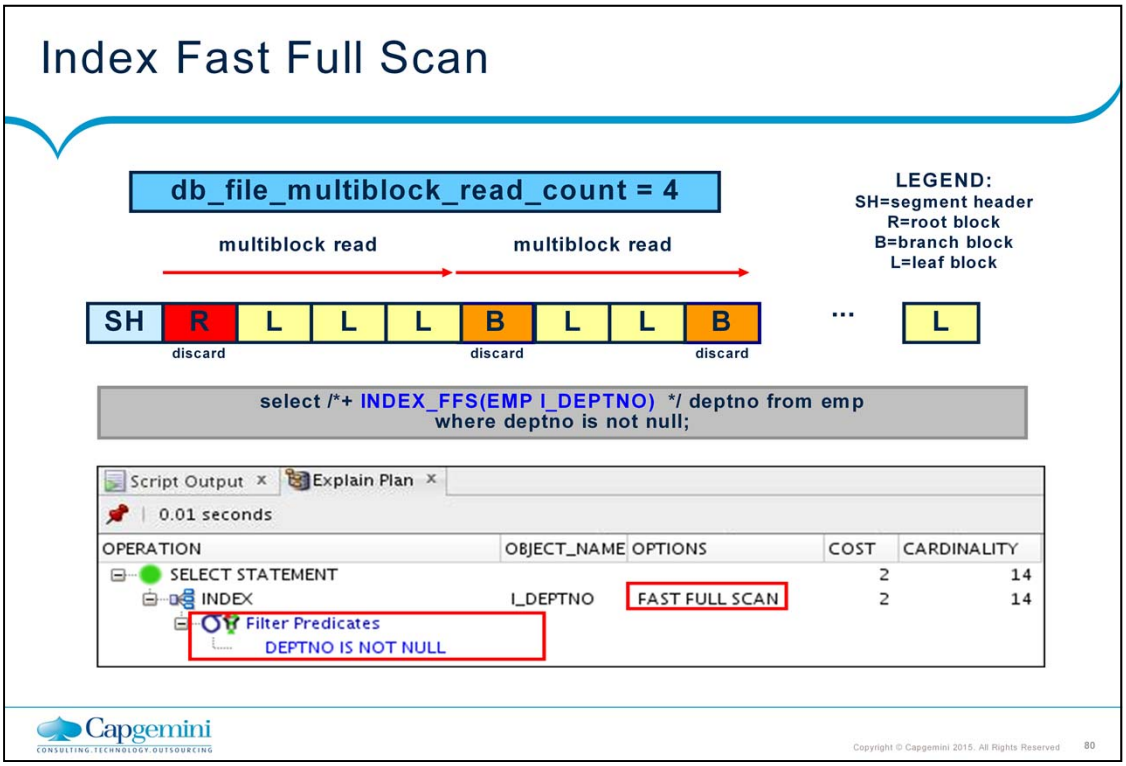
Index Range Scan: Function-Based

- A function-based index can be stored as B*-tree or bitmap structures. These indexes include columns that are either transformed by a function, such as the UPPER function, or included in an expression, such as col1 + col2. With a function-based index, you can store computation-intensive expressions in the index.
- Defining a function-based index on the transformed column or expression allows that data to be returned using the index when that function or expression is used in a WHERE clause or an ORDER BY clause. This allows the system to bypass computing the value of the expression when processing SELECT and DELETE statements. Therefore, a function-based index can be beneficial when frequently-executed SQL statements include transformed columns, or columns in expressions, in a WHERE or ORDER BY clause.
- For example, function-based indexes defined with the UPPER(column_name) or LOWER(column_name) keywords allow non-case-sensitive searches, such as shown in the slide.



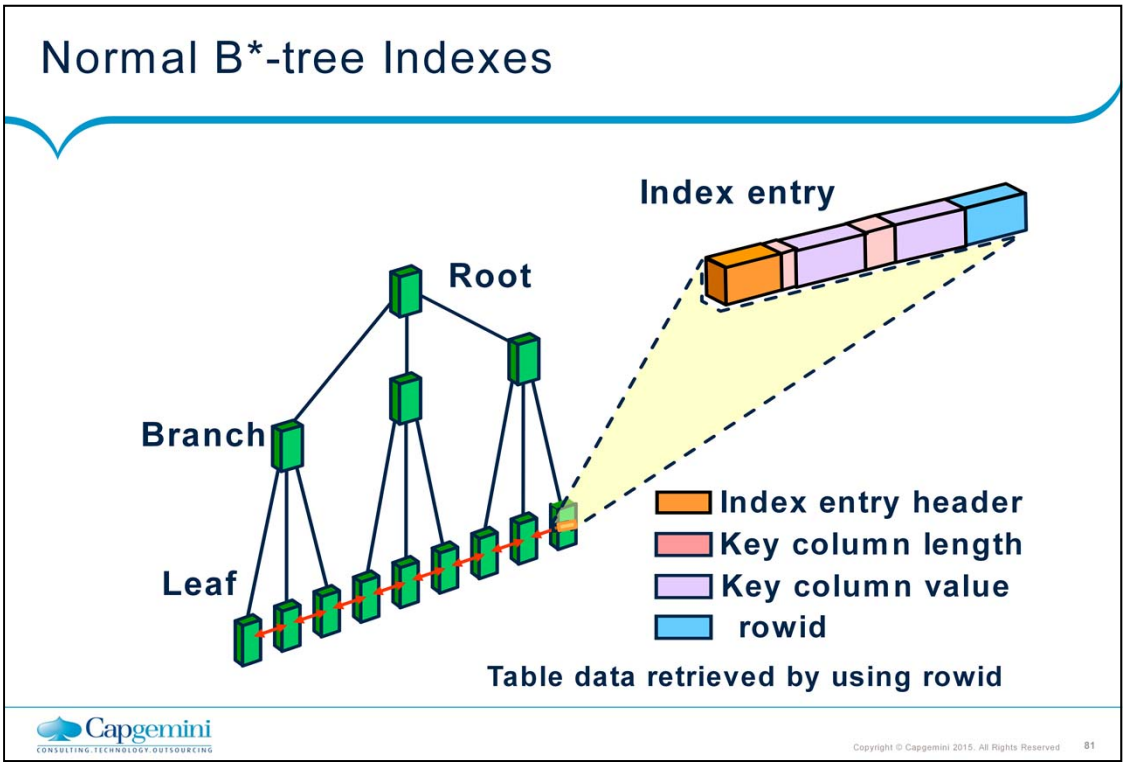
Index Full Scan

- A full scan is available if a predicate references one of the columns in the index. The predicate does not need to be an index driver (leading column). A full scan is also available when there is no predicate, if both the conditions are met:
  - All the columns in the table referenced in the query are included in the index.
  - At least one of the index columns is not null.
- A full scan can be used to eliminate a sort operation because the data is ordered by the index key.
- Note: An index full scan reads index using single-block input/output (I/O) (unlike a fast full index scan).



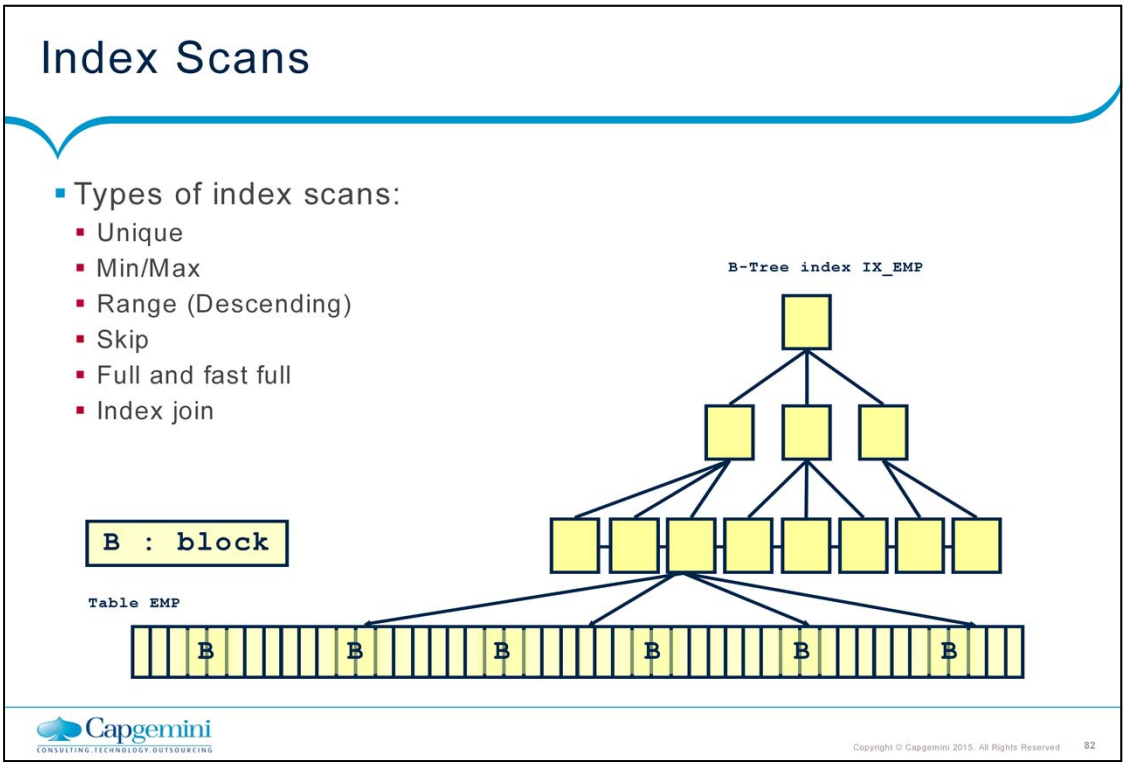
Index Fast Full Scan

- Index fast full scans are an alternative to full table scans when the index contains all the columns that are needed for the query and at least one column in the index key has a NOT NULL constraint. A fast full scan accesses the data in the index itself without accessing the table.
- It cannot be used to eliminate a sort operation because the data is not ordered by the index key. It can be used for the min/avg/sum aggregate functions. In this case, the optimizer must know that all table rows are represented in the index; at least one NOT NULL column.
- This operation reads the entire index using multiblock reads (unlike a full index scan). Fast full index scans cannot be performed against bitmap indexes. A fast full scan is faster than a normal full index scan because it can use multiblock I/O just as a table scan.
- You can specify fast full index scans with the OPTIMIZER_FEATURES_ENABLE initialization parameter or the INDEX_FFS hint as shown in the slide example.
- Note: Index fast full scans are used against an index when it is rebuilt offline.



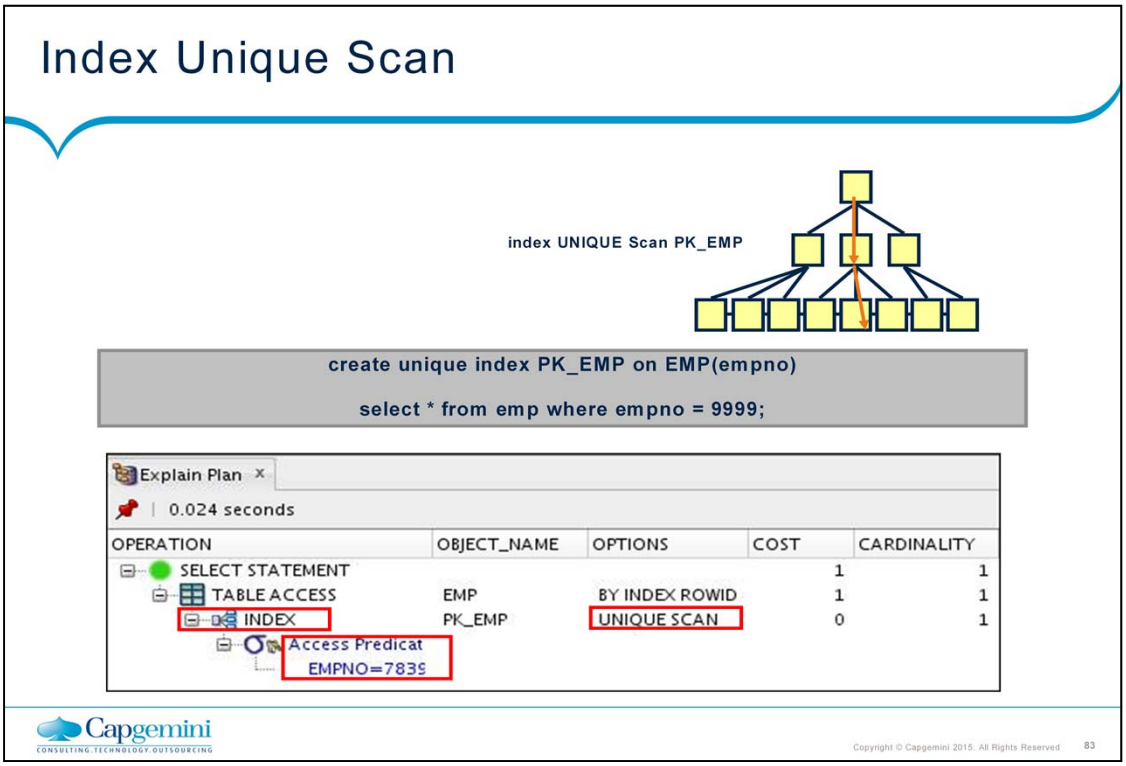
Normal B*-tree Indexes

- Each B*-tree index has a root block as a starting point. Depending on the number of entries, there are multiple branch blocks that can have multiple leaf blocks. The leaf blocks contain all values of the index plus ROWIDs that point to the rows in the associated data segment.
- Previous and next block pointers connect the leaf blocks so that they can be traversed from left to right (and vice versa).
  Indexes are always balanced, and they grow from the top down. In certain situations, the balancing algorithm can cause the B*-tree height to increase unnecessarily. It is possible to reorganize indexes. This is done by the ALTER INDEX … REBUILD | COALESCE command.
- The internal structure of a B*-tree index allows rapid access to the indexed values. The system can directly access rows after it has retrieved the address (the ROWID) from the index leaf blocks.
- Note: The maximum size of a single index entry is approximately one-half of the data block size.



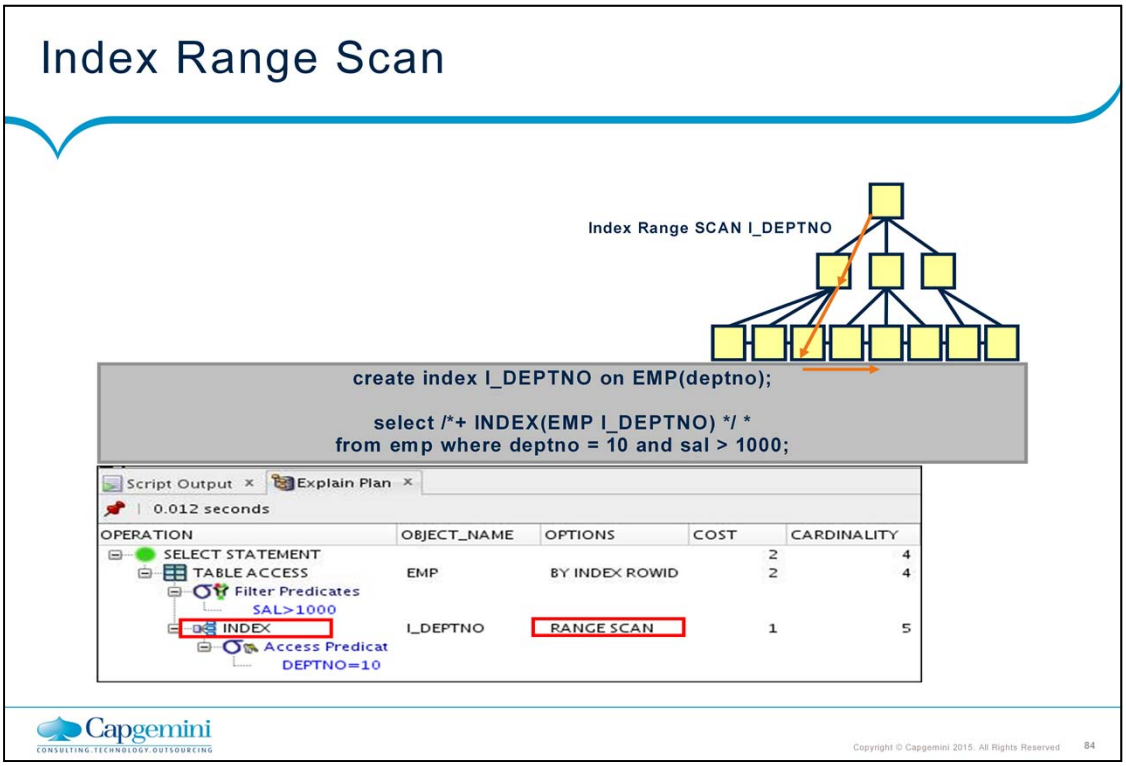
Index Scans

- An index scan can be one of the following types:
- A row is retrieved by traversing the index, using the indexed column values specified by the statement's WHERE clause. An index scan retrieves data from an index based on the value of one or more columns in the index. To perform an index scan, the system searches the index for the indexed column values accessed by the statement. If the statement accesses only columns of the index, the system reads the indexed column values directly from the index, rather than from the table.
- The index contains not only the indexed value, but also the rowids of rows in the table that have the value. Therefore, if the statement accesses other columns in addition to the indexed columns, the system can find the rows in the table by using either a table access by rowid or a cluster scan.
- Note: The graphic shows a case where four rows are retrieved from the table using their rowids obtained by the index scan.



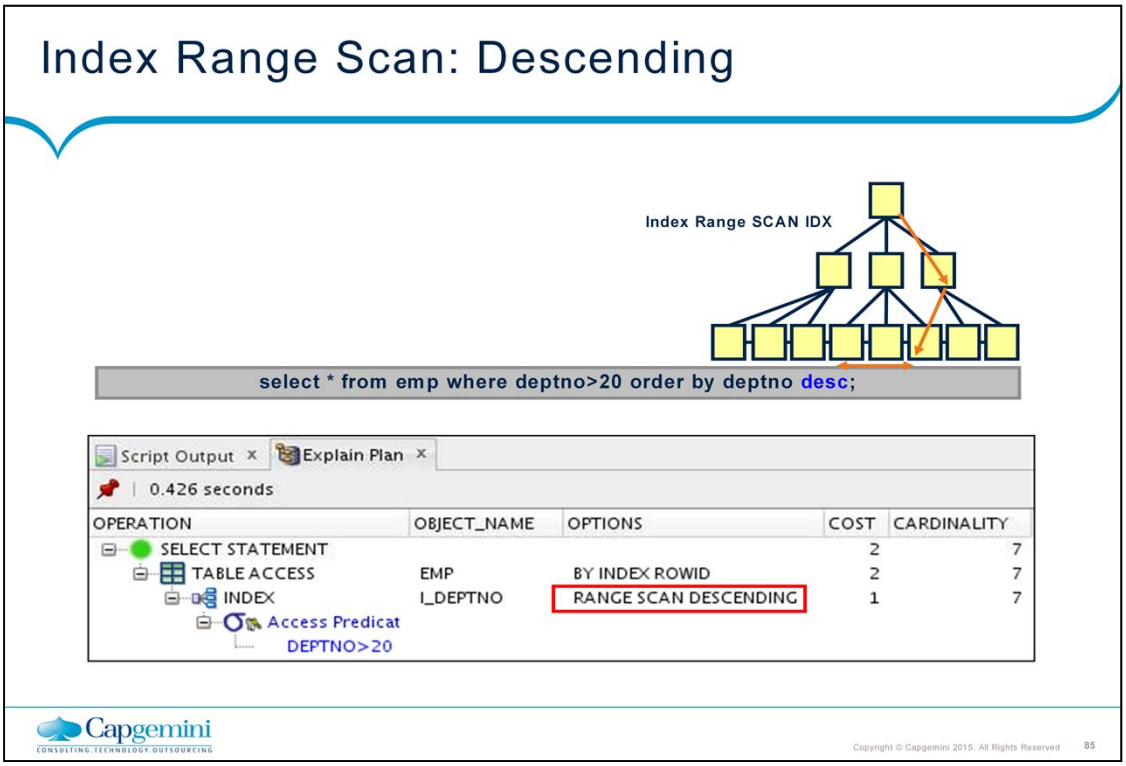
Index Unique Scan

- An index unique scan returns, at most, a single ROWID. The system performs a unique scan if a statement contains a UNIQUE or a PRIMARY KEY constraint that guarantees that only a single row is accessed. This access path is used when all the columns of a unique (B*-tree) index are specified with equality conditions.
- Key values and ROWIDs are obtained from the index, and table rows are obtained using ROWIDs.
- You can look for access conditions in the “Predicate Information” section of the execution plan (The execution plan is dealt with in detail in the lesson titled “Interpreting Execution Plans.”). Here the system accesses only matching rows for which EMPNO=9999.
- Note: Filter conditions filter rows after the fetch operation and output the filtered rows.



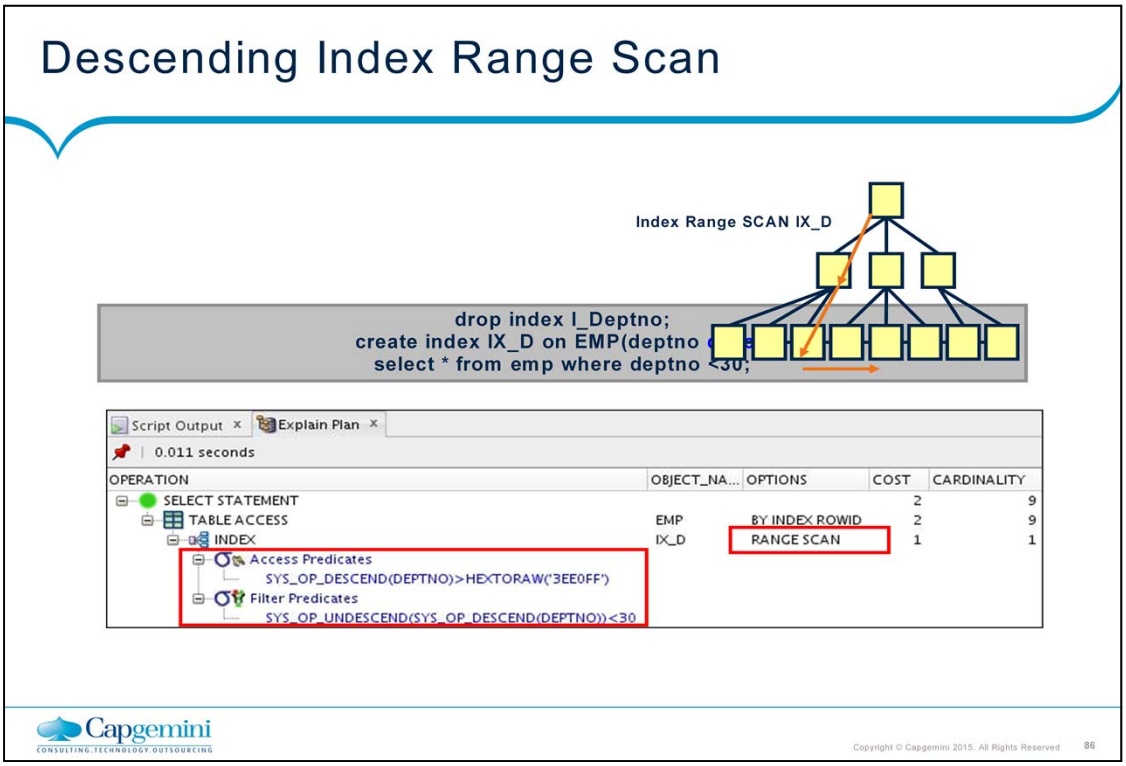
Index Range Scan

- An index range scan is a common operation for accessing selective data. It can be bounded (on both sides) or unbounded (on one or both sides). Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted in the ascending order by ROWID.
- The optimizer uses a range scan when it finds one or more leading columns of an index specified in conditions (the WHERE clause), such as col1 = :b1, col1 < :b1, col1 > :b1, and any combination of the preceding conditions.
- Wildcard searches (col1 like '%ASD') should not be in a leading position, as this does not result in a range scan.
- Range scans can use unique or nonunique indexes. Range scans can avoid sorting when index columns constitute the ORDER BY/GROUP BY clause and the indexed columns are NOT NULL as otherwise they are not considered.
- An index range scan descending is identical to an index range scan, except that the data is returned in the descending order. The optimizer uses index range scan descending when an order by descending clause can be satisfied by an index.
- In the example in the slide, using index I_DEPTNO, the system accesses rows for which EMP.DEPTNO=10. It gets their ROWIDs, fetches other columns from the EMP table, and finally, applies the EMP.SAL >1000 filter from these fetched rows to output the final result.



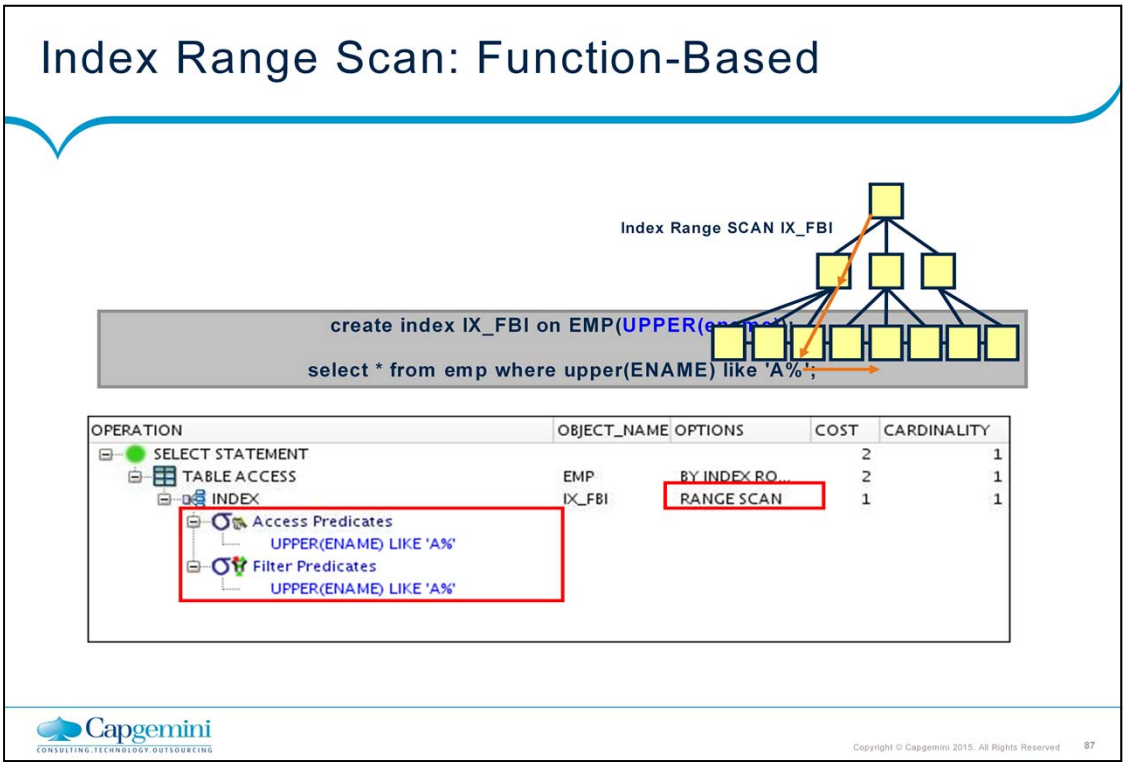
Index Range Scan: Descending

- In addition to index range scans in ascending order, which are described in the previous slide, the system is also able to scan indexes in the reverse order as illustrated by the graphic in the slide.
- The example retrieves rows from the EMP table by descending order on the DEPTNO column. You can see the DESCENDING operation row source for ID 2 in the execution plan that materialized this type of index scans.
- Note: By default an index range scan is done in the ascending order.



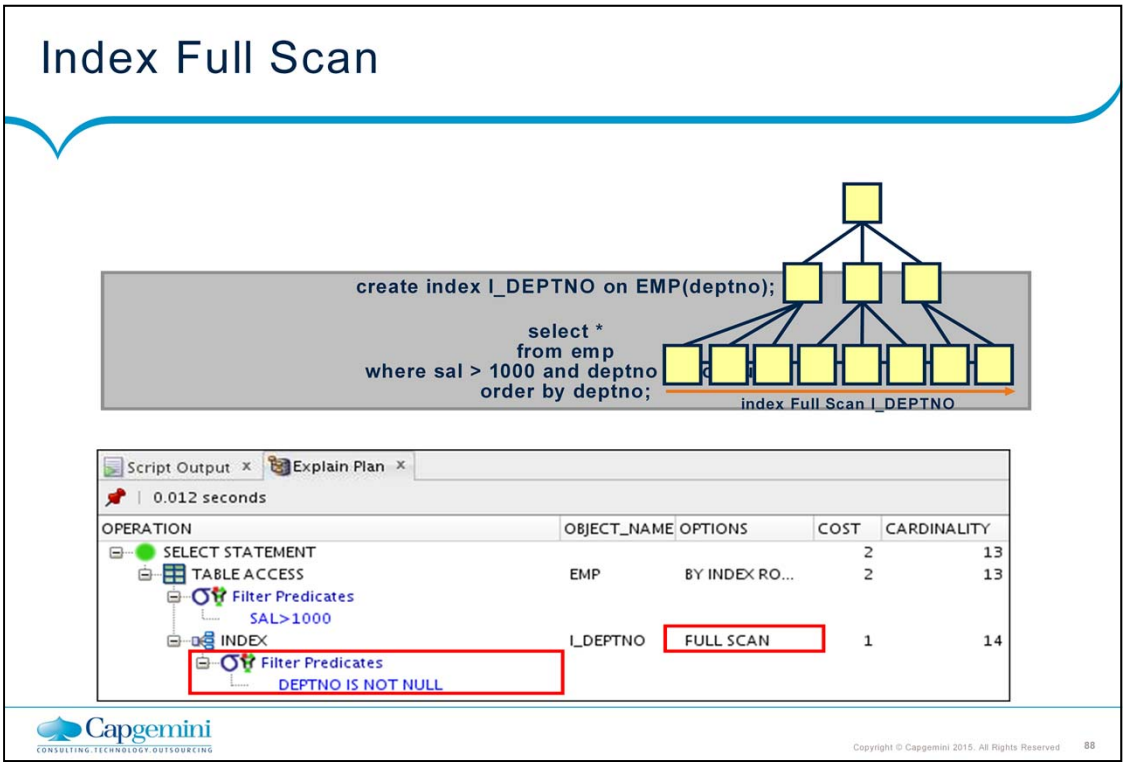
Descending Index Range Scan

- A descending index range scan is identical to an index range scan, except that the data is returned in descending order. Descending indexes allow for data to be sorted from “big to small” (descending) instead of “small to big” (ascending) in the index structure. Usually, this scan is used when ordering data in a descending order to return the most recent data first, or when seeking a value less than a specified value as in the example in the slide.
- The optimizer uses descending index range scan when an order by descending clause can be satisfied by a descending index.
- The INDEX_DESC(table_alias index_name) hint can be used to force this access path if possible.
- Note: The system treats descending indexes as function-based indexes. The columns marked DESC are stored in a special descending order in the index structure that is reversed again using the SYS_OP_UNDESCEND function.



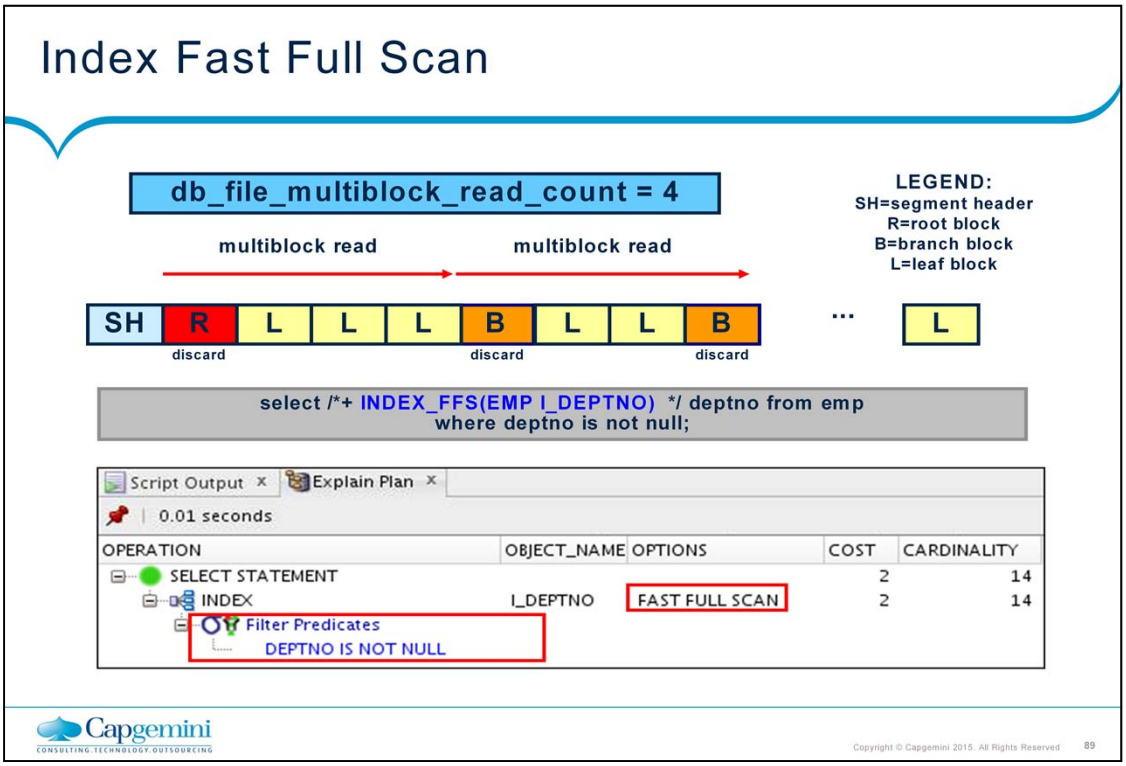
Index Range Scan: Function-Based

- A function-based index can be stored as B*-tree or bitmap structures. These indexes include columns that are either transformed by a function, such as the UPPER function, or included in an expression, such as col1 + col2. With a function-based index, you can store computation-intensive expressions in the index.
- Defining a function-based index on the transformed column or expression allows that data to be returned using the index when that function or expression is used in a WHERE clause or an ORDER BY clause. This allows the system to bypass computing the value of the expression when processing SELECT and DELETE statements. Therefore, a function-based index can be beneficial when frequently-executed SQL statements include transformed columns, or columns in expressions, in a WHERE or ORDER BY clause.
- For example, function-based indexes defined with the UPPER(column_name) or LOWER(column_name) keywords allow non-case-sensitive searches, such as shown in the slide.



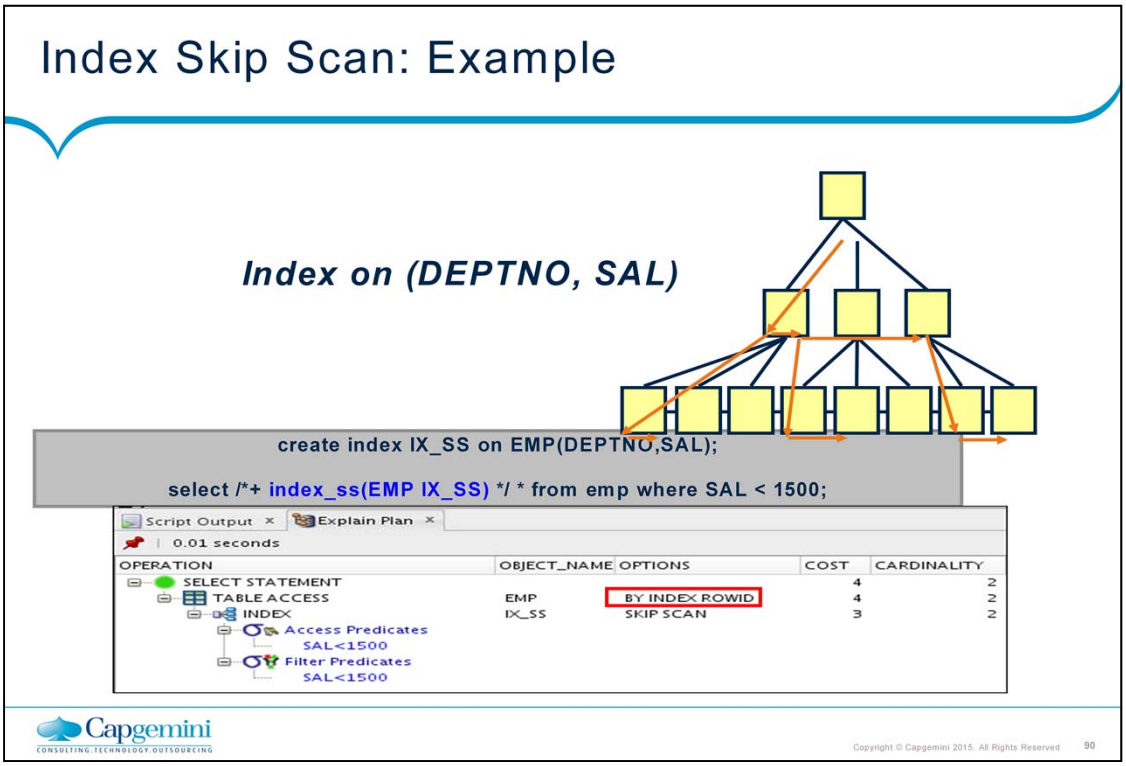
Index Full Scan

- A full scan is available if a predicate references one of the columns in the index. The predicate does not need to be an index driver (leading column). A full scan is also available when there is no predicate, if both the conditions are met:
  - All the columns in the table referenced in the query are included in the index.
  - At least one of the index columns is not null.
- A full scan can be used to eliminate a sort operation because the data is ordered by the index key.
- Note: An index full scan reads index using single-block input/output (I/O) (unlike a fast full index scan).



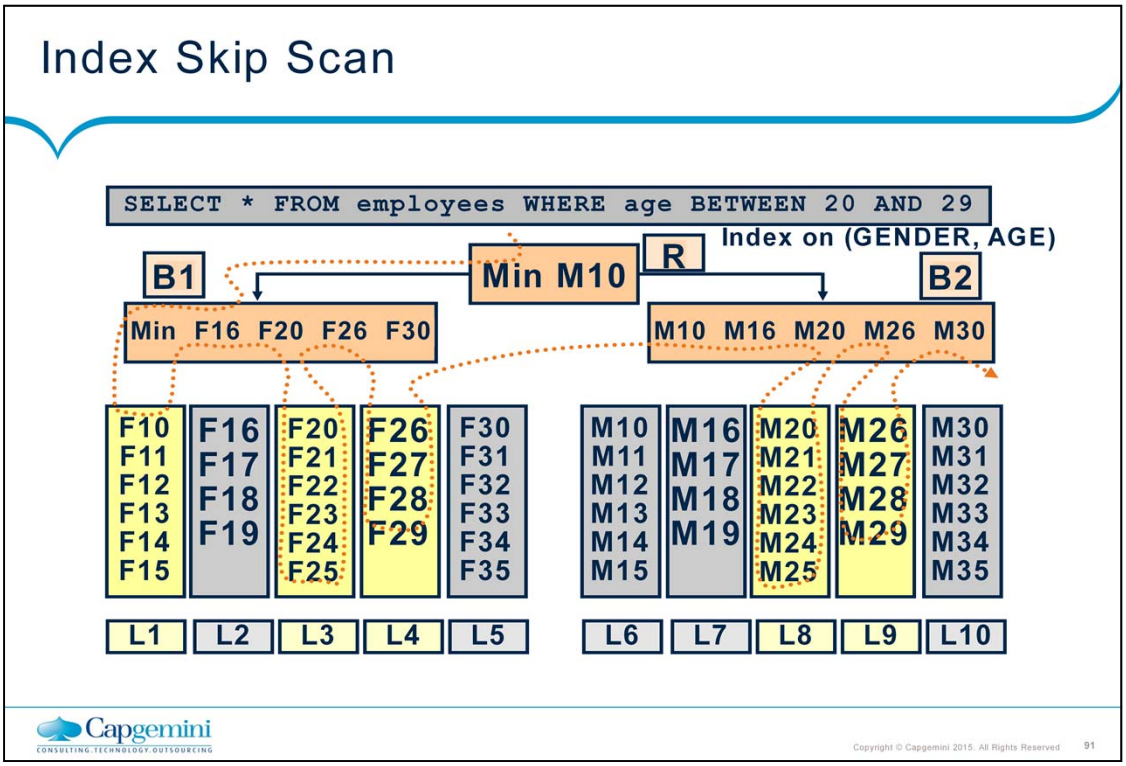
Index Fast Full Scan

- Index fast full scans are an alternative to full table scans when the index contains all the columns that are needed for the query and at least one column in the index key has a NOT NULL constraint. A fast full scan accesses the data in the index itself without accessing the table.
- It cannot be used to eliminate a sort operation because the data is not ordered by the index key. It can be used for the min/avg/sum aggregate functions. In this case, the optimizer must know that all table rows are represented in the index; at least one NOT NULL column.
- This operation reads the entire index using multiblock reads (unlike a full index scan). Fast full index scans cannot be performed against bitmap indexes. A fast full scan is faster than a normal full index scan because it can use multiblock I/O just as a table scan.
- You can specify fast full index scans with the OPTIMIZER_FEATURES_ENABLE initialization parameter or the INDEX_FFS hint as shown in the slide example.
- Note: Index fast full scans are used against an index when it is rebuilt offline.



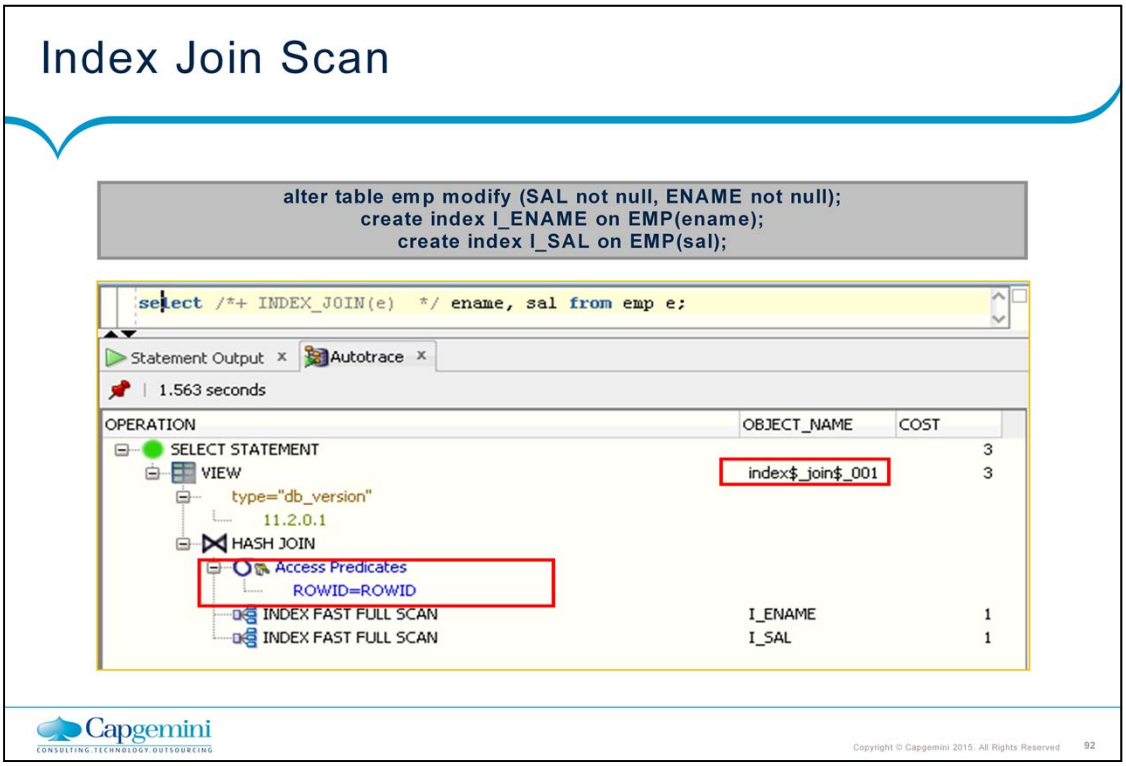
Index Skip Scan: Example

- The example in the slide finds employees who have salary less than 1500 using an index skip scan.
- It is assumed that there is a concatenated index on the DEPTNO and SAL columns.
- As you can see, the query does not have a predicate on the DEPTNO leading column. This leading column only has some discrete values, that is, 10, 20 and 30.
- Skip scanning lets a composite index be split logically into smaller subindexes. The number of logical subindexes is determined by the number of distinct values in the initial column.
- The system pretends that the index is really three little index structures hidden inside one big one. In the example, it is three index structures:
  - where deptno = 10
  - where deptno = 20
  - where deptno = 30
- The output is ordered by DEPTNO.
- Note: Skip scanning is advantageous if there are few distinct values in the leading column of the composite index and many distinct values in the nonleading key of the index.



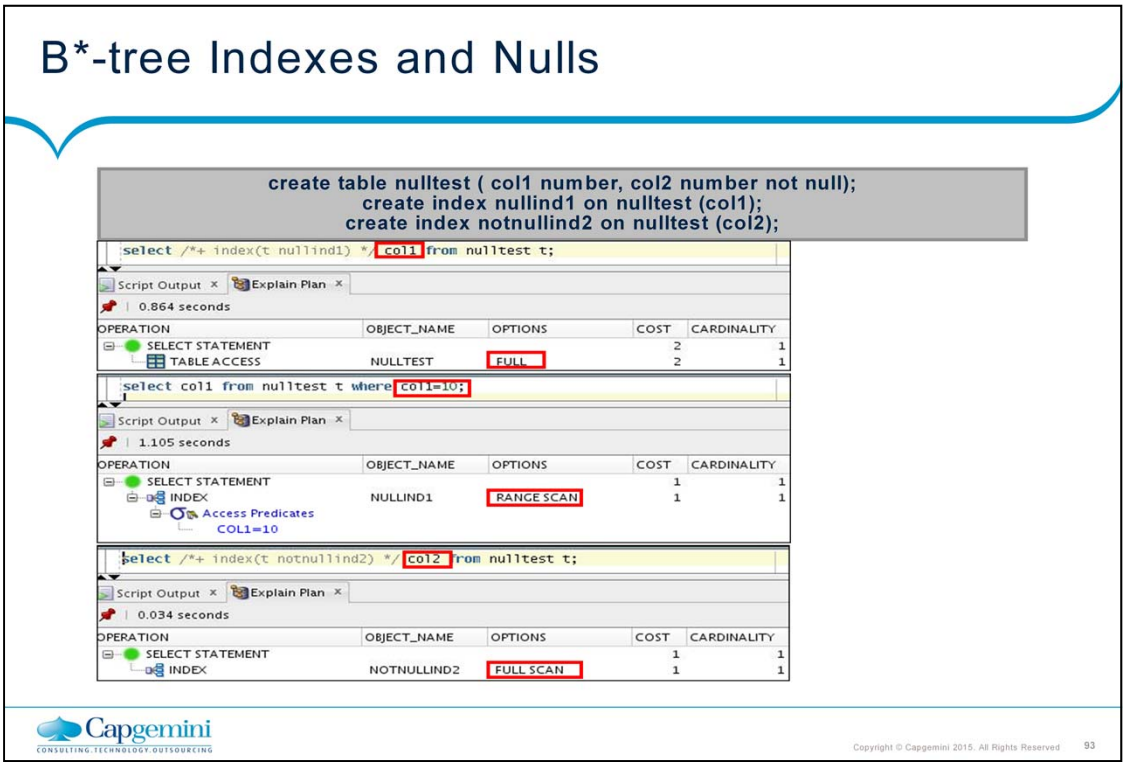
Index Skip Scan

- Index skip scans improve index scans by skipping blocks that could never contain keys matching the filter column values. Scanning index blocks is often faster than scanning table data blocks. Skip scanning can happen when the initial (leading) column of the composite index is not specified in a query. Suppose that there is a concatenated index on the GENDER and AGE columns in the EMPLOYEES table. This example illustrates how skip scanning is processed to answer the query in the slide.
- The system starts from the root of the index [R] and proceeds to the left branch block [B1]. From there, the system identifies a first entry to be F16, goes to the left leaf [L1], and starts to scan it because it could contain A25 (that is, where the “gender” is before “F” in the alphabet). The server identifies that this is not possible because the first entry is F10. It is thus not possible to find an entry such as A25 in this leaf, so it can be skipped.
- Backtracking to the first branch block [B1], the server identifies that the next subtree (F16) does not need to be scanned because the next entry in [B1] is F20. Because the server is certain that it is not possible to find a 25 between F16 and F20, the second leaf block [L2] can be skipped.
- Returning to [B1], the server finds that the next two entries have a common prefix of F2. This identifies possible subtrees to scan. The system knows that these subtrees are ordered by age.



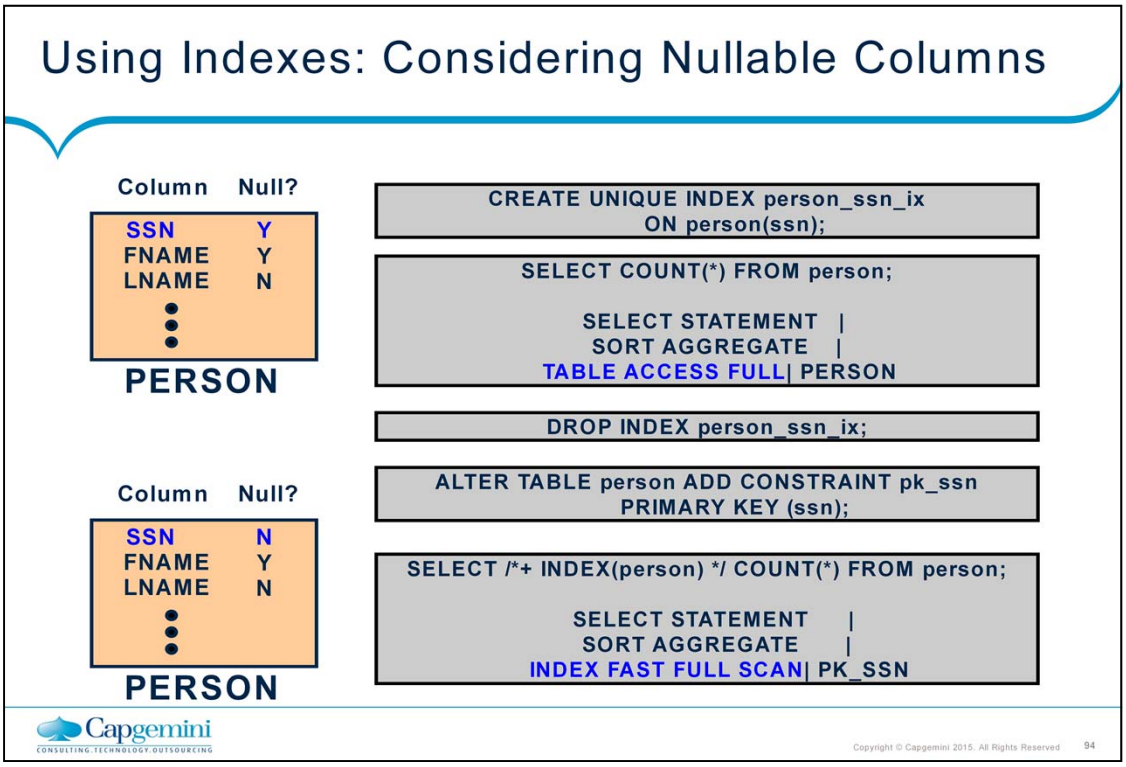
Index Join Scan

- An index join is a hash join of several indexes that together contain all the table columns that are referenced in the query. If an index join is used, no table access is needed because all the relevant column values can be retrieved from the indexes. An index join cannot be used to eliminate a sort operation.
- The index join is not a real join operation (note that the example is a single table query), but is built using index accesses followed by a join operation on rowid. The example in the slide assumes that you have two separate indexes on the ENAME and SAL columns of the EMP table.
- Note: You can specify an index join with the INDEX_JOIN hint as shown in the example.



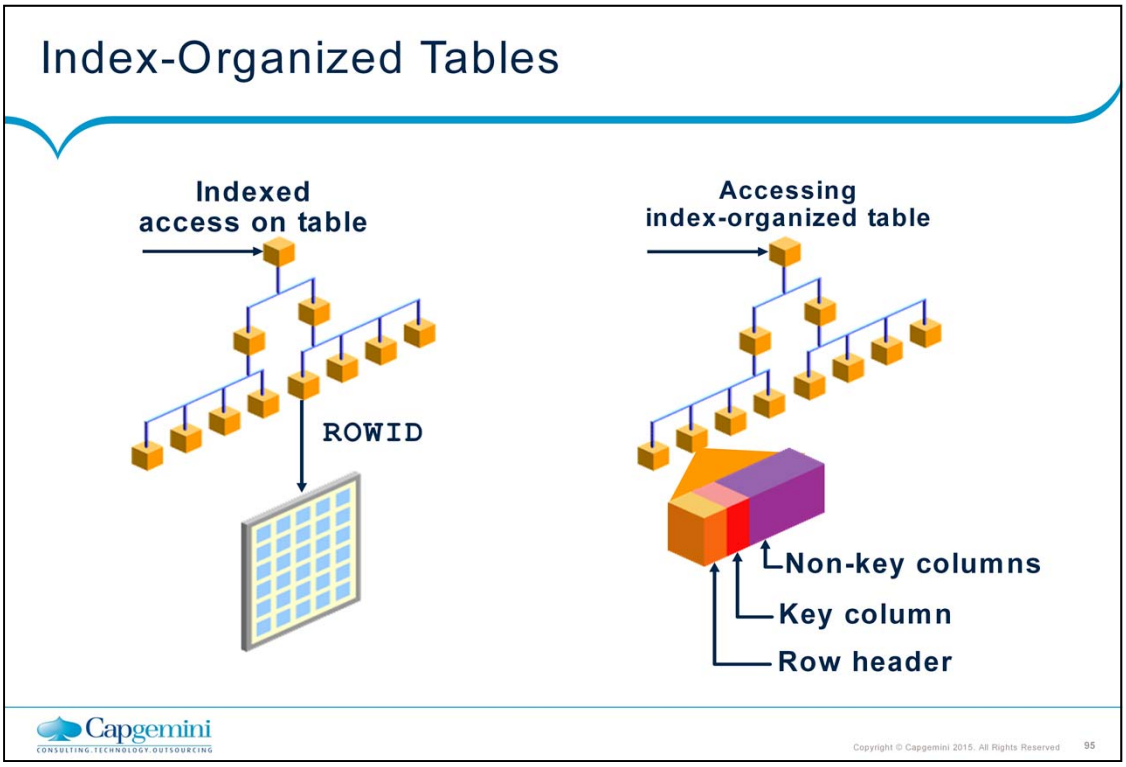
B*-tree Indexes and Nulls

- It is a common mistake to forget about nulls when dealing with B*-tree indexes. Single-column
- B*-tree indexes do not store null values and so indexes on nullable columns cannot be used to drive queries unless there is something that eliminates the null values from the query.
- In the slide example, you create a table containing a nullable column called COL1, and COL2, which cannot have null values. One index is built on top of each column.
- The first query, retrieves all COL1 values. Because COL1 is nullable, the index cannot be used without a predicate. Hinting the index on COL1 (nullind1) to force index utilization makes no difference because COL1 is nullable. Because you only search for COL1 values, there is no need to read the table itself.
- However, with the second query, the effect of the predicate against COL1 is to eliminate nulls from the data returned from the column. This allows the index to be used.
- The third query can directly use the index because the corresponding column is declared NOT NULL at table-creation time.
- Note: The index could also be used by forcing the column to return only NOT NULL values using the COL1 IS NOT NULL predicate.



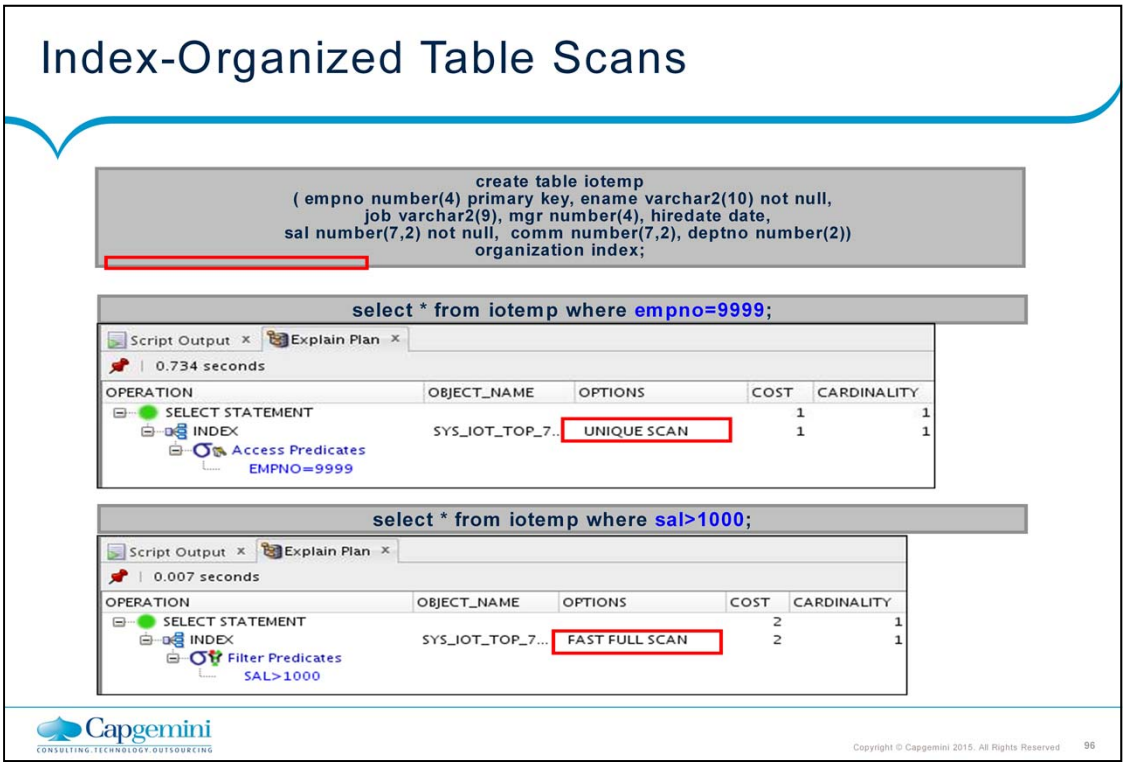
Using Indexes: Considering Nullable Columns

- Some queries look as if they should use an index to compute a simple count of rows in the table. This is typically more efficient that scanning the table. But the index to be used must not be built on a column that can contain null values. Single-column B*-tree indexes never store null values, so the rows are not represented in the index, and thus, do not contribute to the COUNT being computed, for example.
- In the example in the slide, there is a unique index on the SSN column of the PERSON table. The SSN column is defined as allowing null values, and creating a unique index on it does nothing to change that. This index is not used when executing the count query in the slide. Any rows with null for SSN are not represented in the index, so the count across the index is not necessarily accurate. This is one reason why it is better to create a primary key rather than a unique index. A primary key column cannot contain null values. In the slide, after the unique index is dropped in the place of designating a primary key, the index is used to compute the row count.
- Note: The PRIMARY KEY constraints combine a NOT NULL constraint and a unique constraint in a single declaration.



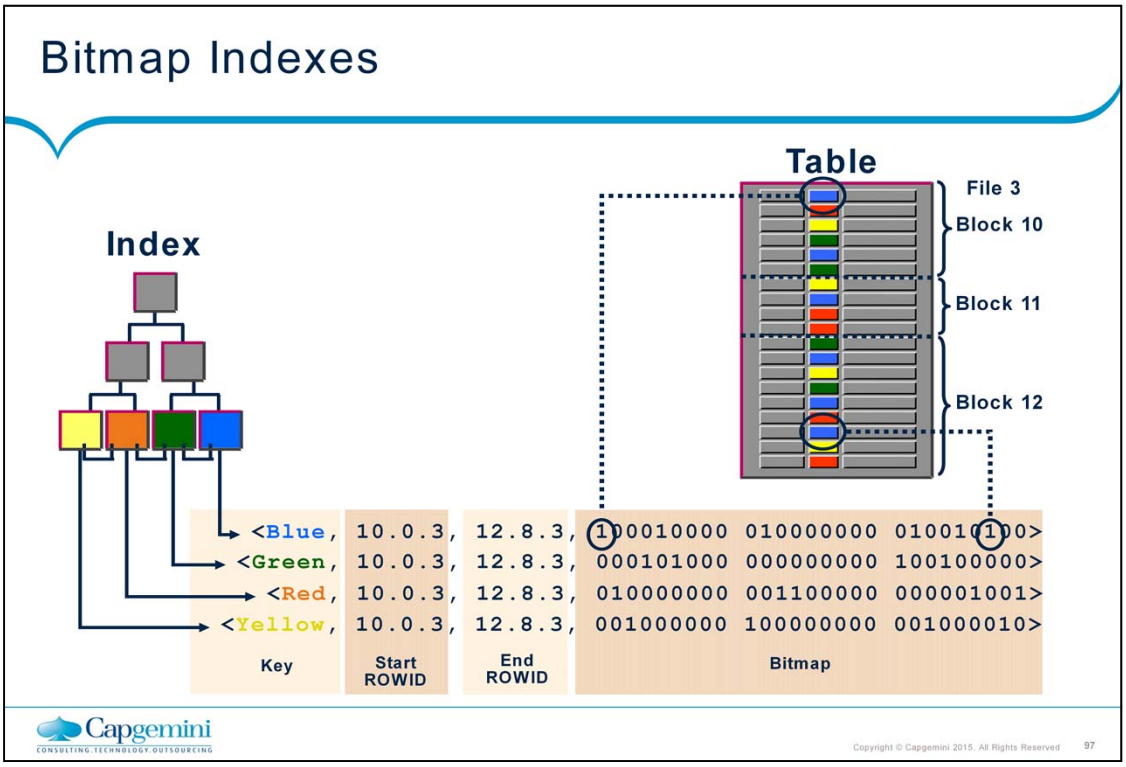
Index-Organized Tables

- An index-organized table (IOT) is a table physically stored in a concatenated index structure. The key values (for the table and the B*-tree index) are stored in the same segment. An IOT contains:
  - Primary key values
  - Other (non-key) column values for the row
- The B*-tree structure, which is based on the primary key of the table, is organized in the same way as an index. The leaf blocks in this structure contain the rows instead of the ROWIDs. This means that the rows in the IOT are always maintained in the order of the primary key.
- You can create additional indexes on IOTs. The primary key can be a composite key. Because large rows of an IOT can destroy the dense and efficient storage of the B*-tree structure, you can store part of the row in another segment, which is called an overflow area.
- Index-organized tables provide fast key-based access to table data for queries involving exact match and range searches. Changes to the table data result only in updating the index structure. Also, storage requirements are reduced because key columns are not duplicated in the table and index. The remaining non-key columns are stored in the index structure. IOTs are particularly useful when you use applications that must retrieve data based on a primary key and have only a few, relatively short non-key columns.
- Note: The descriptions mentioned here are correct if no overflow segments exist. Overflow segments should be used with long rows.



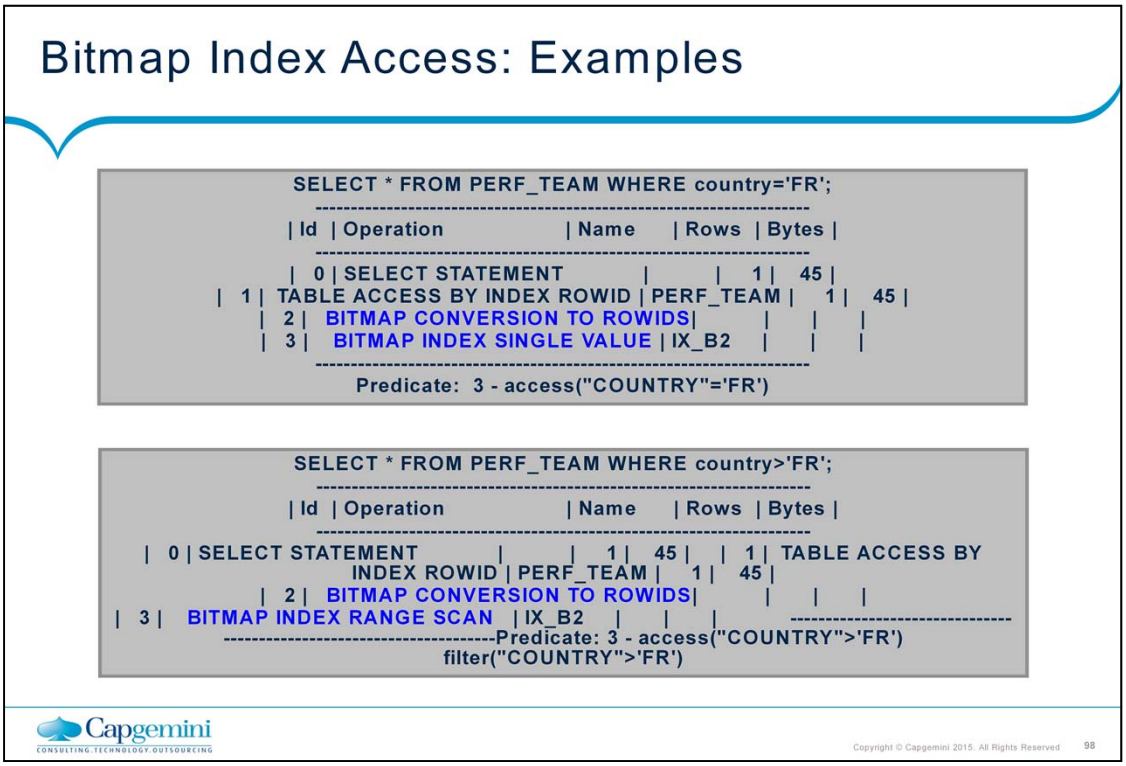
Index-Organized Table Scans

- Index-organized tables are just like indexes. They use the same access paths that you saw for normal indexes.
- The major difference from a heap-organized table is that there is no need to access both an index and a table to retrieve indexed data.
- Note: SYS_IOT_TOP_75664 is the system-generated name of the segment used to store the IOT structure. You can retrieve the link between the table name and the segment from USER_INDEXES with these columns: INDEX_NAME, INDEX_TYPE, TABLE_NAME.



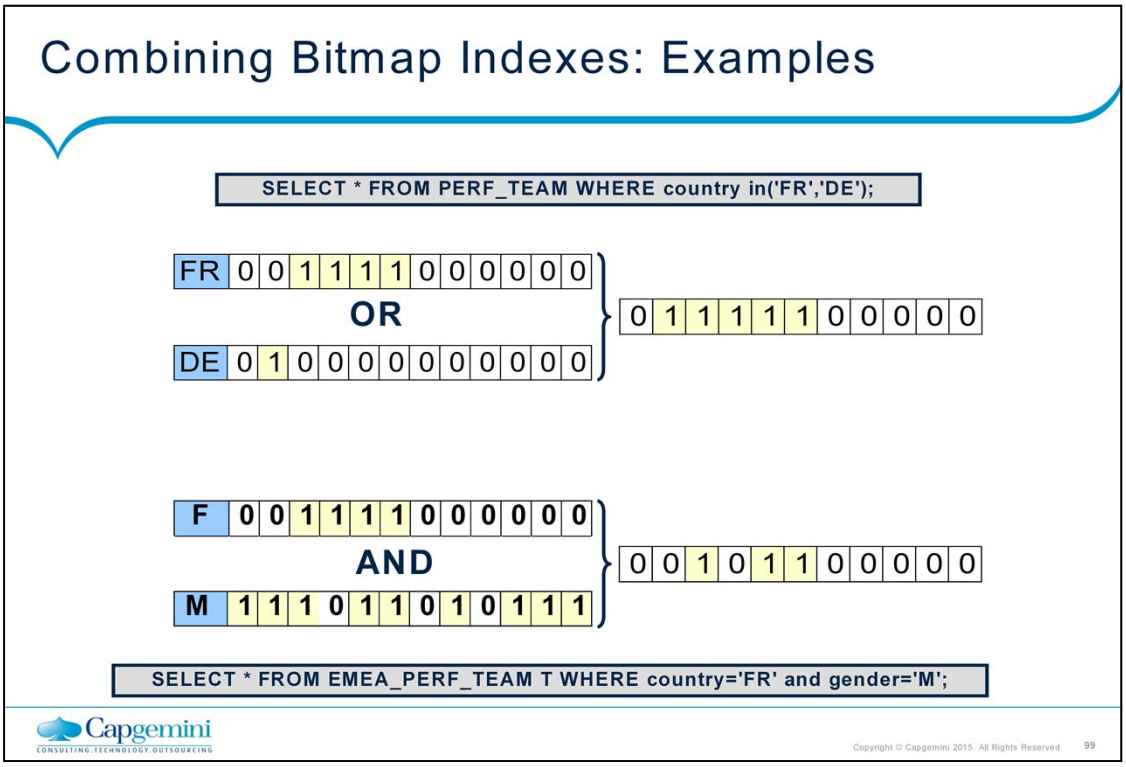
Bitmap Indexes

- In a B*-tree, there is a one-to-one relationship between an index entry and a row; an index entry points to a row. A bitmap index is organized as a B*-tree index but, with bitmap indexes, a single index entry uses a bitmap to point to many rows simultaneously. If a bitmap index involves more than one column, there is a bitmap for every possible combination. Each bitmap header stores start and end ROWIDs. Based on these values, the system uses an internal algorithm to map bitmaps onto ROWIDs. This is possible because the system knows the maximum possible number of rows that can be stored in a system block. Each position in a bitmap maps to a potential row in the table even if that row does not exist. The contents of that position in the bitmap for a particular value indicate whether that row has that value in the bitmap columns. The value stored is 1 if the row values match the bitmap condition; otherwise it is 0. Bitmap indexes are widely used in data warehousing environments. These environments typically have large amounts of data and ad hoc queries, but no concurrent data manipulation language (DML) transactions because when locking a bitmap, you lock many rows in the table at the same time. For such applications, bitmap indexing provides reduced response time for large classes of ad hoc queries, reduced storage requirements compared to other indexing techniques, dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory, and efficient maintenance during parallel DML and loads.



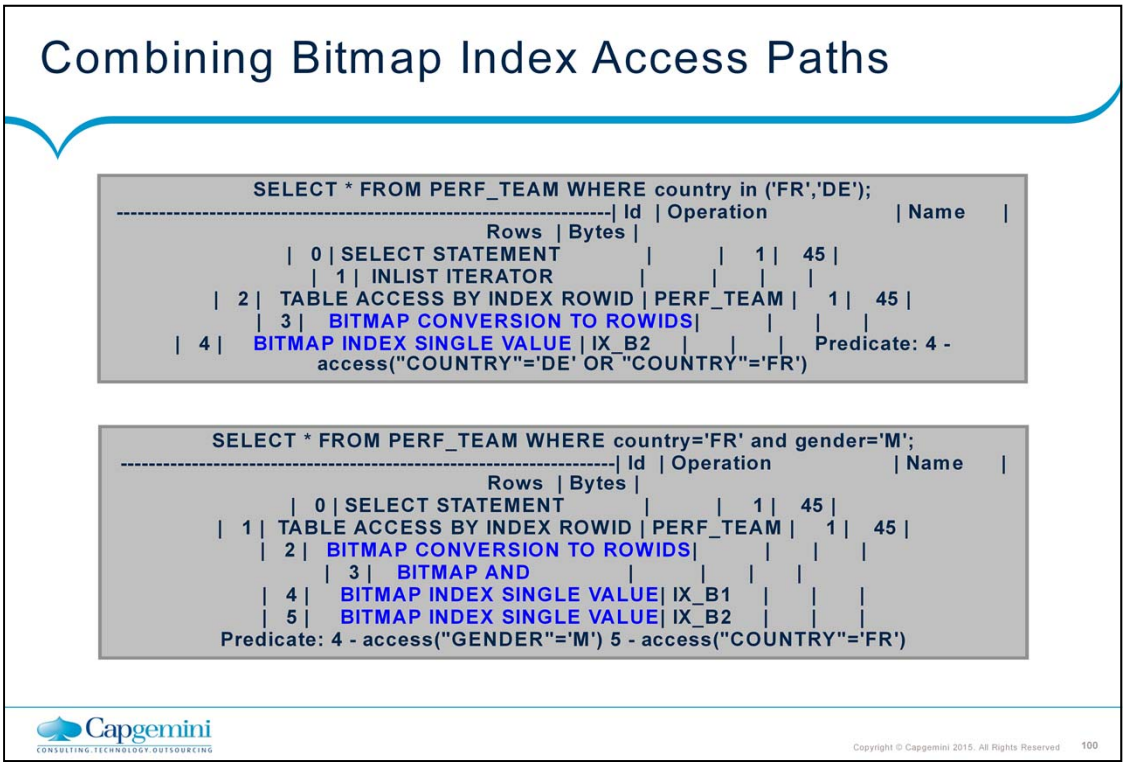
Bitmap Index Access: Examples

- The examples in the slide illustrate two possible access paths for bitmap indexes—BITMAP INDEX SINGLE VALUE and BITMAP INDEX RANGE SCAN—depending on the type of predicate you use in the queries.
- The first query scans the bitmap for country “FR” for positions containing a “1.” Positions with a “1” are converted into ROWIDs and have their corresponding rows returned for the query.
- In some cases (such as a query counting the number of rows with COUNTRY FR), the query might simply use the bitmap itself and count the number of 1s (not needing the actual rows).



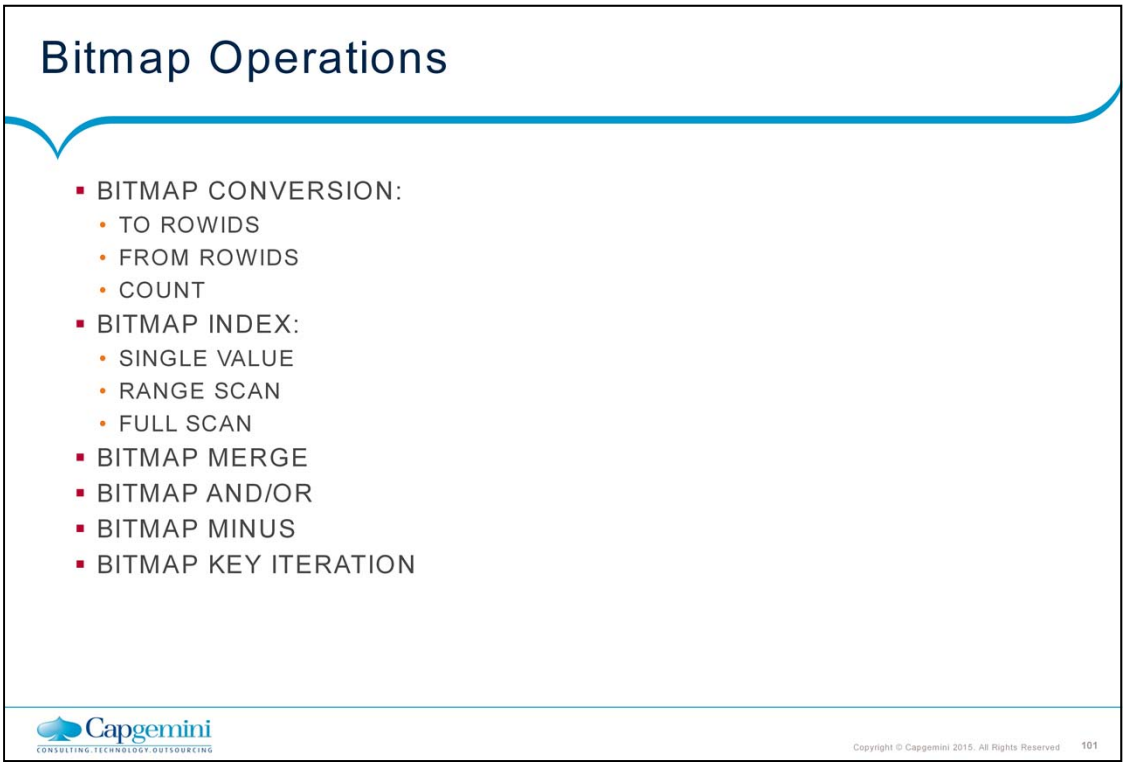
Combining Bitmap Indexes: Examples

- Bitmap indexes are the most effective for queries that contain multiple conditions in the WHERE clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically. As the bitmaps from bitmap indexes can be combined quickly, it is usually best to use single-column bitmap indexes.
- Due to fast bit-and, bit-minus, and bit-or operations, bitmap indexes are efficient when:
  - Using IN (value_list)
  - Predicates are combined with AND or OR



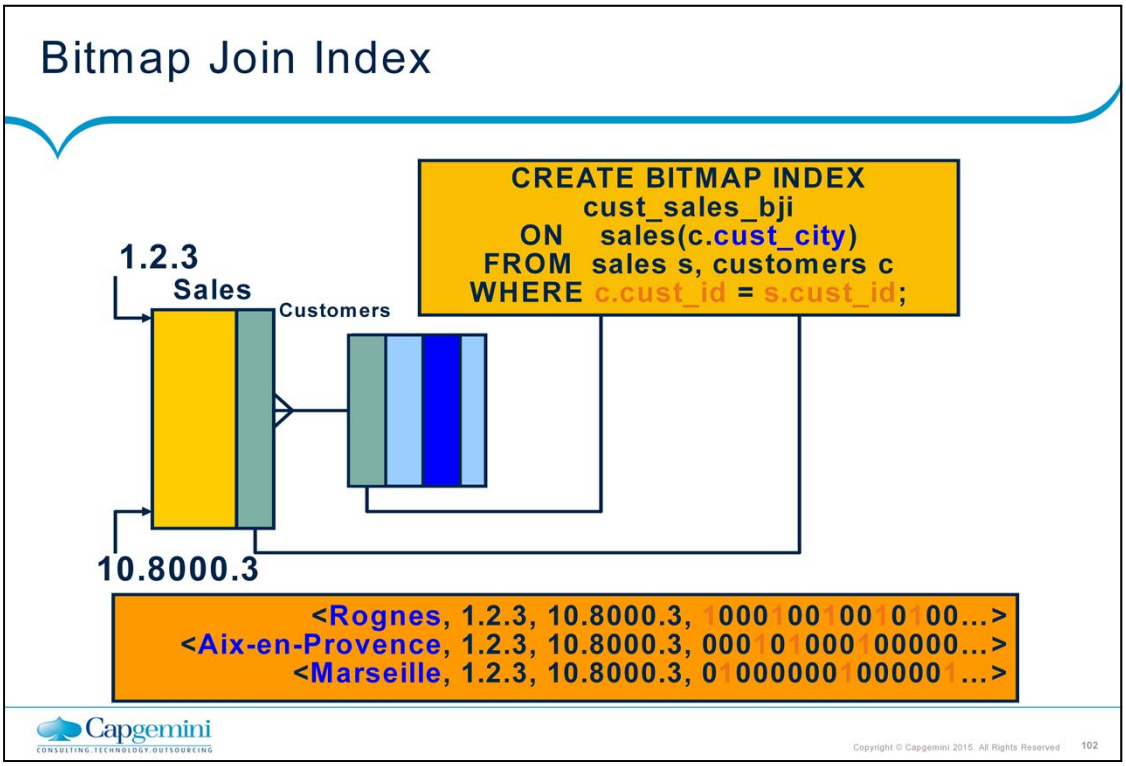
Combining Bitmap Index Access Paths

- Bitmap indexes can be used efficiently when a query combines several possible values for a column or when two separately-indexed columns are used.
- In some cases, a WHERE clause might reference several separately indexed columns as in the examples shown in the slide.
- If both the COUNTRY and GENDER columns have bitmap indexes, a bit-and operation on the two bitmaps quickly locates the rows that you look for. The more complex the compound WHERE clauses become, the more benefit you get from bitmap indexing.



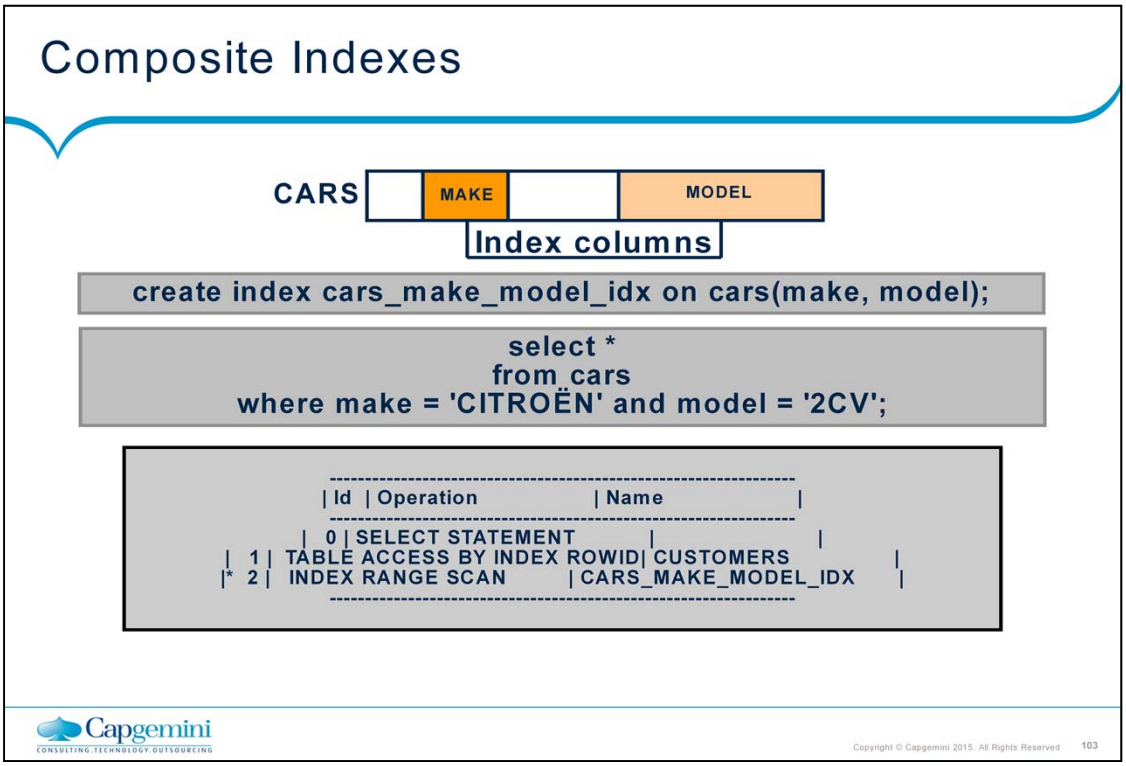
Bitmap Operations

- The slide summarizes all the possible bitmap operations. The following operations have not been explained so far:
  - BITMAP CONVERSION FROM ROWID: B*-tree index converted by the optimizer into BITMAP (cost is lower than other methods) to make these efficient bitmaps comparison operations available. After the bitmap comparison has been done, the resultant bitmap is converted back into ROWIDs (BITMAP CONVERSION TO ROWIDS) to perform the data lookup.
  - BITMAP MERGE merges several bitmaps resulting from a range scan into one bitmap.
  - BITMAP MINUS is a dual operator that takes the second bitmap operation and negates it by changing ones to zeros, and zeros to ones. The bitmap minus operation can then be performed as a BITMAP AND operation using this negated bitmap. This would typically be the case with the following combination of predicates: C1=2 and C2<>6.
  - BITMAP KEY ITERATION takes each row from a table row source and finds the corresponding bitmap from a bitmap index. This set of bitmaps is then merged into one bitmap in a BITMAP MERGE operation.



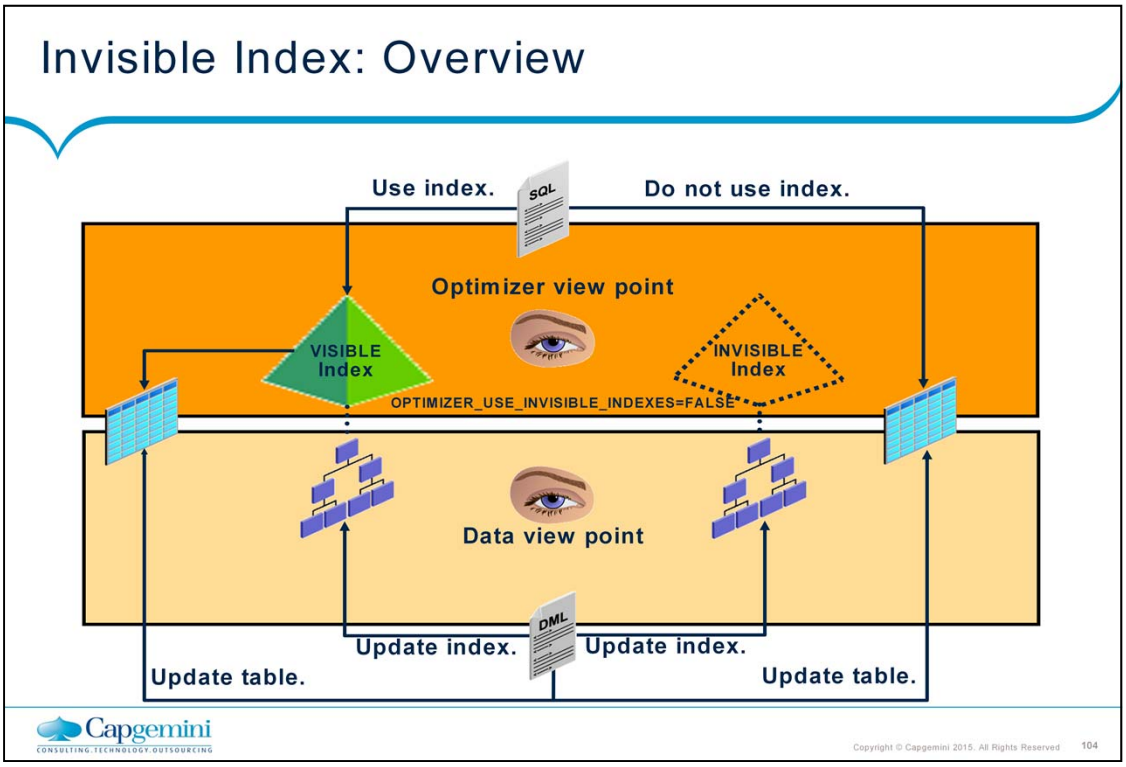
Bitmap Join Index

- In addition to a bitmap index on a single table, you can create a bitmap join index. A bitmap join index is a bitmap index for the join of two or more tables. A bitmap join index is a space-efficient way of reducing the volume of data that must be joined by performing the join in advance. Note: Bitmap join indexes are much more efficient in storage than materialized join views. For a row source example, see the lesson titled “Case Study: Star Transformation.”
- Here, you create a new bitmap join index named cust_sales_bji on the SALES table. The key of this index is the CUST_CITY column of the CUSTOMERS table. This example assumes that there is an enforced primary key constraint on CUSTOMERS to ensure that what is stored in the bitmap reflects the reality of the data in the tables. The CUST_ID column is the primary key of CUSTOMERS, but is also a foreign key inside SALES, although not required.
- The FROM and WHERE clause in the CREATE statement allow the system to make the link between the two tables. They represent the join condition between the two tables. The middle part of the graphic shows you a theoretical implementation of this bitmap join index. Each entry or key in the index represents a possible city found in the CUSTOMERS table. A bitmap is then associated to one particular key. Each bit in a bitmap corresponds to one row in the SALES table. In the first key in the slide (Rognes), you see that the first row in the SALES table corresponds to a product sold to a Rognes customer, while the second bit is not a product sold to a Rognes customer. By storing the result of a join, the join can be avoided completely for SQL statements using a bitmap join index.



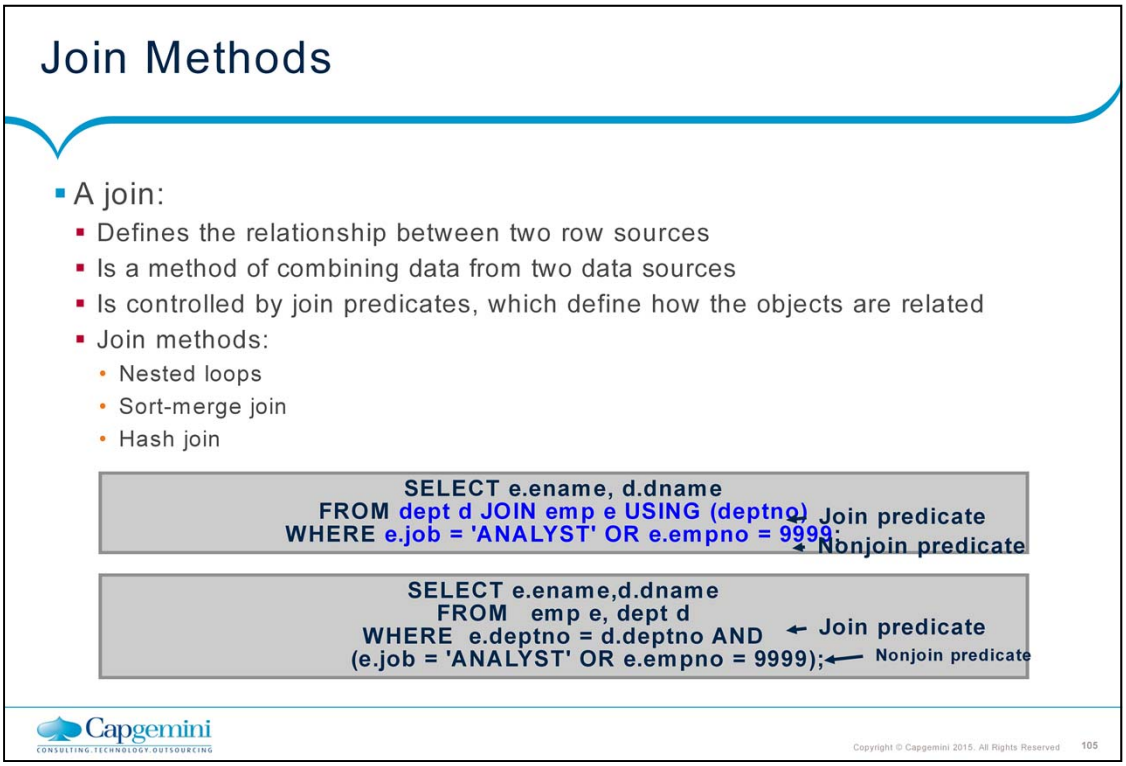
Composite Indexes

- A composite index is also referred to as a concatenated index because it concatenates column values together to form the index key value. In the illustration in the slide, the MAKE and MODEL columns are concatenated together to form the index. It is not required that the columns in the index are adjacent. And, you can include up to 32 columns in the index, unless it is a bitmap composite index, in which case the limit is 30.
- Composite indexes can provide additional advantages over single-column indexes:
  - Improved selectivity: Sometimes two or more columns or expressions, each with poor selectivity, can be combined to form a composite index with higher selectivity.
  - Reduced I/O: If all columns selected by a query are in a composite index, the system can return these values from the index without accessing the table.
- A composite index is mainly useful when you often have a WHERE clause that references all, or the leading portion of the columns in the index. If some keys are used in WHERE clauses more frequently, and you decided to create a composite index, be sure to create the index so that the more frequently selected keys constitute a leading portion for allowing the statements that use only these keys to use the index.
- Note: It is also possible for the optimizer to use a concatenated index even though your query does not reference a leading part of that index. This is possible since index skip scans and fast full scans were implemented.



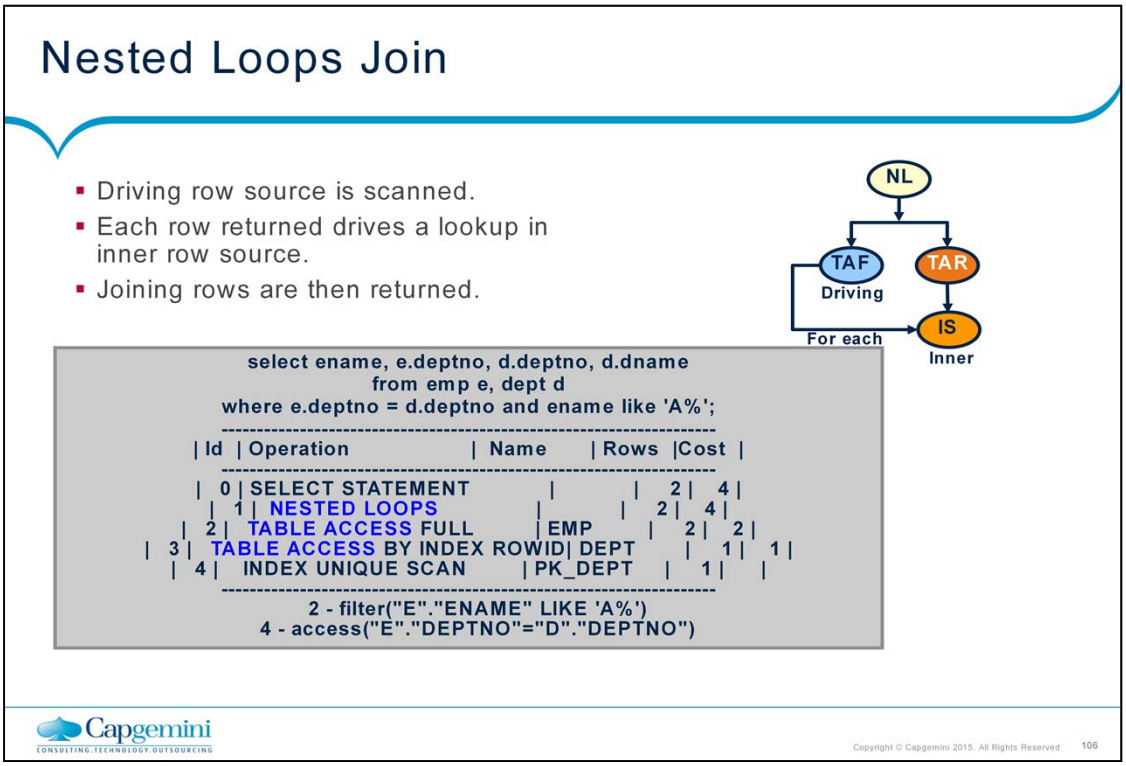
Invisible Index: Overview

- An invisible index is an index that is ignored by the optimizer unless you explicitly set the OPTIMIZER_USE_INVISIBLE_INDEXES initialization parameter to TRUE at the session or system level. The default value for this parameter is FALSE.
- Making an index invisible is an alternative to making it unusable or dropping it. Using invisible indexes, you can perform the following actions:
  - Test the removal of an index before dropping it.
  - Use temporary index structures for certain operations or modules of an application without affecting the overall application.
- Unlike unusable indexes, an invisible index is maintained during DML statements.



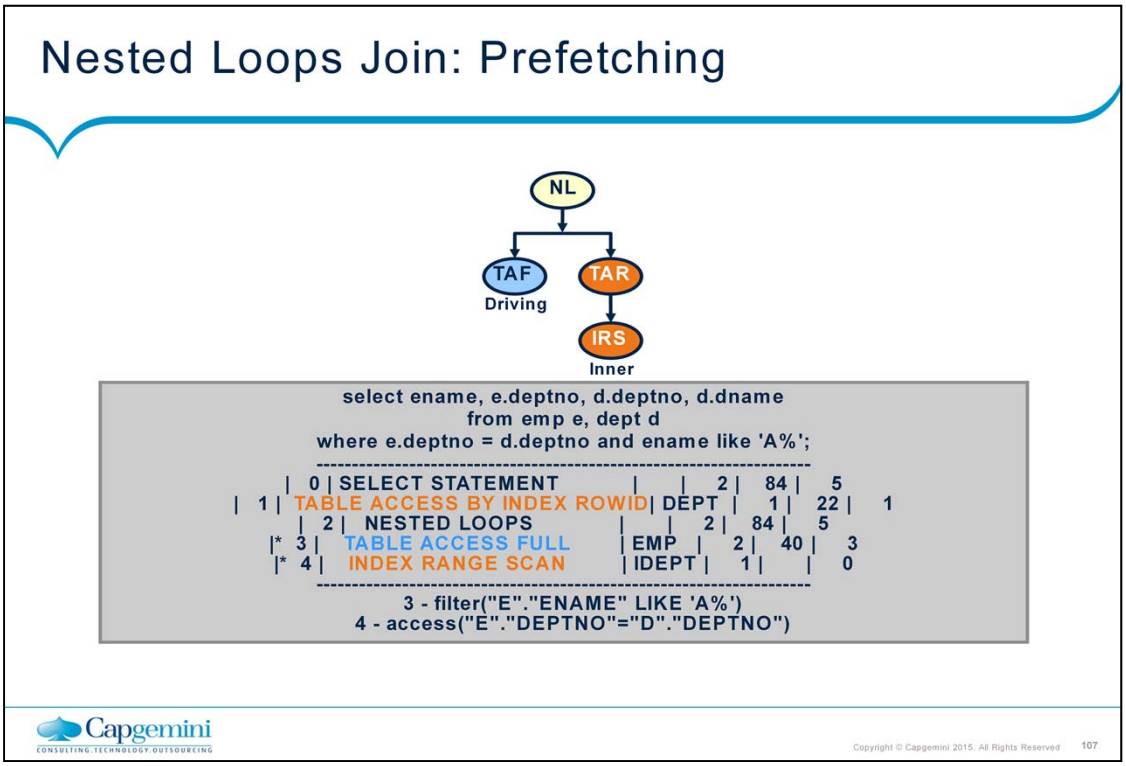
Join Methods

- A row source is a set of data that can be accessed in a query. It can be a table, an index, a nonmergeable view, or even the result set of a join tree consisting of many different objects.
- A join predicate is a predicate in the WHERE clause that combines the columns of two of the tables in the join.
- A nonjoin predicate is a predicate in the WHERE clause that references only one table.
- A join operation combines the output from two row sources (such as tables or views) and returns one resulting row source (data set). The optimizer supports different join methods such as the following:
  - Nested loop join: Useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the second table
  - Sort-merge join: Can be used to join rows from two independent sources. Hash joins generally perform better than sort-merge joins. On the other hand, sort-merge joins can perform better than hash joins if one or two row sources are already sorted.
  - Hash join: Used for joining large data sets. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows. This method is best used when the smaller table fits in the available memory. The cost is then limited to a single read pass over the data for the two tables.
- Note: The slide shows you the same query using both the American National Standards Institute (ANSI) and non-ANSI join syntax. The ANSI syntax is the first example.



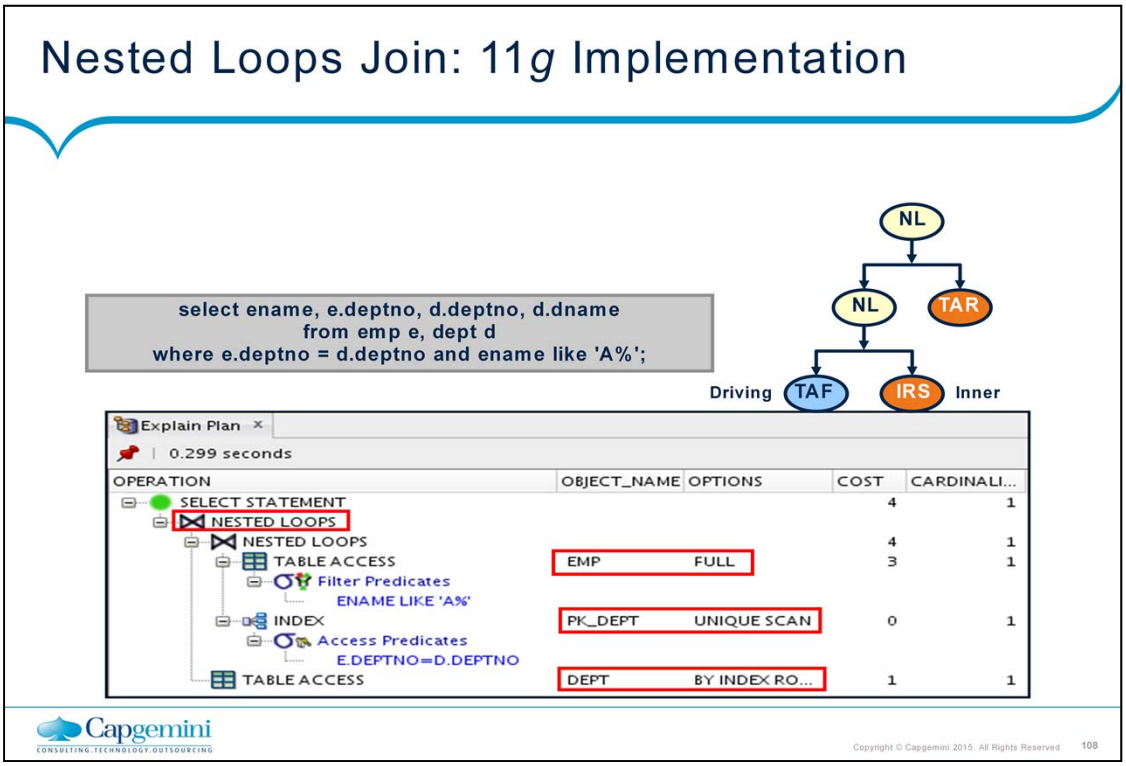
Nested Loops Join

- In the general form of the nested loops join, one of the two tables is defined as the outer table, or the driving table. The other table is called the inner table, or the right-hand side.
- For each row in the outer (driving) table that matches the single table predicates, all rows in the inner table that satisfy the join predicate (matching rows) are retrieved. If an index is available, it can be used to access the inner table by rowid.
- Any nonjoin predicates on the inner table are considered after this initial retrieval, unless a composite index combining both the join and the nonjoin predicate is used.
- The code to emulate a nested loop join might look as follows:
- for r1 in (select rows from EMP that match single table predicate) loop
- for r2 in (select rows from DEPT that match current row from EMP) loop
- output values from current row of EMP and current row of DEPT
- end loop
- end loop
- The optimizer uses nested loop joins when joining small number of rows, with a good driving condition between the two tables. You drive from the outer loop to the inner loop, so the order of tables in the execution plan is important. Therefore, you should use other join methods when two independent row sources are joined.



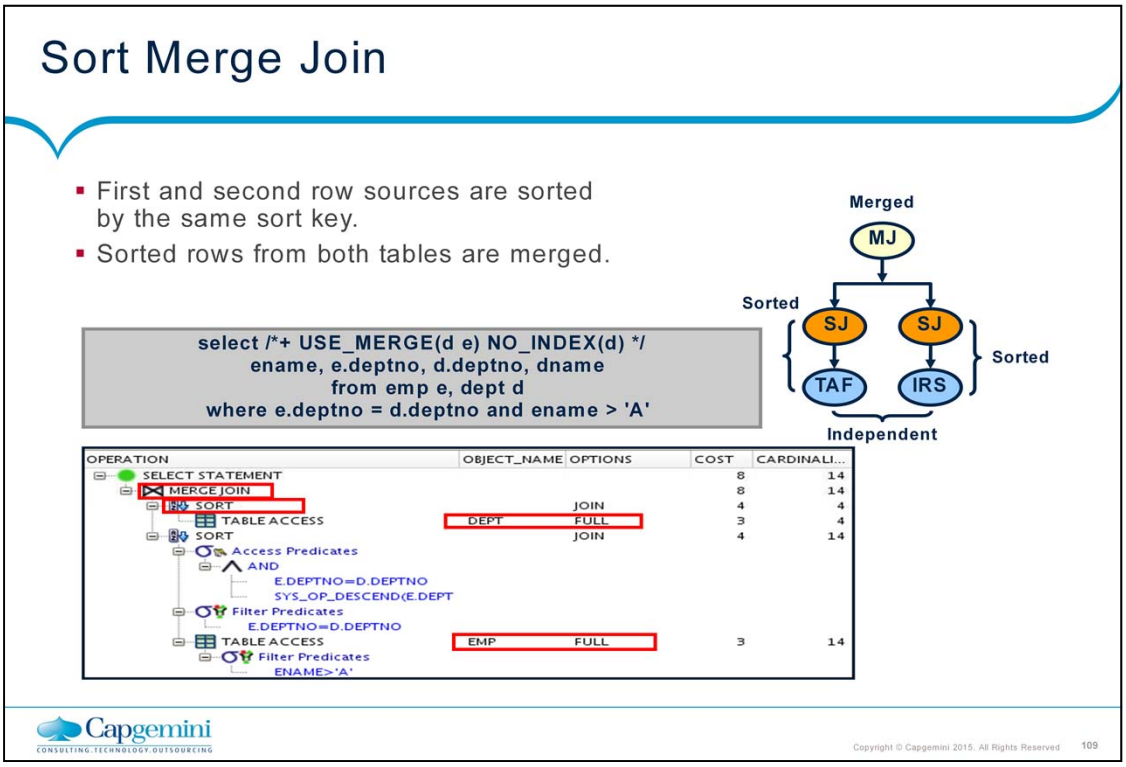
Nested Loops Join: Prefetching

- Oracle 9iR2 introduced a mechanism called nested loops prefetching. The idea is to improve I/O utilization, therefore response time, of index access with table lookup by batching rowid lookups into parallel block reads.
- This change to the plan output is not considered a different execution plan. It does not affect the join order, join method, access method, or parallelization scheme.
- This optimization is only available when the inner access path is index range scan and not if the inner access path is index unique scan.
- The prefetching mechanism is used by table lookup. When an index access path is chosen and the query cannot be satisfied by the index alone, the data rows indicated by the ROWID also must be fetched. This ROWID to data row access (table lookup) is improved using data block prefetching, which involves reading an array of blocks which are pointed at by an array of qualifying ROWIDs.
- Without data block prefetching, accessing a large number of rows using a poorly clustered B*-tree index could be expensive. Each row accessed by the index would likely be in a separate data block and thus would require a separate I/O operation.
- With data block prefetching, the system delays data blocks reads until multiple rows specified by the underlying index are ready to be accessed and then retrieves multiple data blocks simultaneously, rather than reading a single data block at a time.



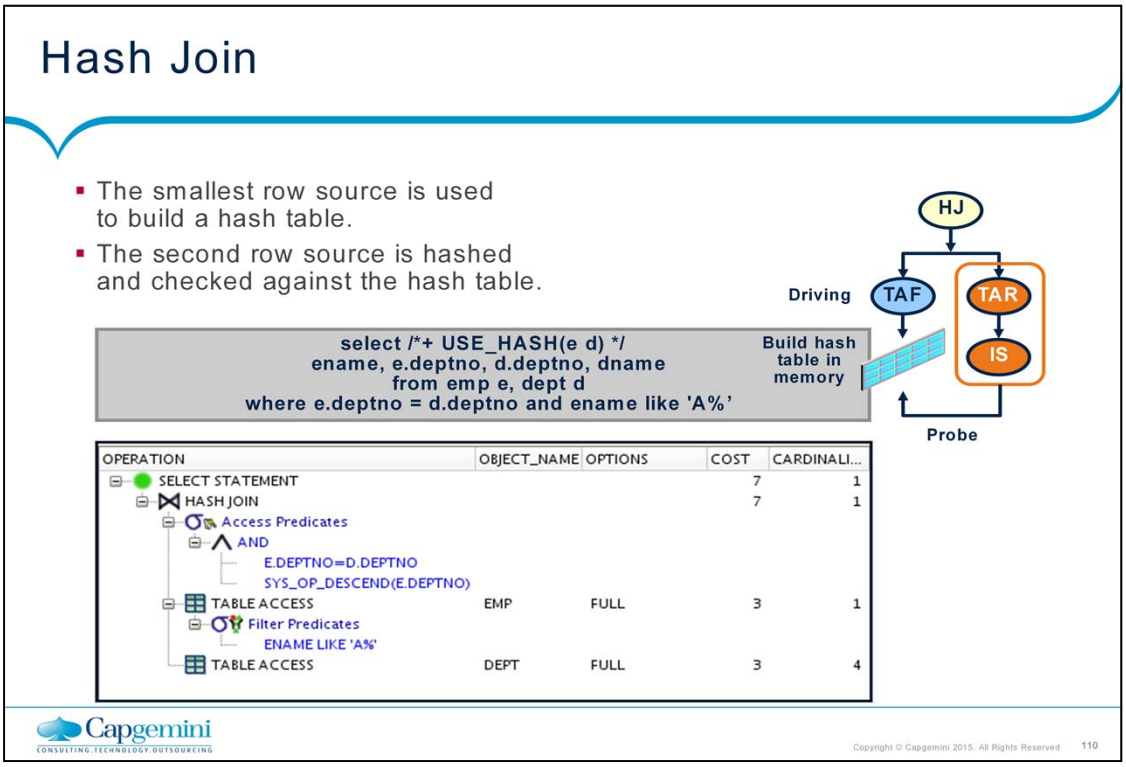
Nested Loops Join: 11g Implementation

- Oracle Database 11g introduces a new way of performing joins with NESTED LOOPS operators. With this NESTED LOOPS implementation, the system first performs a NESTED LOOPS join between the other table and the index. This produces a set of ROWIDs that you can use to look up the corresponding rows from the table with the index. Instead of going to the table for each ROWID produced by the first NESTED LOOPS join, the system batches up the ROWIDs and performs a second NESTED LOOPS join between the ROWIDs and the table. This ROWID batching technique improves performance as the system only reads each block in the inner table once.



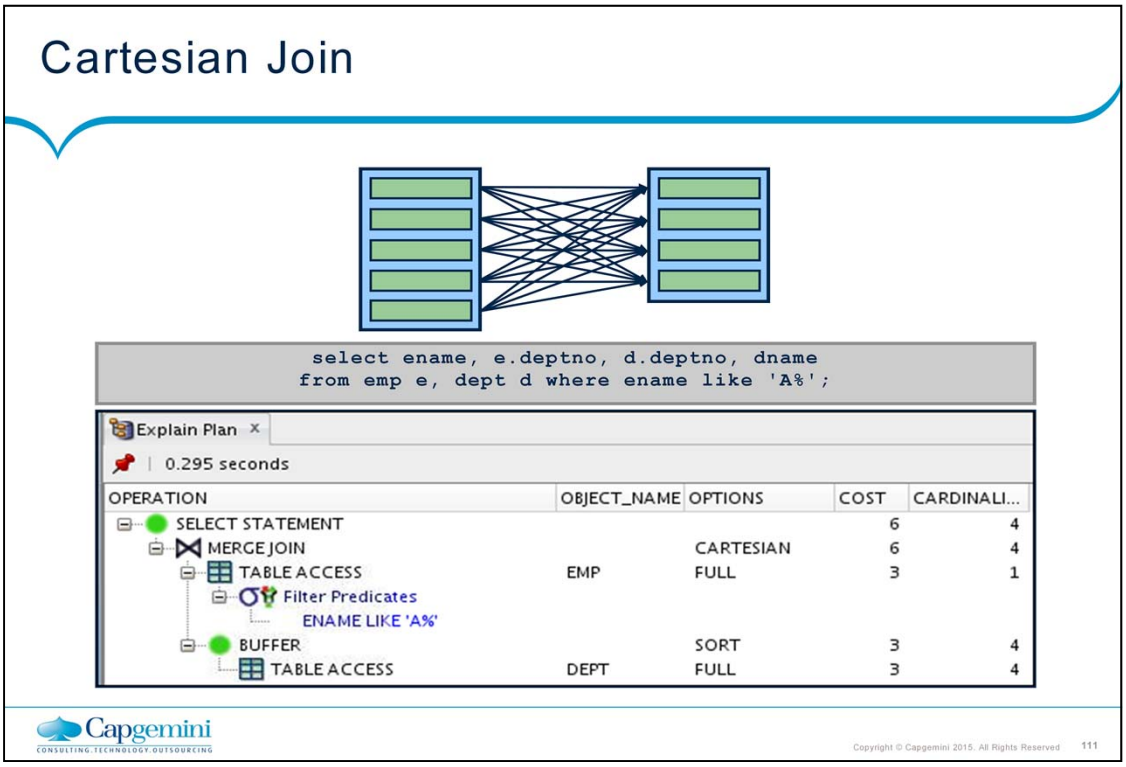
Sort Merge Join

- In a sort merge join, there is no concept of a driving table. A sort merge join is executed as follows:
  - 1. Get the first data set, using any access and filter predicates, and sort it on the join columns.
  - 2. Get the second data set, using any access and filter predicates, and sort it on the join columns.
  - 3. For each row in the first data set, find the start point in the second data set and scan until you find a row that does not join.
- The merge operation combines the two sorted row sources to retrieve every pair of rows that contain matching values for the columns used in the join predicate.
- If one row source has already been sorted in a previous operation (there is an index on the join column, for example), the sort merge operation skips the sort on that row source. When you perform a merge join, you must fetch all rows from the two row sources before to return the first row to the next operation. Sorting could make this join technique expensive, especially if sorting cannot be performed in memory.



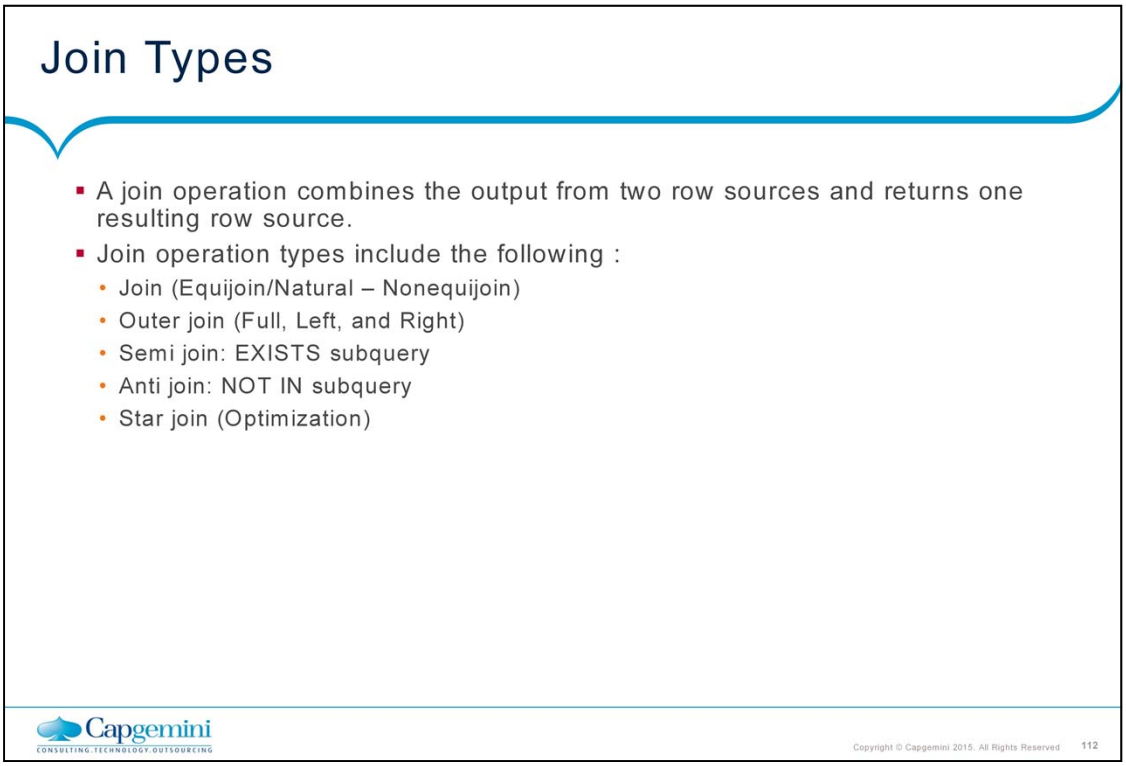
Hash Join

- To perform a hash join between two row sources, the system reads the first data set and builds an array of hash buckets in memory. A hash bucket is little more than a location that acts as the starting point for a linked list of rows from the build table. A row belongs to a hash bucket if the bucket number matches the result that the system gets by applying an internal hashing function to the join column or columns of the row.
- The system starts to read the second set of rows, using whatever access mechanism is most appropriate for acquiring the rows, and uses the same hash function on the join column or columns to calculate the number of the relevant hash bucket. The system then checks to see if there are any rows in that bucket. This is known as probing the hash table.
- If there are no rows in the relevant bucket, the system can immediately discard the row from the probe table.
- If there are some rows in the relevant bucket, the system does an exact check on the join column or columns to see if there is a proper match. Any rows that survive the exact check can immediately be reported (or passed on to the next step in the execution plan). So, when you perform a hash join, you must fetch all rows from the smallest row source to return the first row to next operation.
- Note: Hash joins are performed only for equijoins, and are most useful when joining large amount of data.



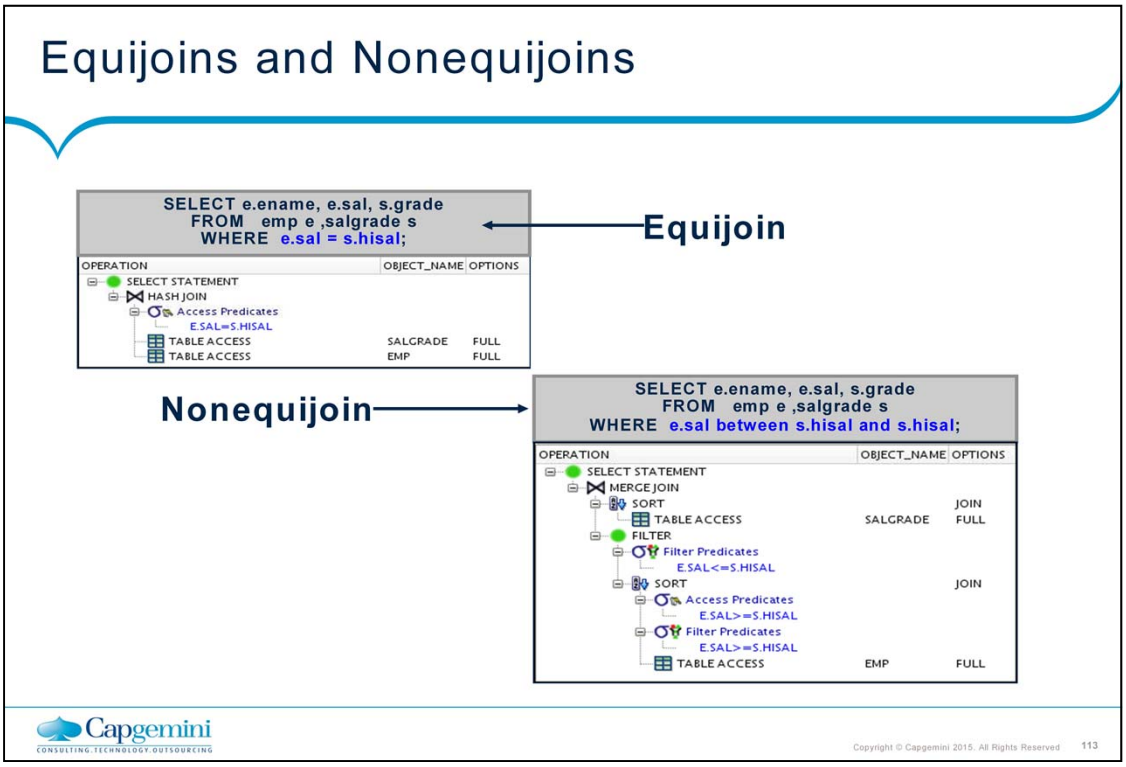
Cartesian Join

- A Cartesian join is used when one or more of the tables does not have any join conditions to any other tables in the statement. The optimizer joins every row from one data source with every row from the other data source, creating the Cartesian product of the two sets.
- A Cartesian join can be seen as a nested loop with no elimination; the first row source is read and then for every row, all the rows are returned from the other row source.
- Note: Cartesian join is generally not desirable. However, it is perfectly acceptable to have one with single-row row source (guaranteed by a unique index, for example) joined to some other table.



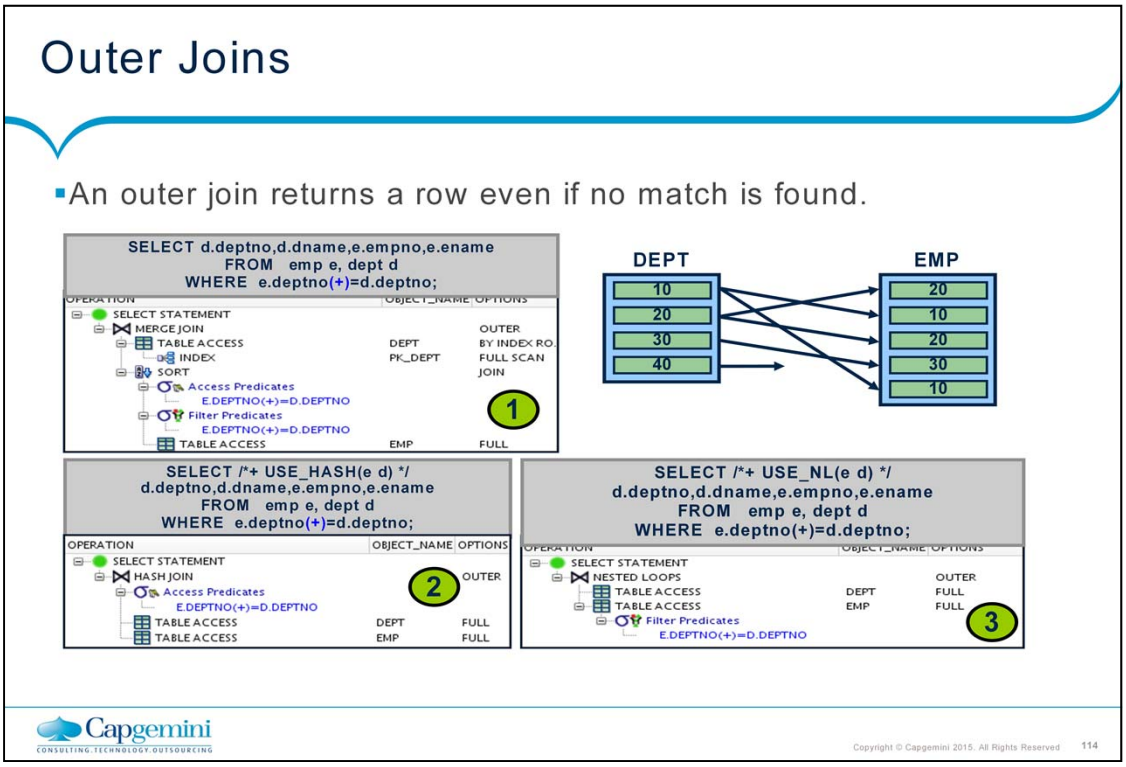
Join Types

- Join operation types include the following:
  - Join (equijoin and nonequijoin): Returns rows that match predicate join
  - Outer join: Returns rows that match predicate join and row when no match is found
  - Semi join: Returns rows that match the EXISTS subquery. Find one match in the inner table, then stop search.
  - Anti join: Returns rows with no match in the NOT IN subquery. Stop as soon as one match is found.
  - Star join: This is not a join type, but just a name for an implementation of a performance optimization to better handle the fact and dimension model.
- Antijoin and semijoin are considered to be join types, even though the SQL constructs that cause them are subqueries. Antijoin and semijoin are internal optimizations algorithms used to flatten subquery constructs in such a way that they can be resolved in a join-like way.



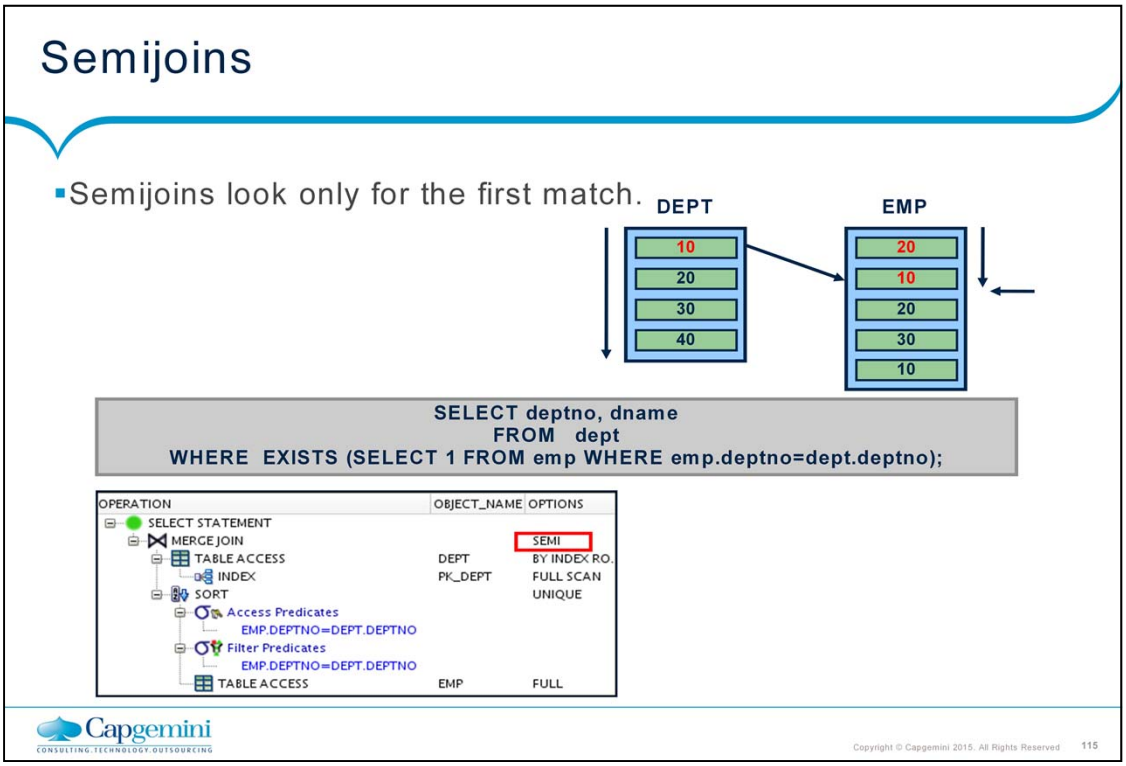
Equijoins and Nonequijoins

- The join condition determines whether a join is an equijoin or a nonequijoin. An equijoin is a join with a join condition containing an equality operator. When a join condition relates two tables by an operator other than equality, it is a nonequijoin.
- Equijoins are the most commonly used. An example each of an equijoin and a nonequijoin are shown in the slide. Nonequijoins are less frequently used.
- To improve SQL efficiency, use equijoins whenever possible. Statements that perform equijoins on untransformed column values are the easiest to tune.
- Note: If you have a nonequijoin, a hash join is not possible.



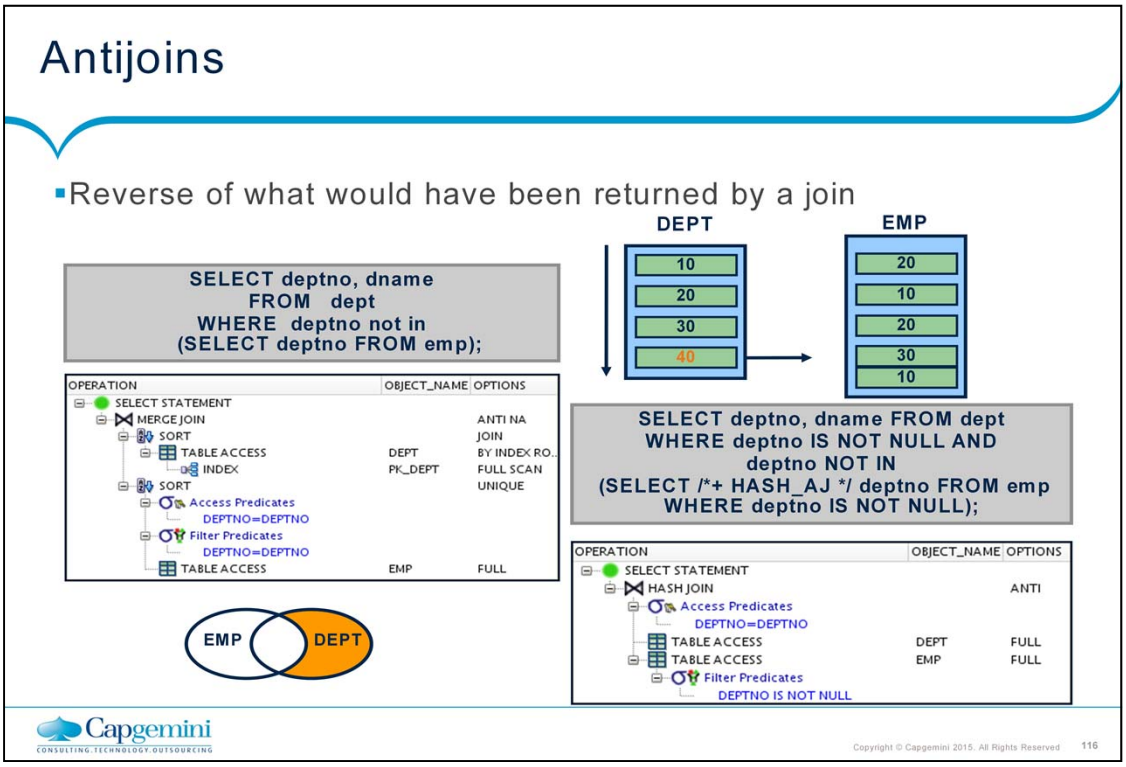
Outer Joins

- The simple join is the most commonly used within the system. Other joins open up extra functionality, but have much more specialized uses. The outer join operator is placed on the deficient side of the query. In other words, it is placed against the table that has the missing join information. Consider EMP and DEPT. There may be a department that has no employees. If EMP and DEPT are joined together, this particular department would not appear in the output because there is no row that matches the join condition for that department. By using the outer join, the missing department can be displayed.
  - 1. Merge Outer joins: By default, the optimizer uses MERGE OUTER JOIN.
  - 2. Outer join with nested loops: The left/driving table is always the table whose rows are being preserved (DEPT in the example). For each row from DEPT, look for all matching rows in EMP. If none is found, output DEPT values with null values for the EMP columns. If rows are found, output DEPT values with these EMP values.
  - 3. Hash Outer joins: The left/outer table whose rows are being preserved is used to build the hash table, and the right/inner table is used to probe the hash table. When a match is found, the row is output and the entry in the hash table is marked as matched to a row. After the inner table is exhausted, the hash table is read over once again, and any rows that are not marked as matched are output with null values for the EMP columns. The system hashes the table whose rows are not being preserved, and then reads the table whose rows are being preserved, probing the hash table to see whether there was a row to join to.
- Note: You can also use the ANSI syntax for full, left, and right outer joins (not shown in the slide).



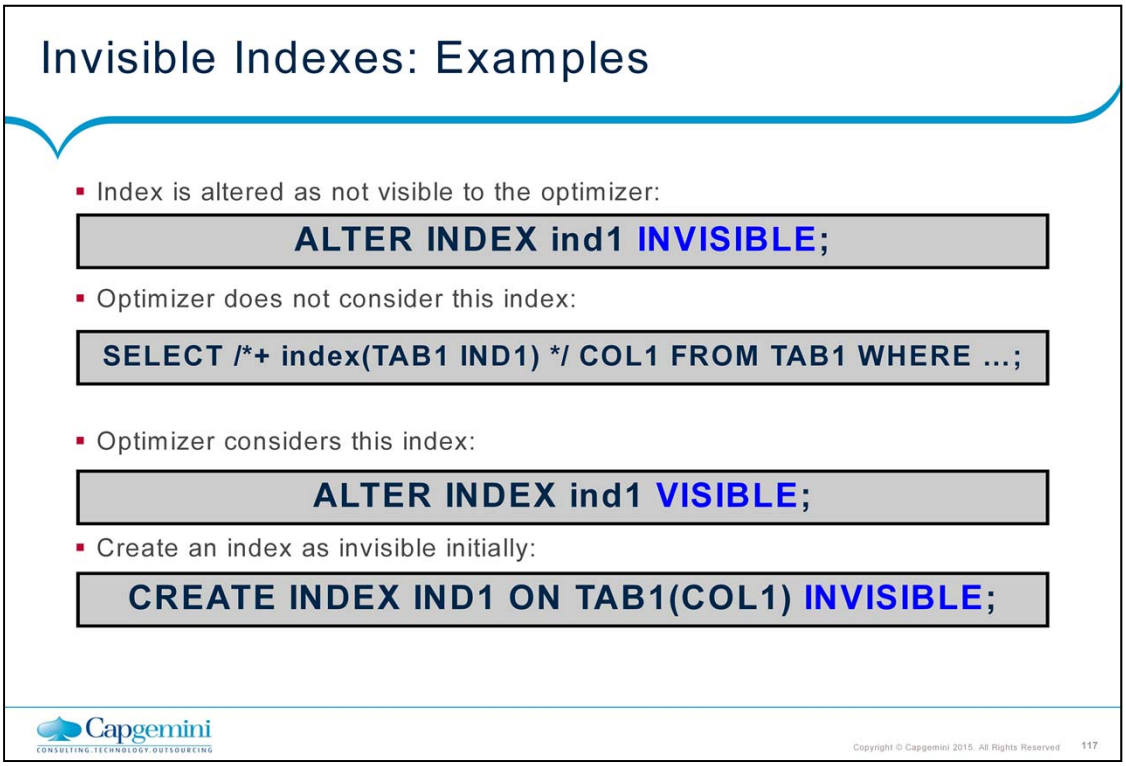
Semijoins

- Semijoins return a result when you hit the first joining record. A semijoin is an internal way of transforming an EXISTS subquery into a join. However, you cannot see this occur anywhere.
- Semijoins return rows that match an EXISTS subquery without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery.
- In the above diagram, for each DEPT record, only the first matching EMP record is returned as a join result. This prevents scanning huge numbers of duplicate rows in a table when all you are interested in is if there are any matches.
- When the subquery is not unnested, a similar result could be achieved by using a FILTER operation and scanning a row source until a match is found, then returning it.
- Note: A semijoin can always use a Merge join. The optimizer may choose nested-loop, or hash joins methods to perform semijoins as well.



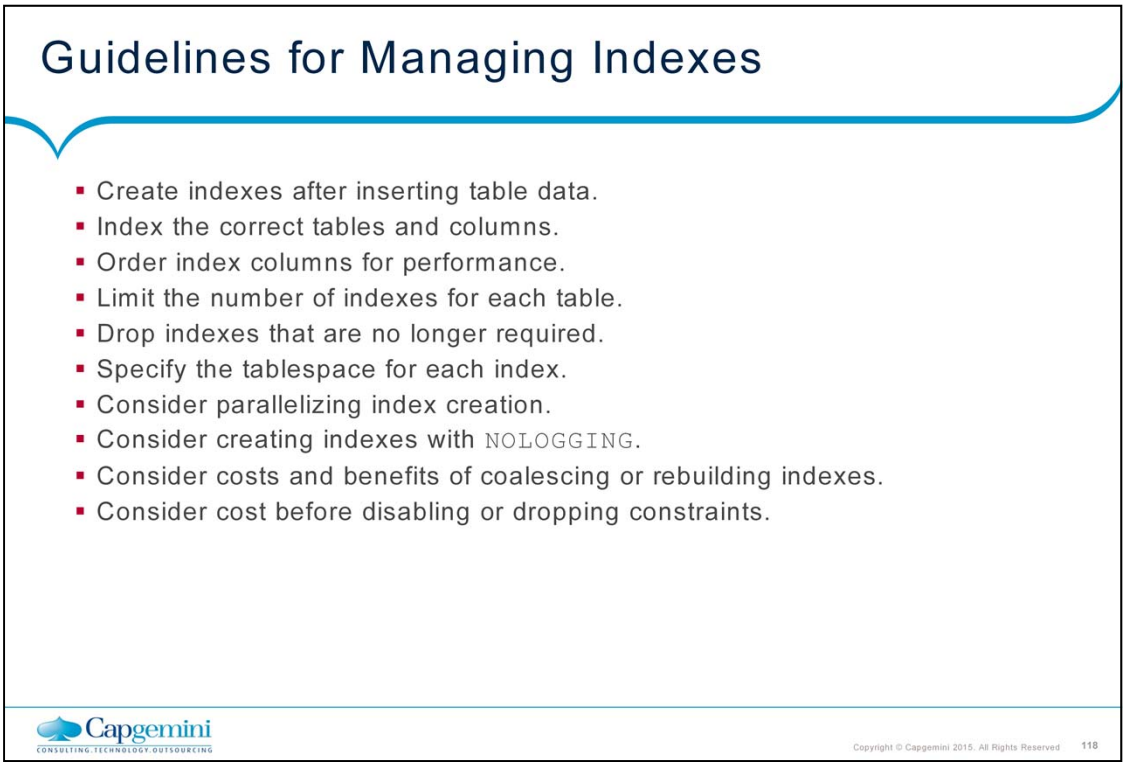
Antijoins

- Antijoins return rows that fail to match (NOT IN) the subquery at the right side. For example, an antijoin can select a list of departments which do not have any employee.
- The optimizer uses a merge antijoin algorithm for NOT IN subqueries by default. However, if the HASH_AJ or NL_AJ hints are used and various required conditions are met, the NOT IN uncorrelated subquery can be changed. Although antijoins are mostly transparent to the user, it is useful to know that these join types exist and could help explain unexpected performance changes between releases.



Invisible Indexes: Examples

- When an index is invisible, the optimizer selects plans that do not use the index. If there is no discernible drop in performance, you can drop the index. You can also create an index initially as invisible, perform testing, and then determine whether to make the index visible.
- You can query the VISIBILITY column of the *_INDEXES data dictionary views to determine whether the index is VISIBLE or INVISIBLE.
- Note: For all the statements given in the slide, it is assumed that OPTIMIZER_USE_INVISIBLE_INDEXES is set to FALSE.



Guidelines for Managing Indexes

- Create indexes after inserting table data: Data is often inserted or loaded into a table using either the SQL*Loader or an import utility. It is more efficient to create an index for a table after inserting or loading the data.
- Index the correct tables and columns: Use the following guidelines for determining when to create an index:

  Create an index if you frequently want to retrieve less than 15% of the rows in a large table.

  To improve performance on joins of multiple tables, index the columns used for joins.

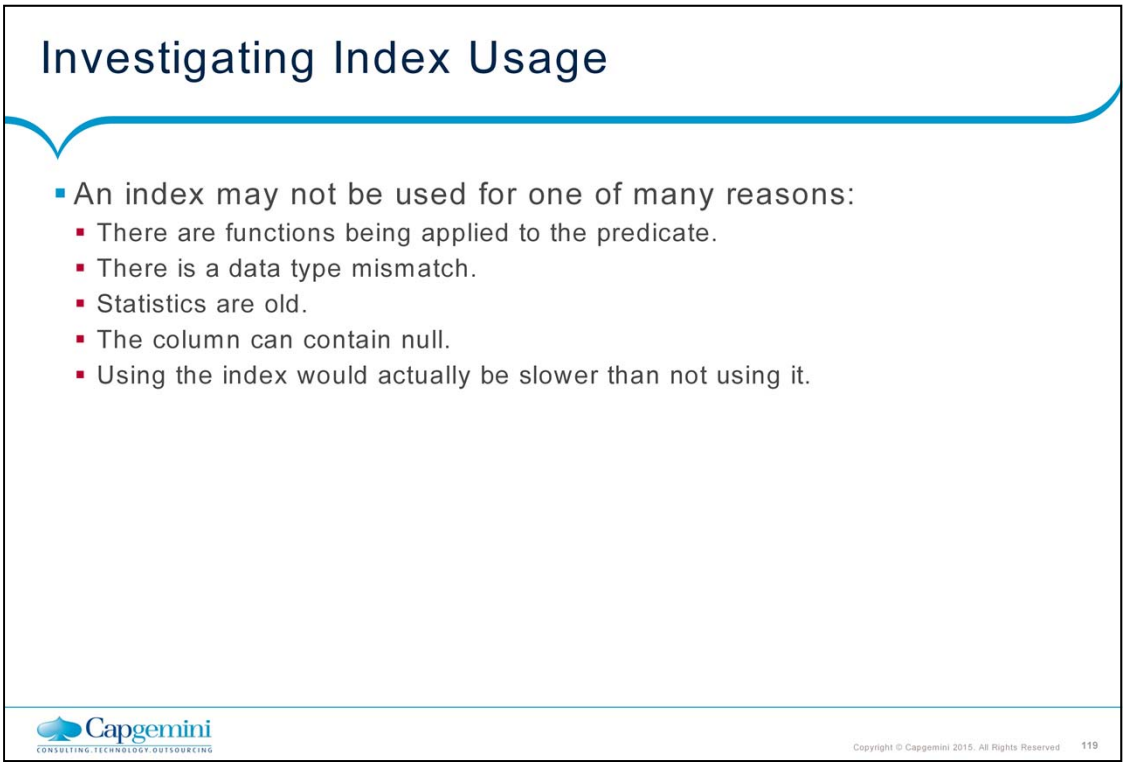
  Small tables do not require indexes.
- Columns suitable for indexing: Some columns are strong candidates for indexing:

  Values are relatively unique in the column.

  There is a wide range of values (good for regular indexes).

  There is a small range of values (good for bitmap indexes).

  The column contains many nulls, but queries often select all rows having a value.



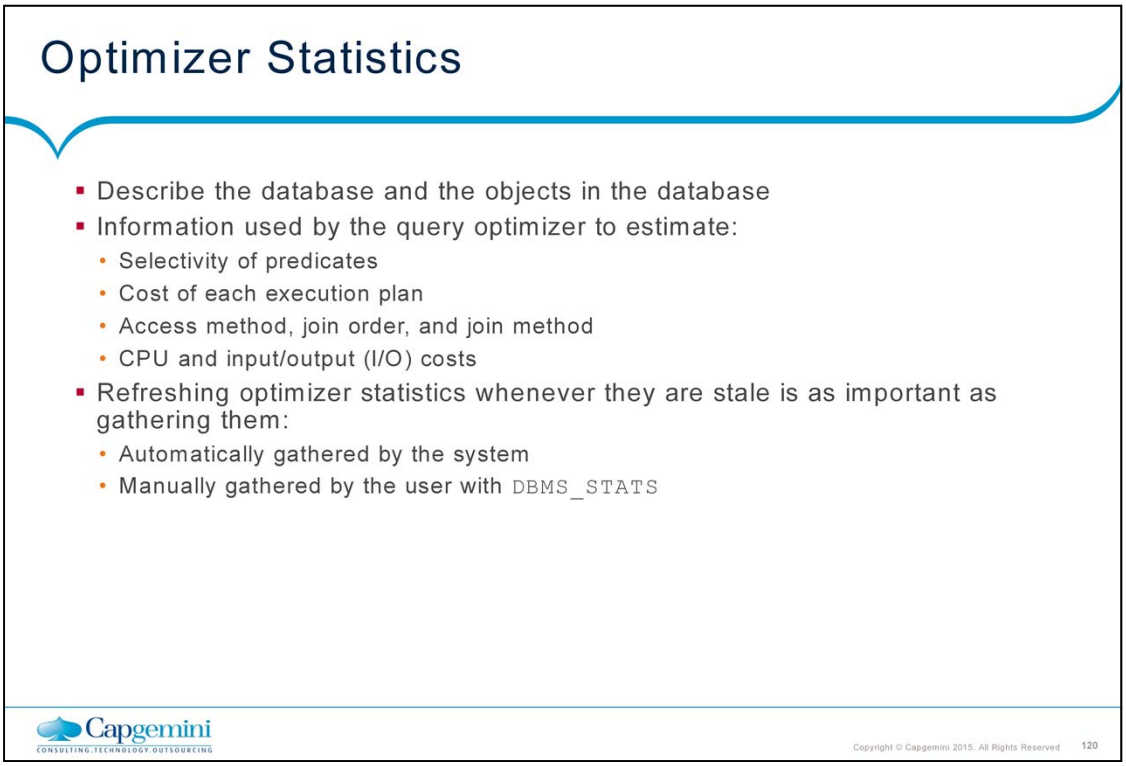
Investigating Index Usage

- You may often run a SQL statement expecting a particular index to be used, and it is not. This can be because the optimizer is unaware of some information, or because it should not use the index.
- Functions
- If you apply a function to the indexed column in the WHERE clause, the index cannot be used; the index is based on column values without the effect of the function. For example, the following statement does not use an index on the salary column:

```
SELECT * FROM employees WHERE 1.10*salary >
    10000
```

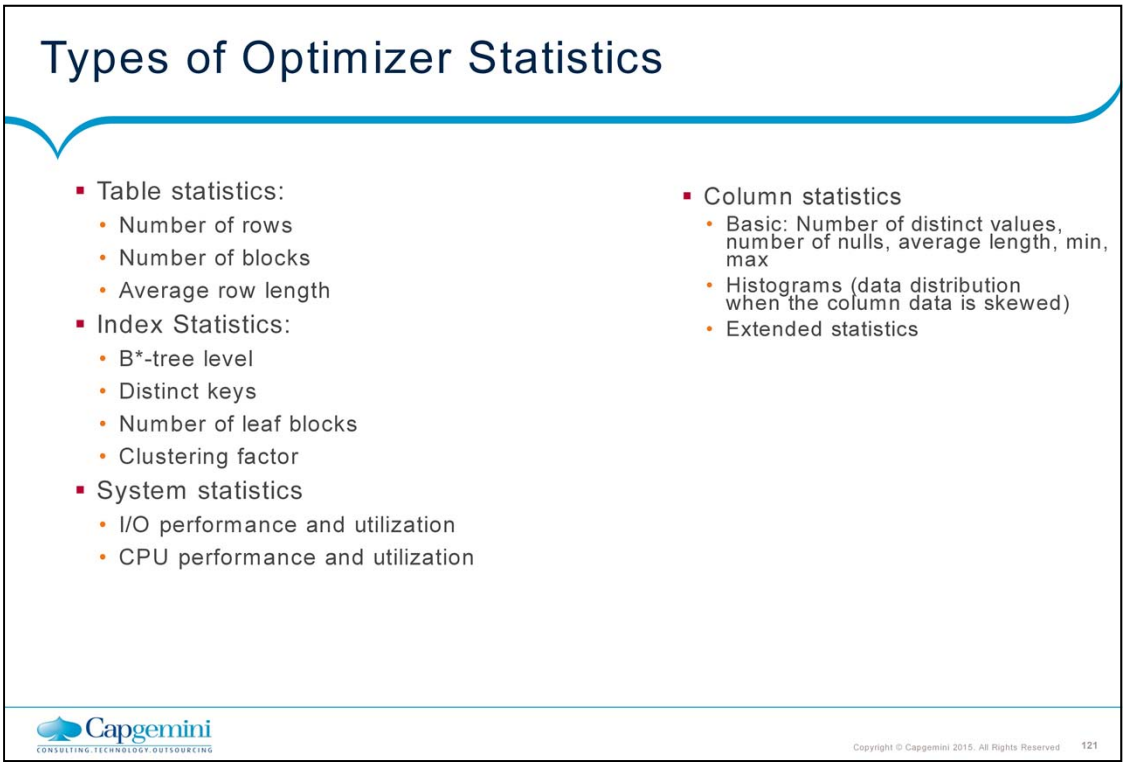
- If you want an index to be used in this case, you can create a function-based index. Function-based indexes were covered under “Index Range Scan: Function-Based” earlier in this lesson.
- Data Type Mismatch
- If there is a data type mismatch between the indexed column and the compared value, the index is not used. This is due to the implicit data type conversion. For example, if the SSN column is of the VARCHAR2 type, the following does not use the index on SSN:

```
SELECT * FROM person WHERE SSN =
    123456789
```



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 10g, 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.

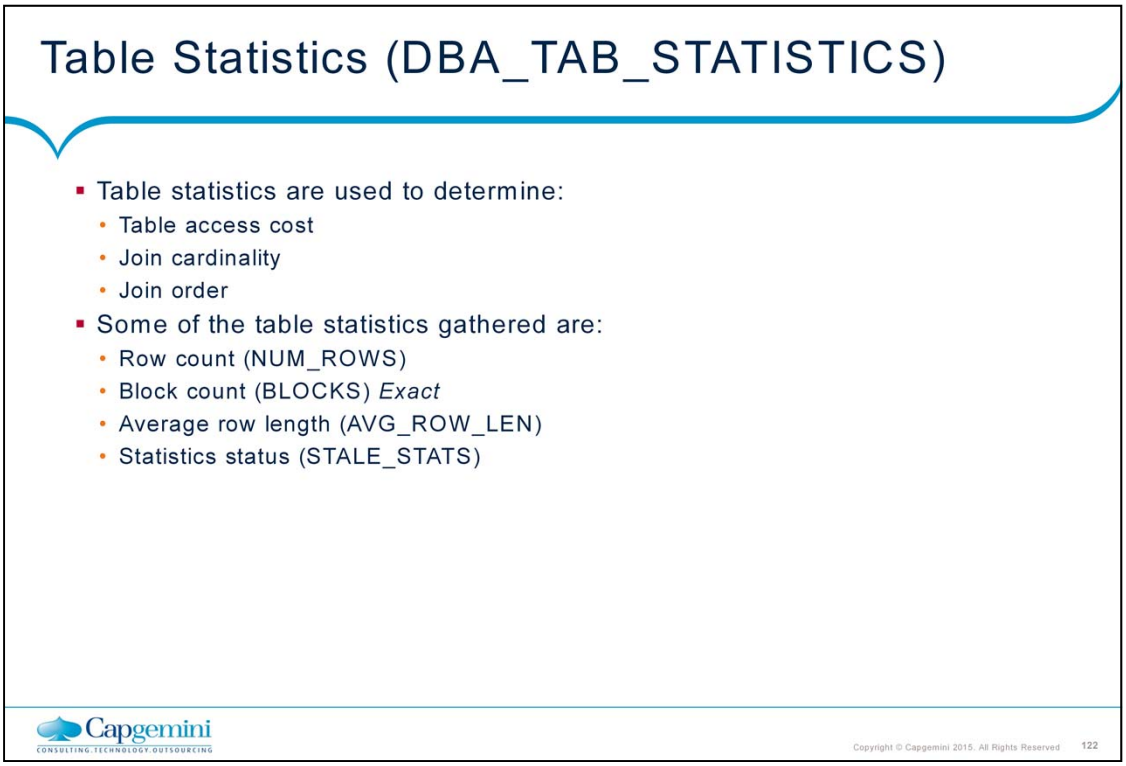
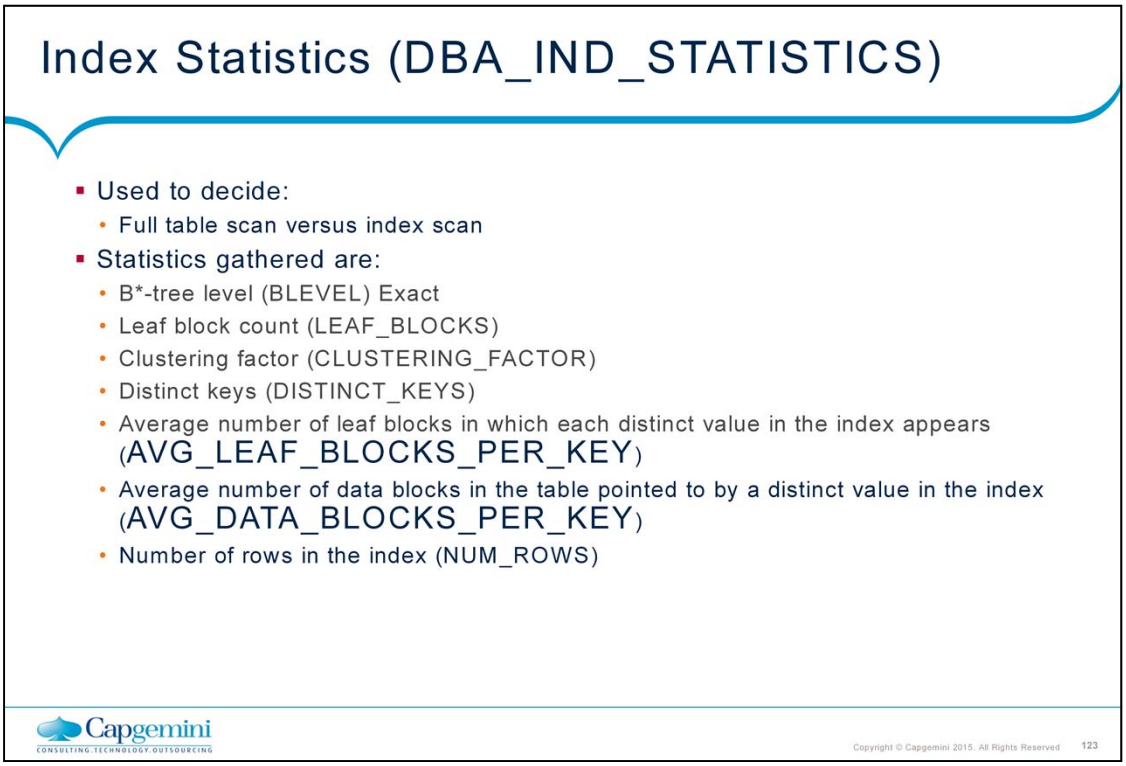


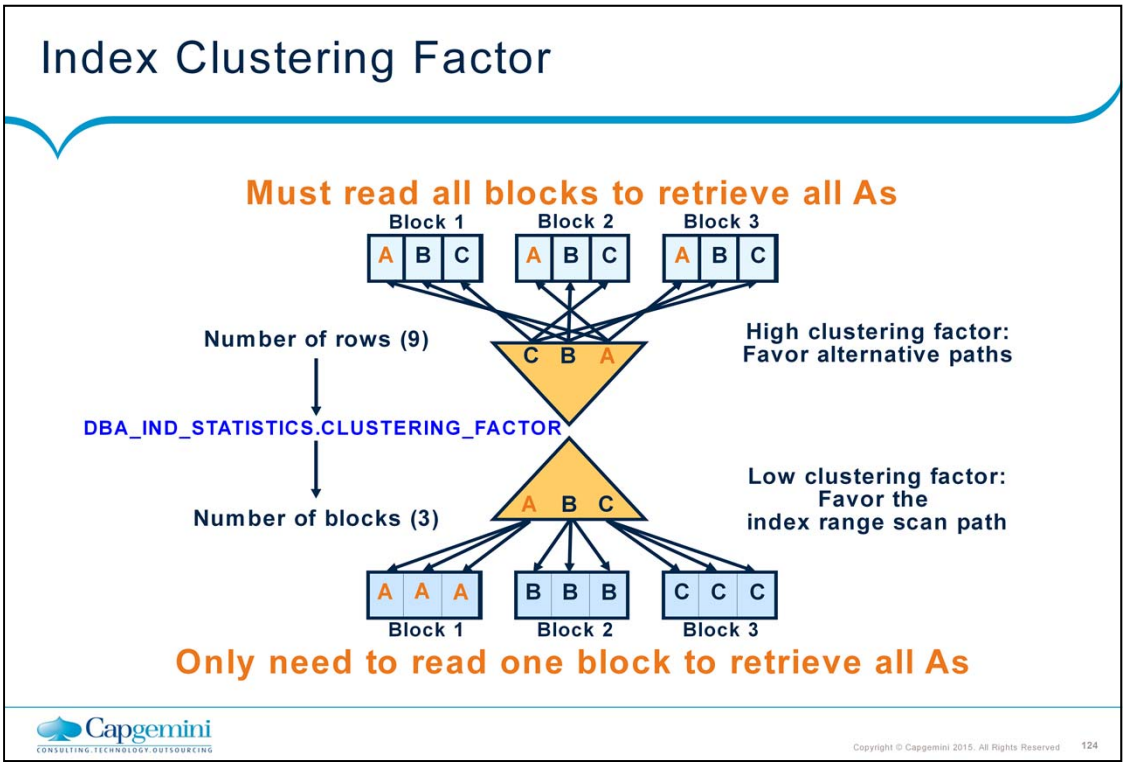
Table Statistics (DBA_TAB_STATISTICS)

- 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.



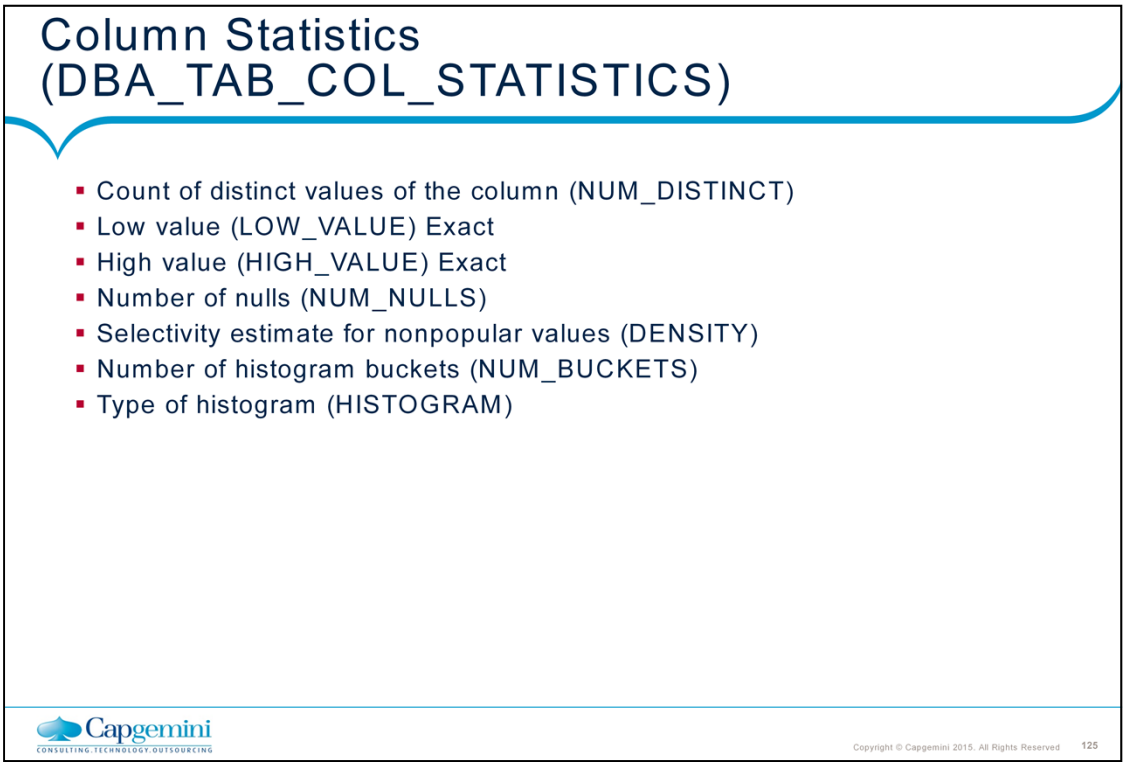
Index Statistics (DBA_IND_STATISTICS)

- 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.



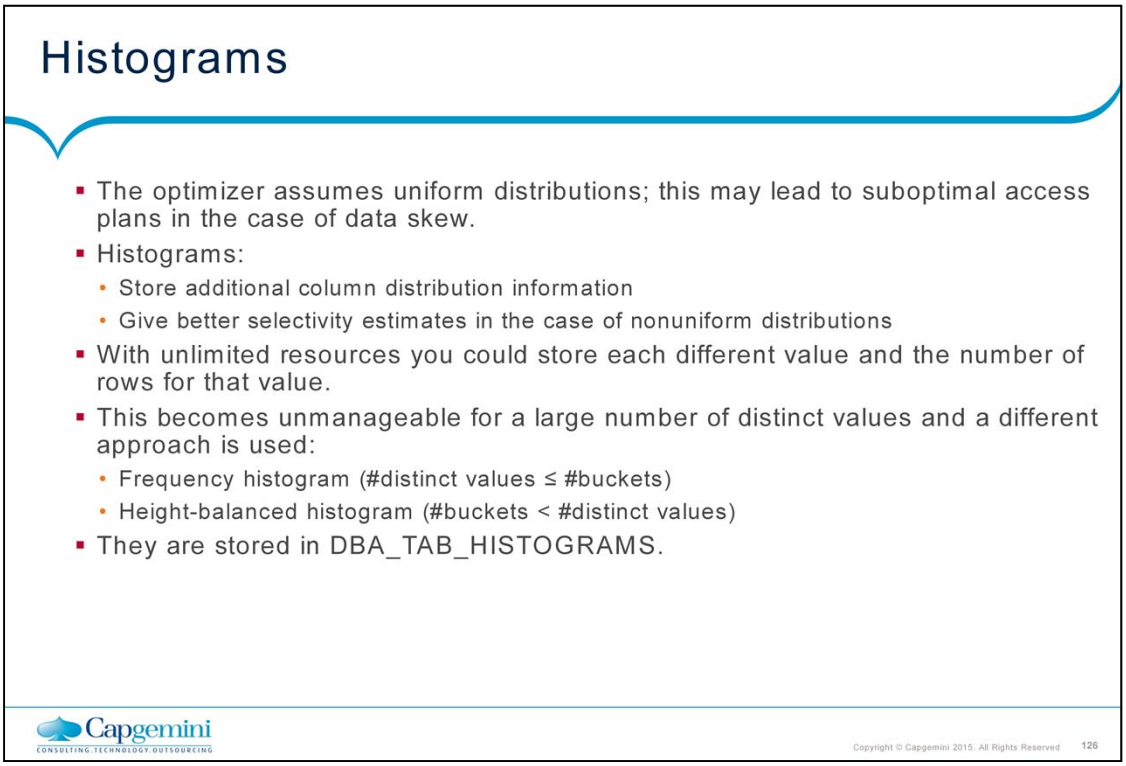
Index Clustering Factor

- 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.



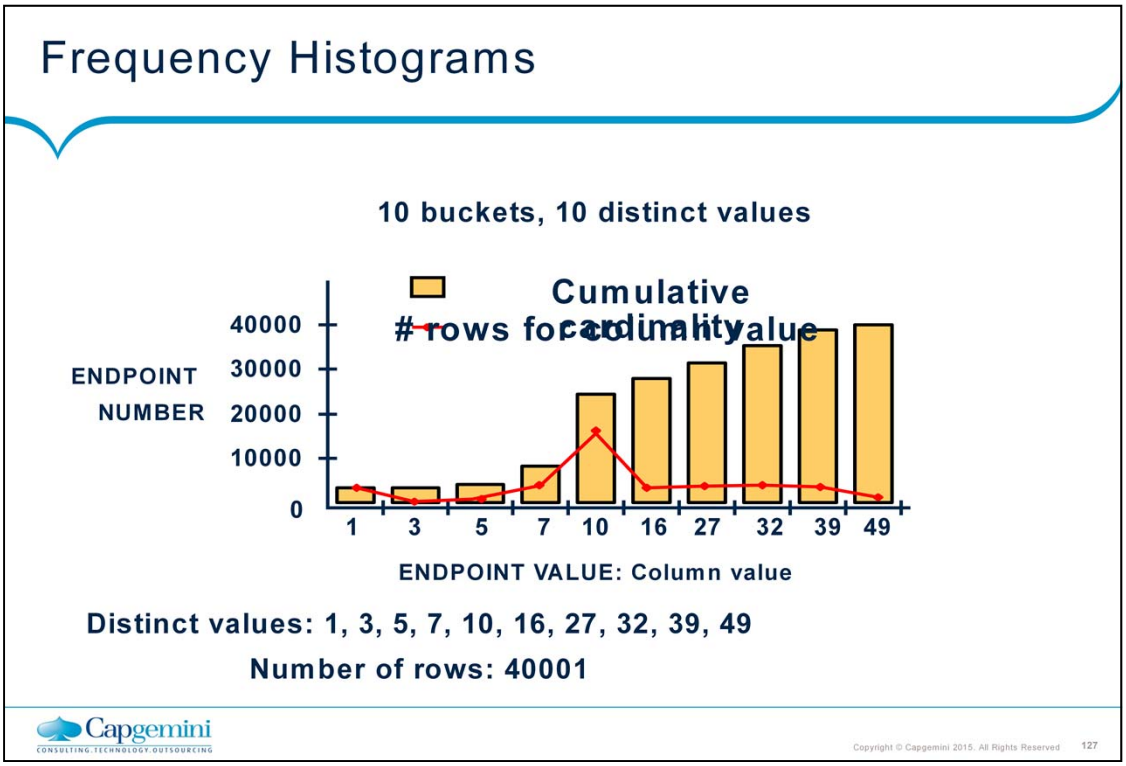
Column Statistics (DBA_TAB_COL_STATISTICS)

- 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



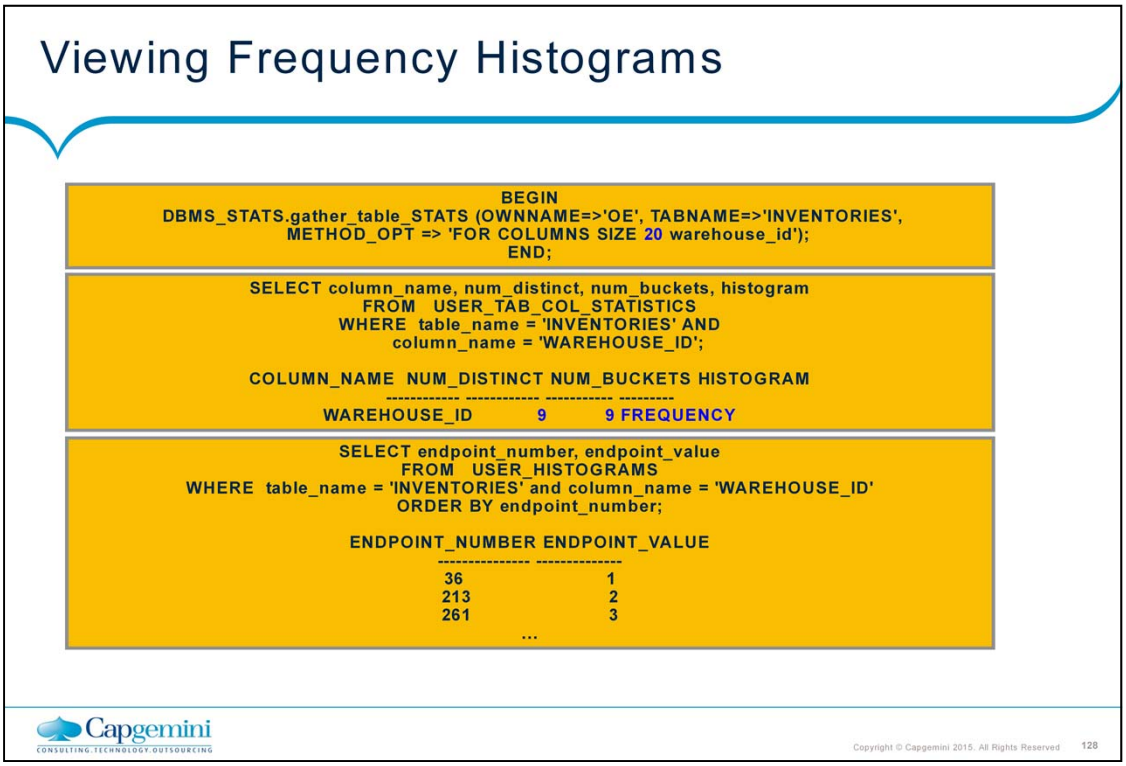
Histograms

- 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.



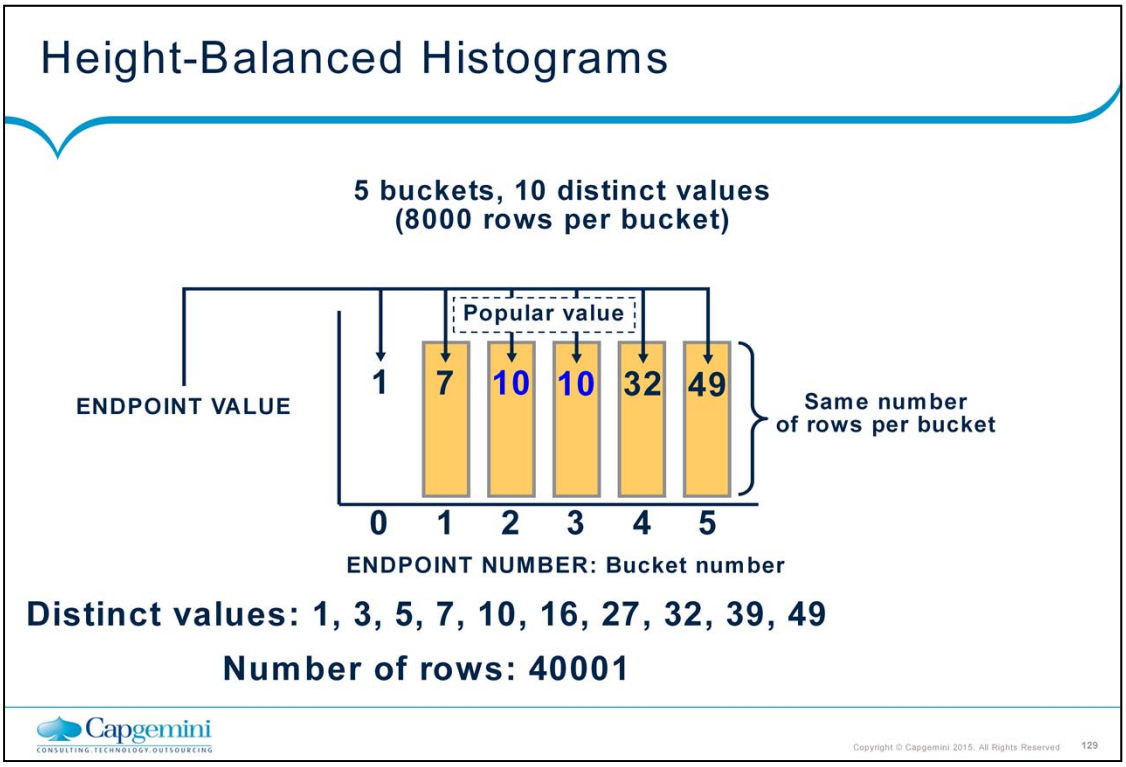
Frequency Histograms

- 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.



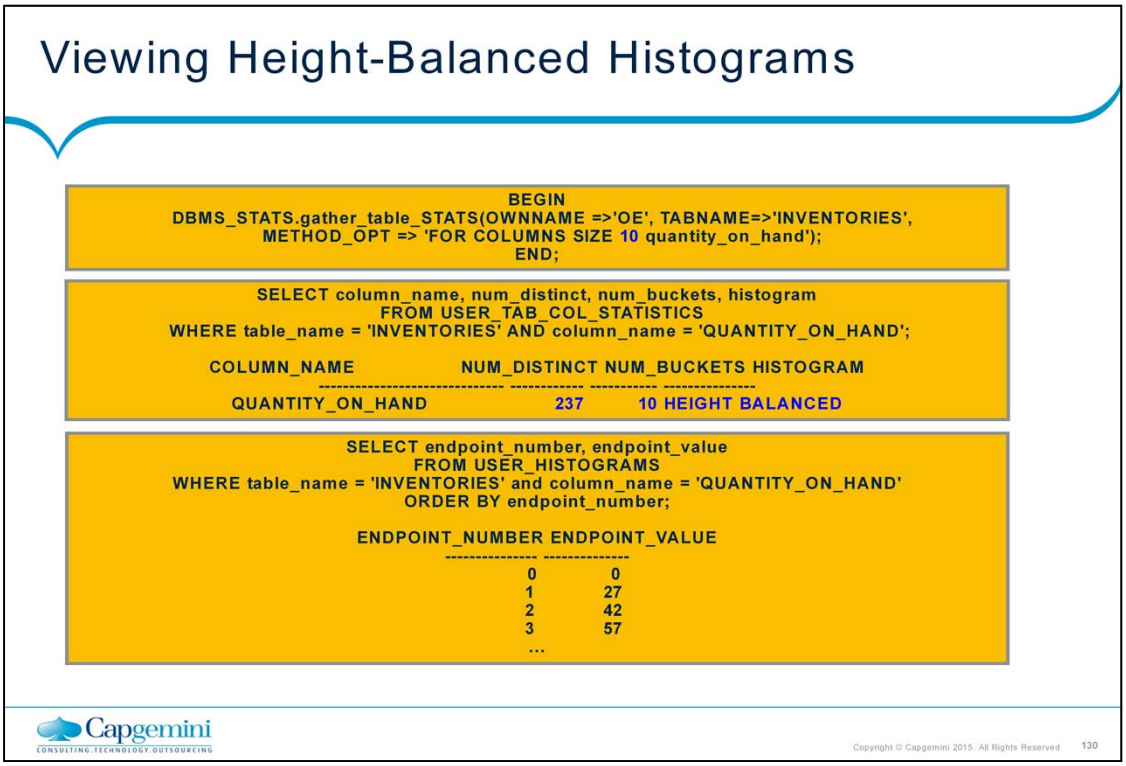
Viewing Frequency Histograms

- 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.



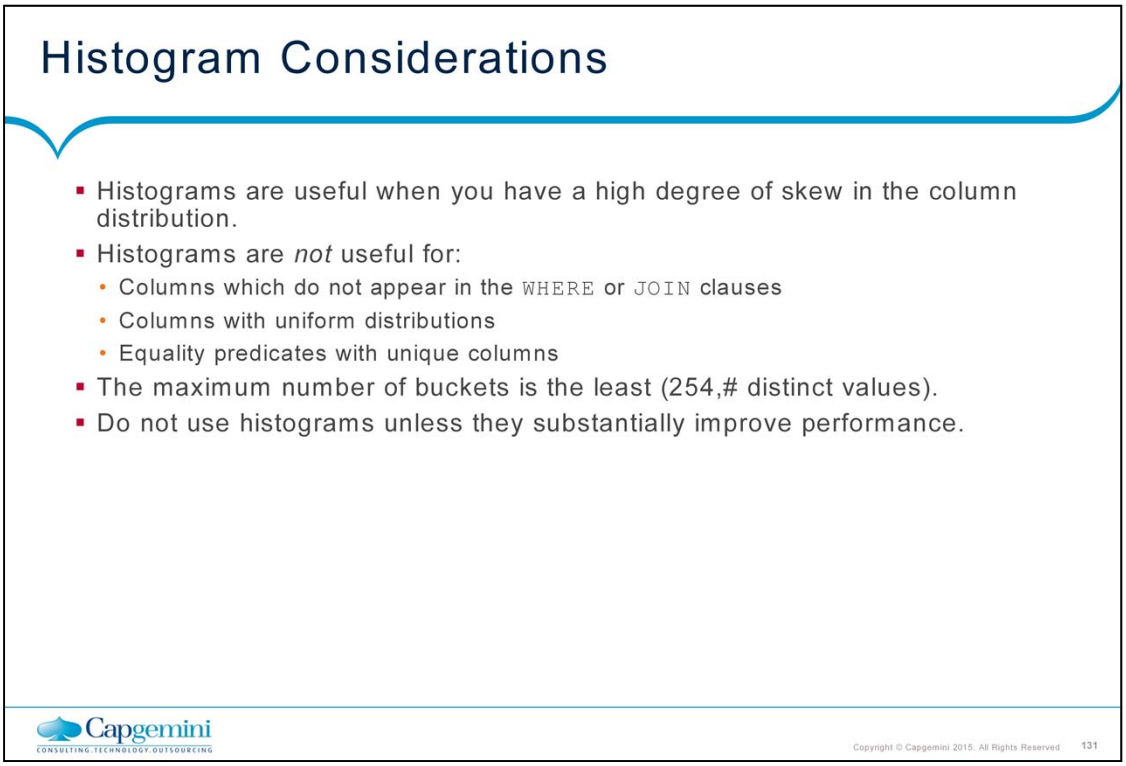
Height-Balanced Histograms

- 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.



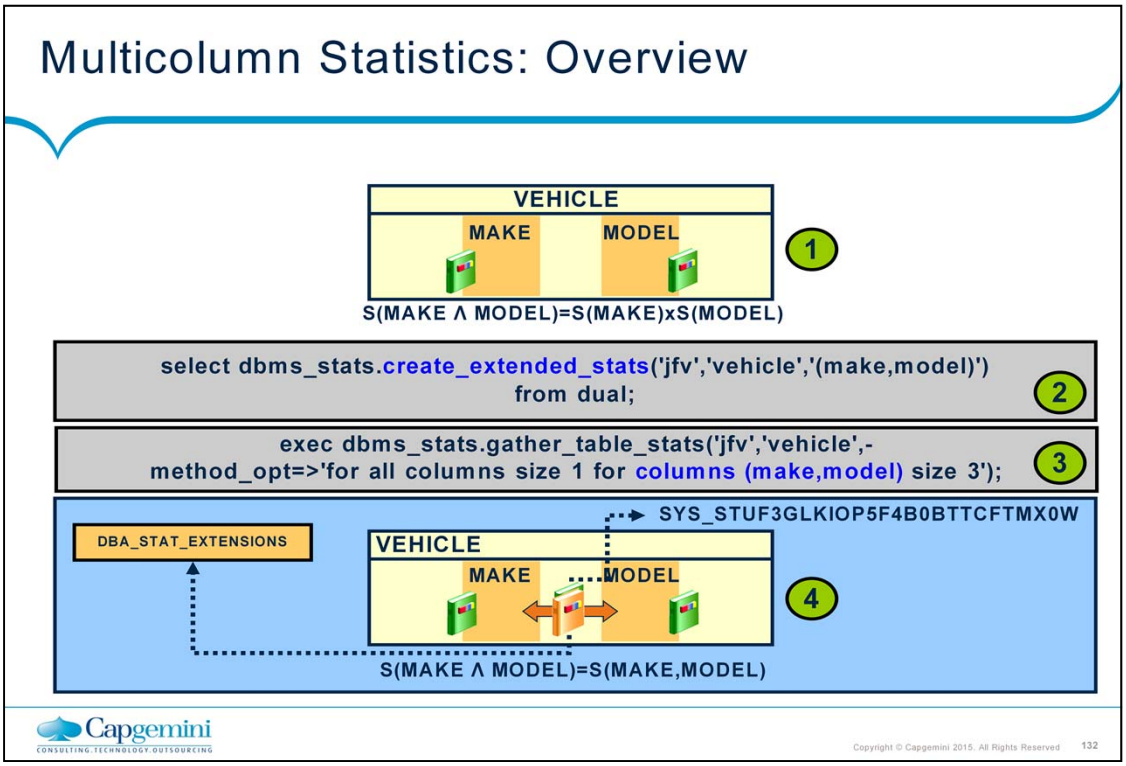
Viewing Height-Balanced Histograms

- 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.



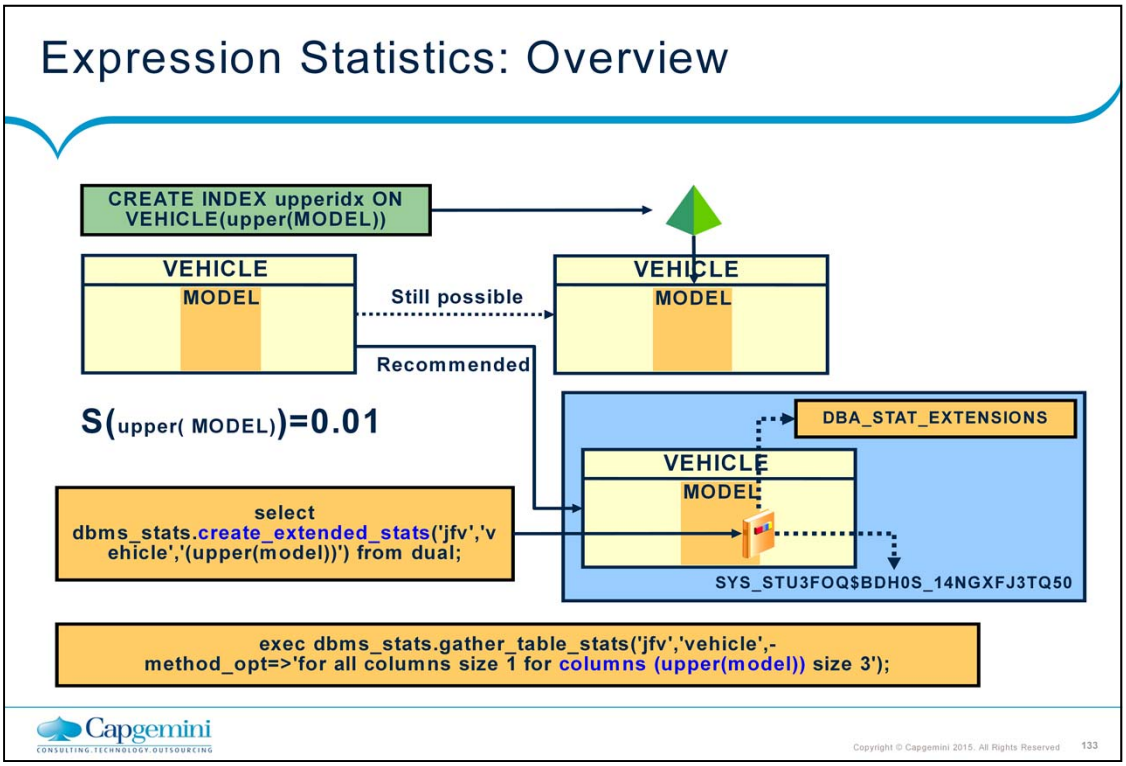
Histogram Considerations

- 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.



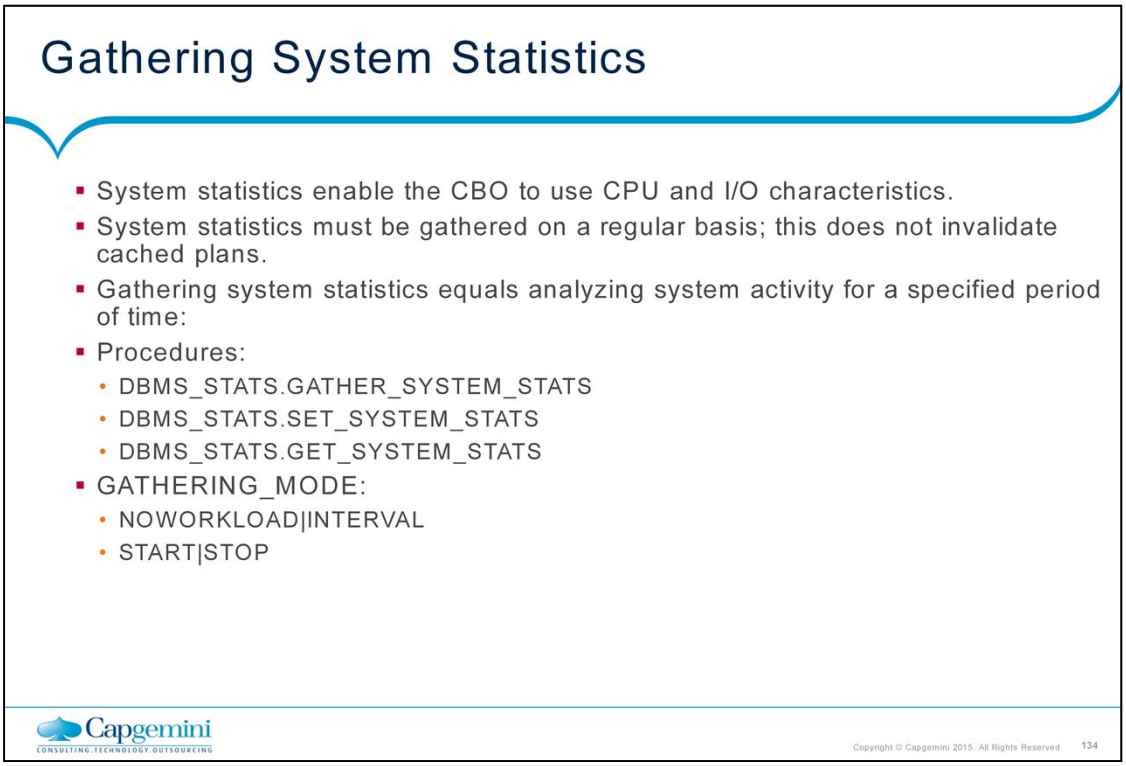
Multicolumn Statistics: Overview

- With Oracle Database 10g, 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 11g 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.



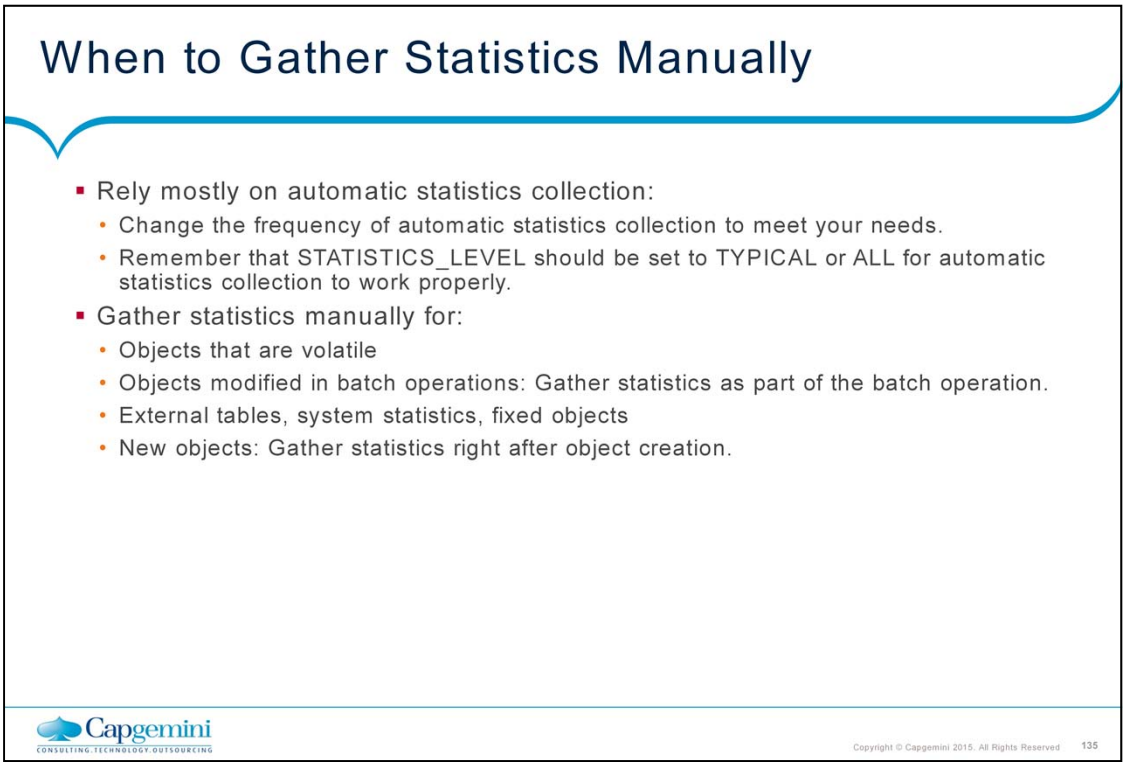
Expression Statistics: Overview

- 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 11g, 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.



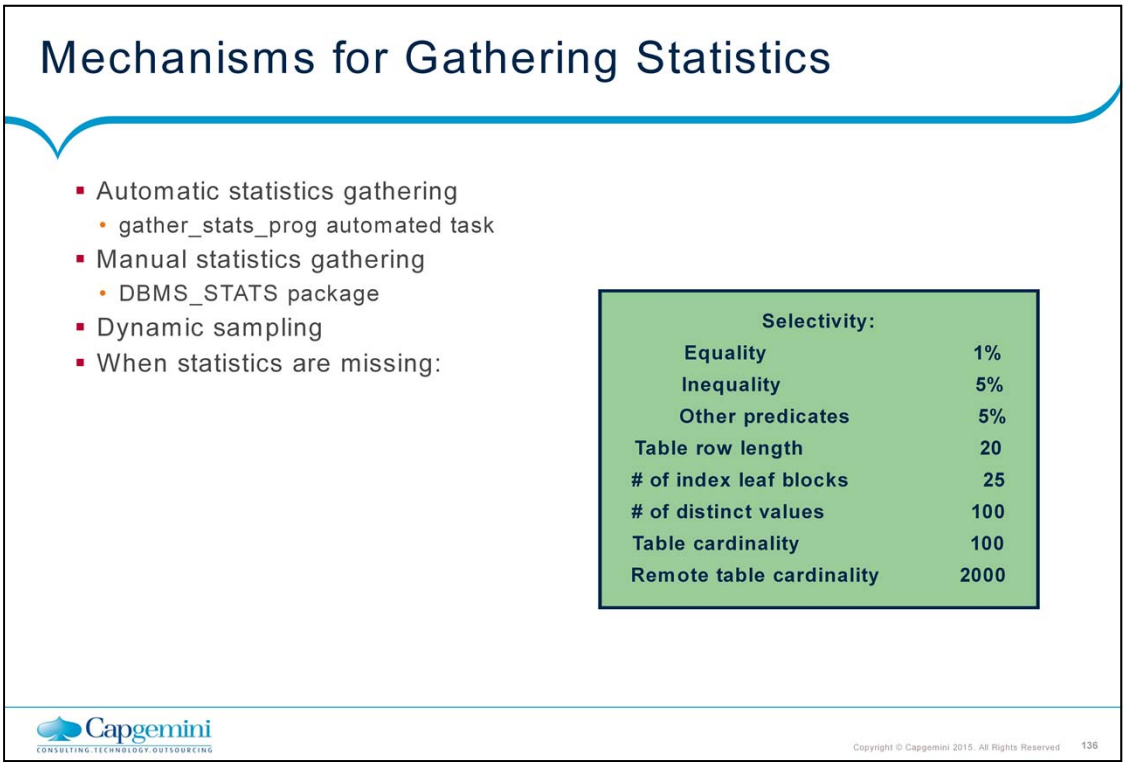
Gathering System Statistics

- 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.



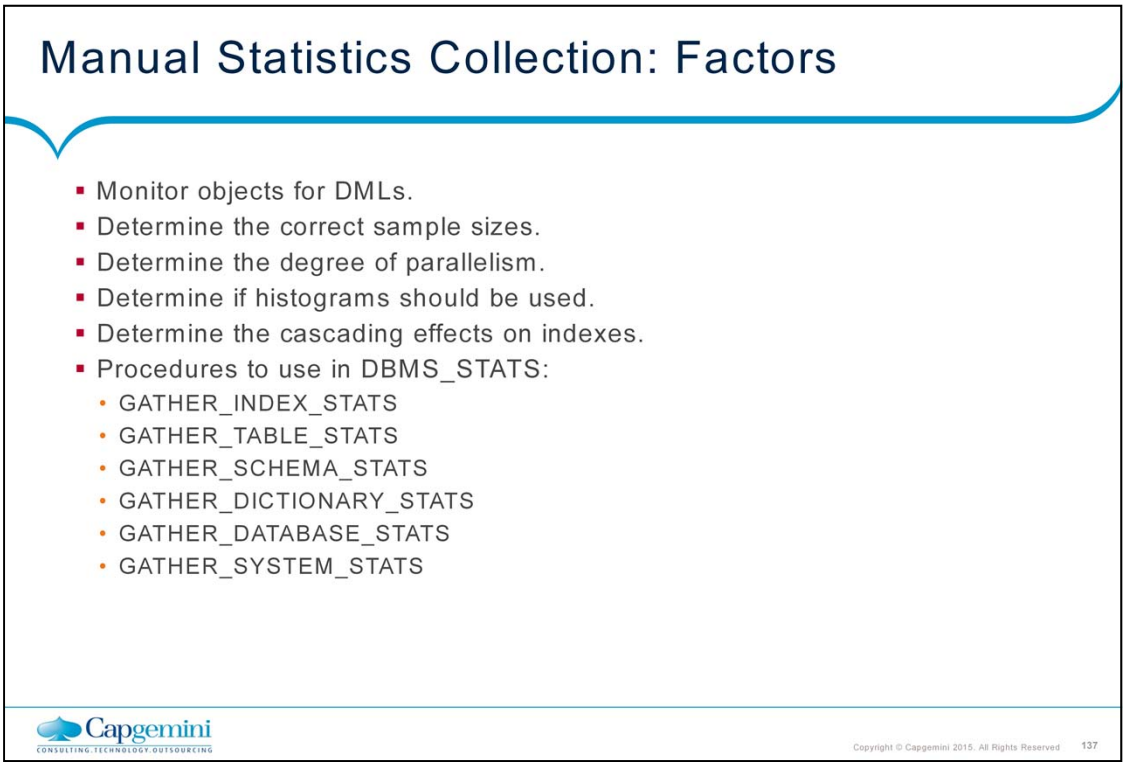
When to Gather Statistics Manually

- 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.



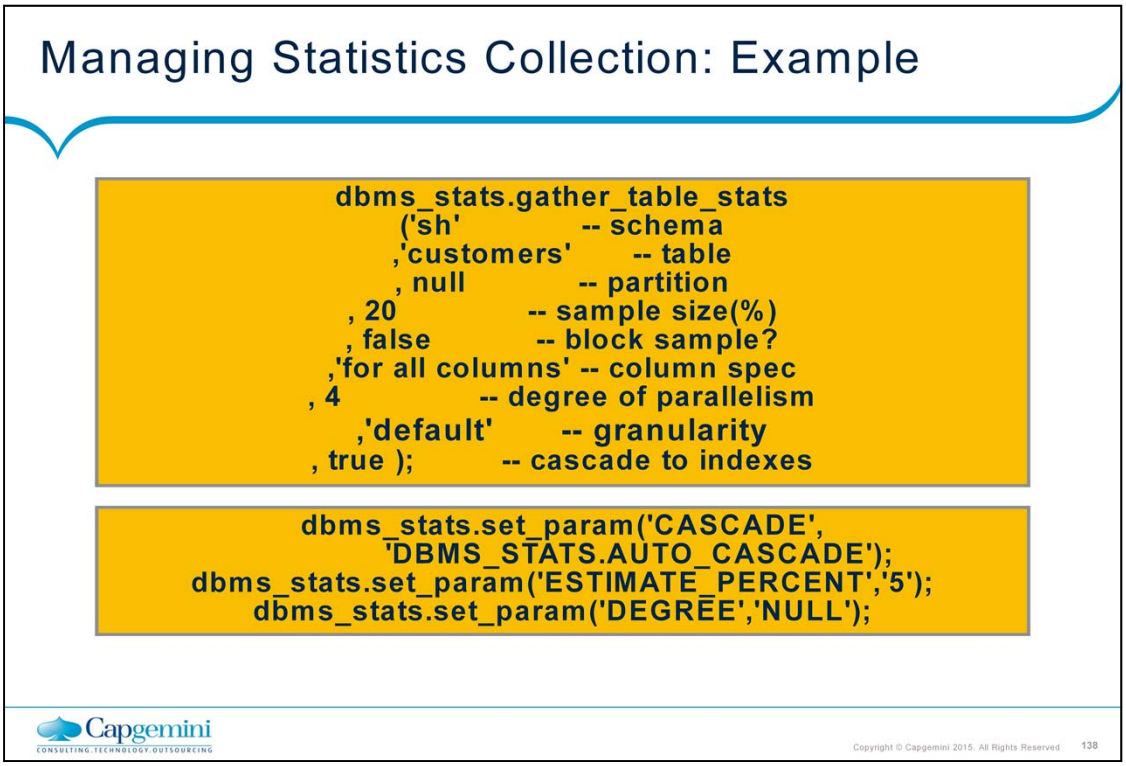
Mechanisms for Gathering Statistics

- 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.



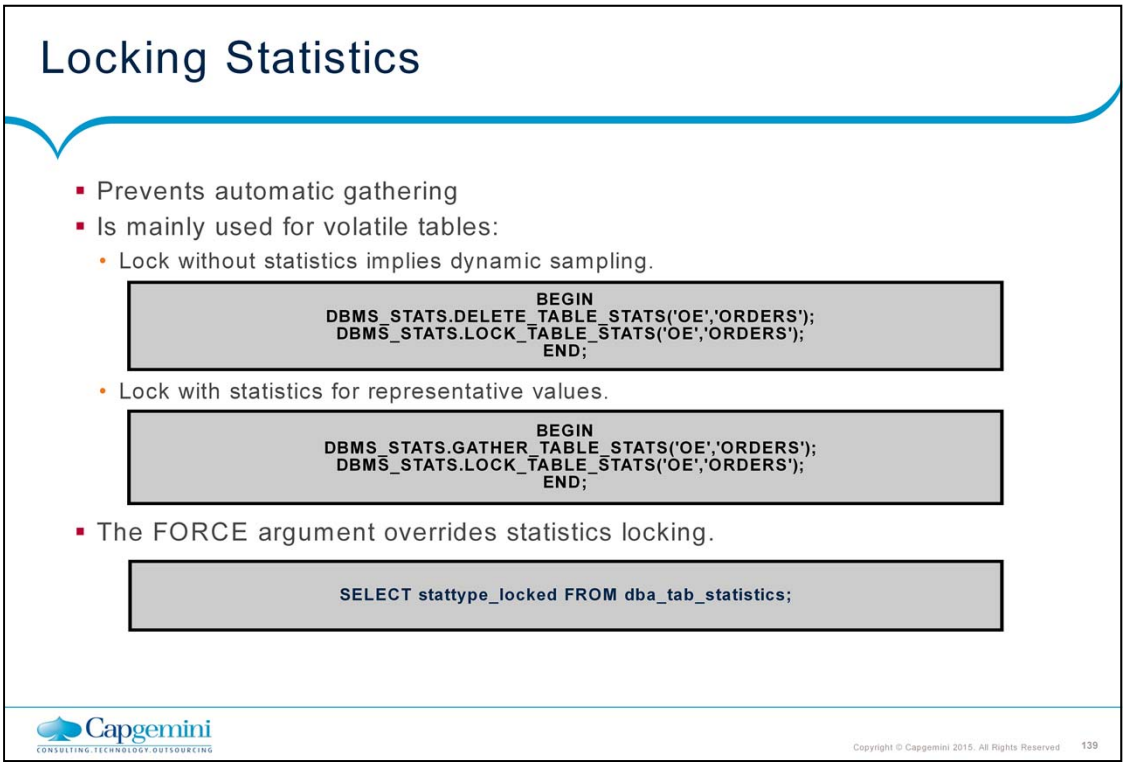
Manual Statistics Collection: Factors

- 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



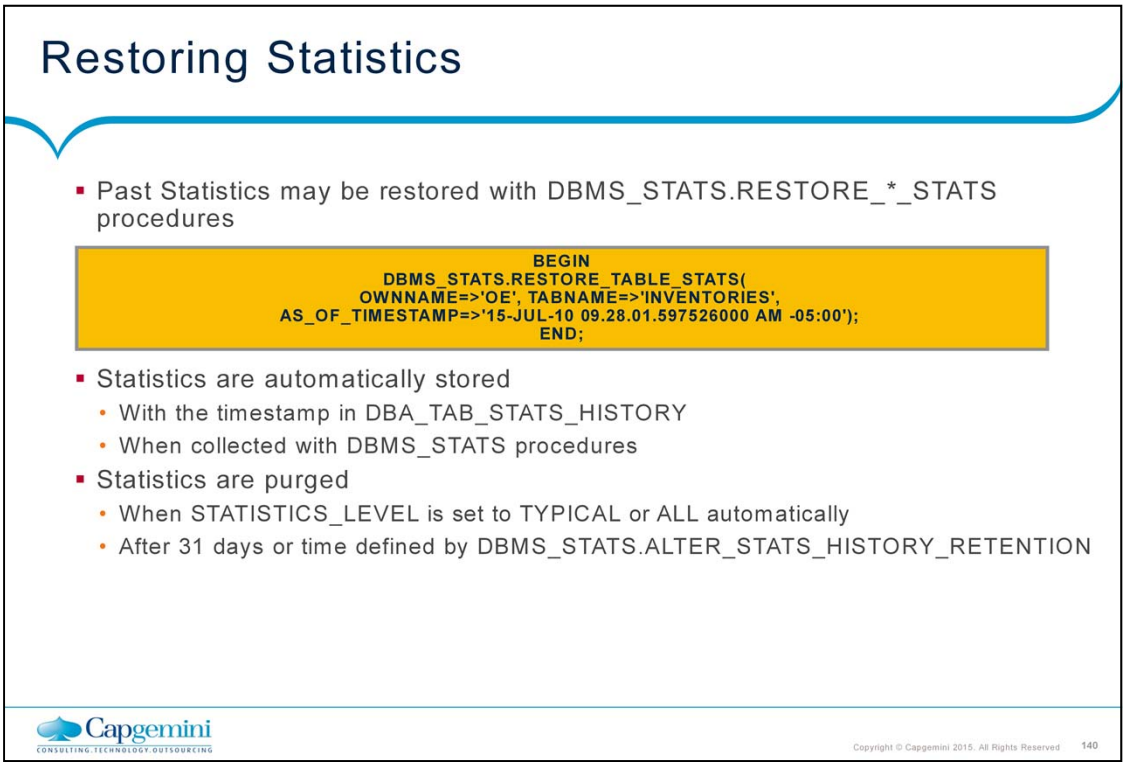
Managing Statistics Collection: Example

- 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.



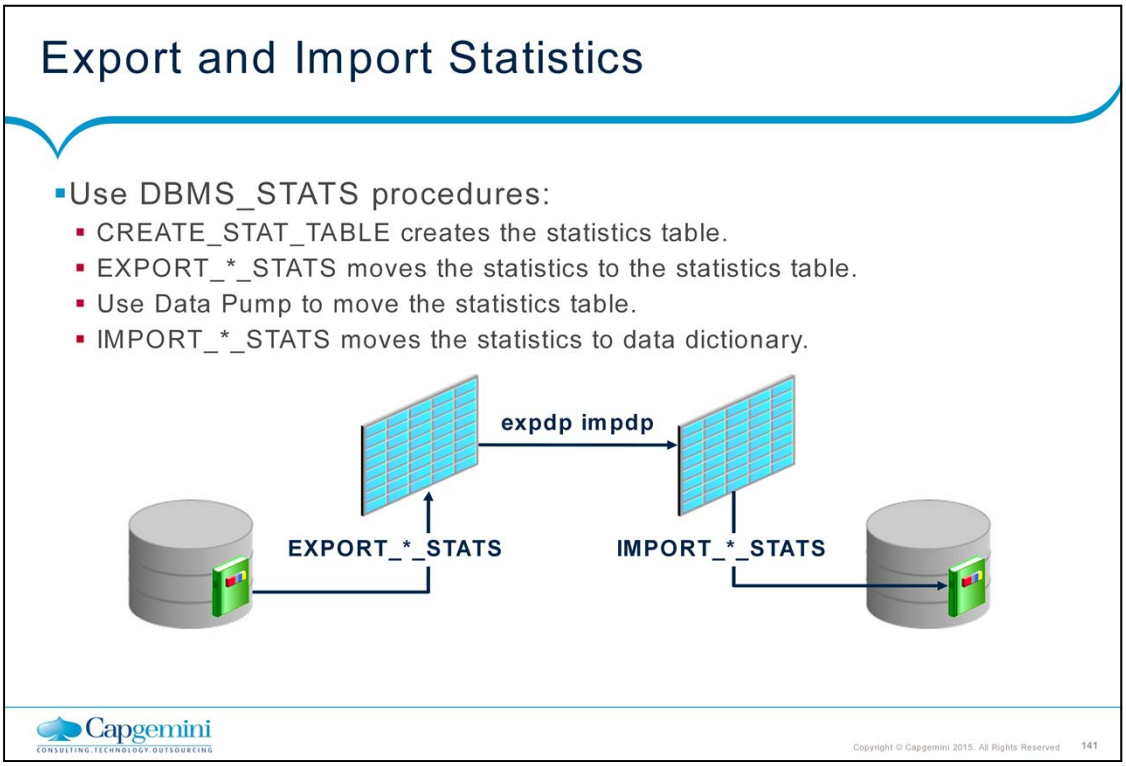
Locking Statistics

- Starting with Oracle Database 10g, 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.



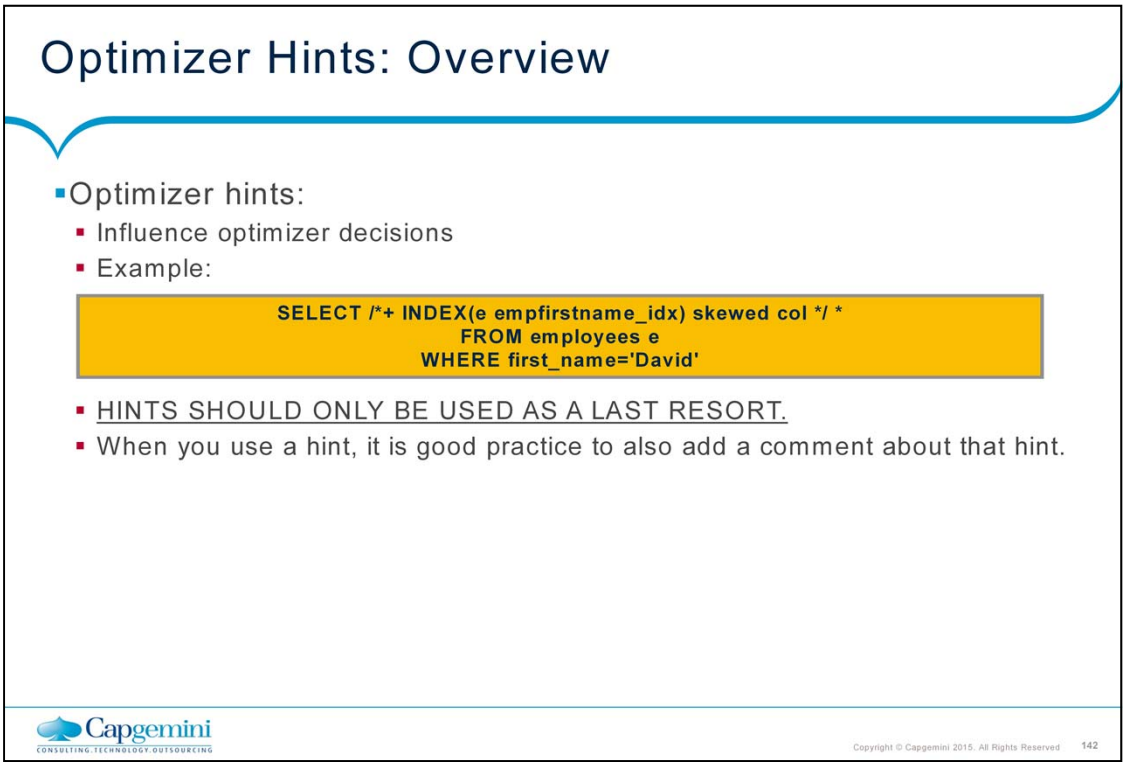
Restoring Statistics

- 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.



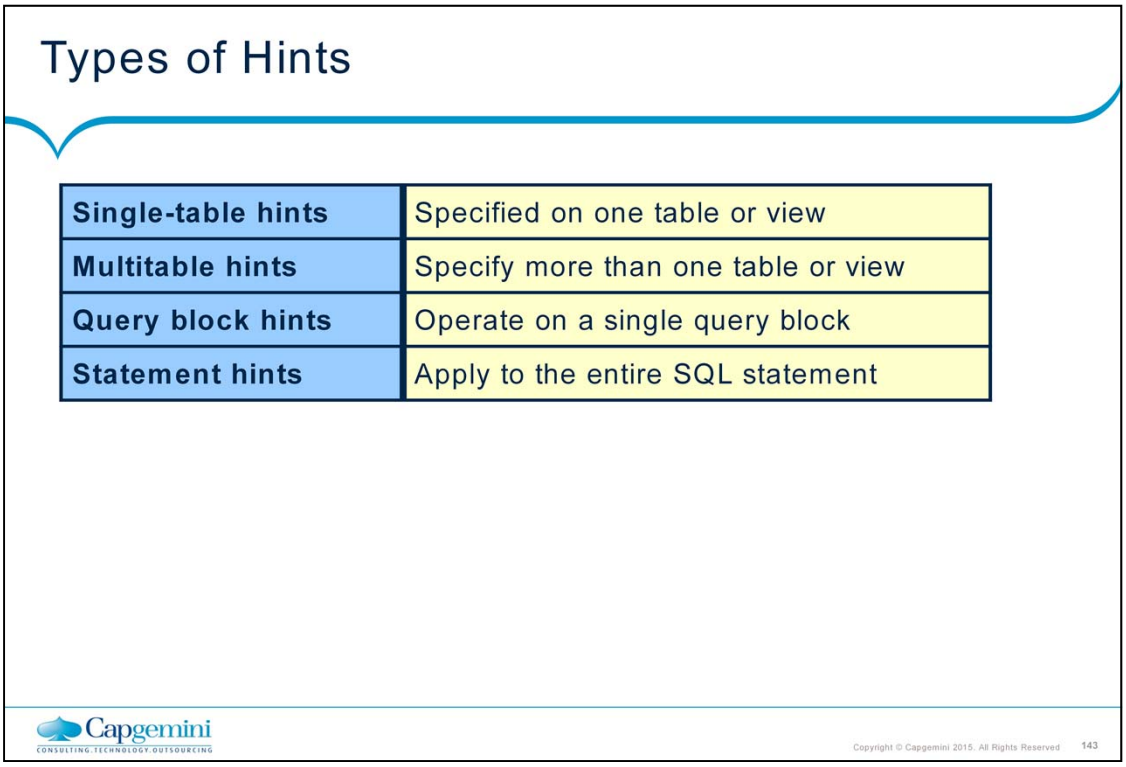
Export and Import Statistics

- 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.



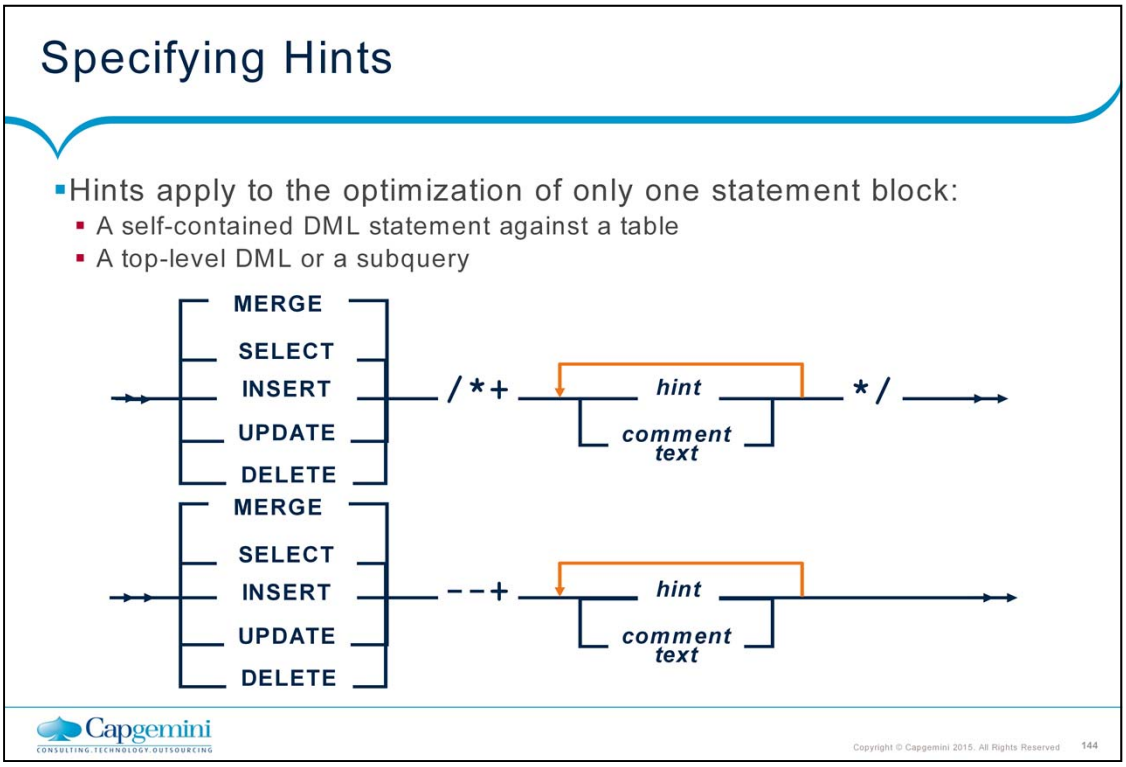
Optimizer Hints: Overview

- Hints enable you to influence decisions made by the optimizer. Hints provide a mechanism to direct the optimizer to select a certain query execution plan based on the specific criteria.
- For example, you might know that a certain index is more selective for certain queries. Based on this information, you might be able to select a more efficient execution plan than the plan that the optimizer recommends. In such a case, use hints to force the optimizer to use the optimal execution plan. This is illustrated in the slide example where you force the optimizer to use the EMPFIRSTNAME_IDX index to retrieve the data. As you can see, you can use comments in a SQL statement to pass instructions to the optimizer.
- The plus sign (+) causes the system to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter. No space is permitted.
- Hints should be used sparingly, and only after you have collected statistics on the relevant tables and evaluated the optimizer plan without hints using the EXPLAIN PLAN statement. Changing database conditions as well as query performance enhancements in subsequent releases can have a significant impact on how hints in your code affect performance.
- In addition, the use of hints involves extra code that must be managed, checked, and controlled.



Types of Hints

- Single-table: Single-table hints are specified on one table or view. INDEX and USE_NL are examples of single-table hints.
- Multitable: Multitable hints are like single-table hints, except that the hint can specify one or more tables or views. LEADING is an example of a multitable hint.
- Query block: Query block hints operate on single query blocks. STAR_TRANSFORMATION and UNNEST are examples of query block hints.
- Statement: Statement hints apply to the entire SQL statement. ALL_ROWS is an example of a statement hint.
- Note: USE_NL(table1 table2) is not considered a multitable hint because it is actually a shortcut for USE_NL(table1) and USE_NL(table2).

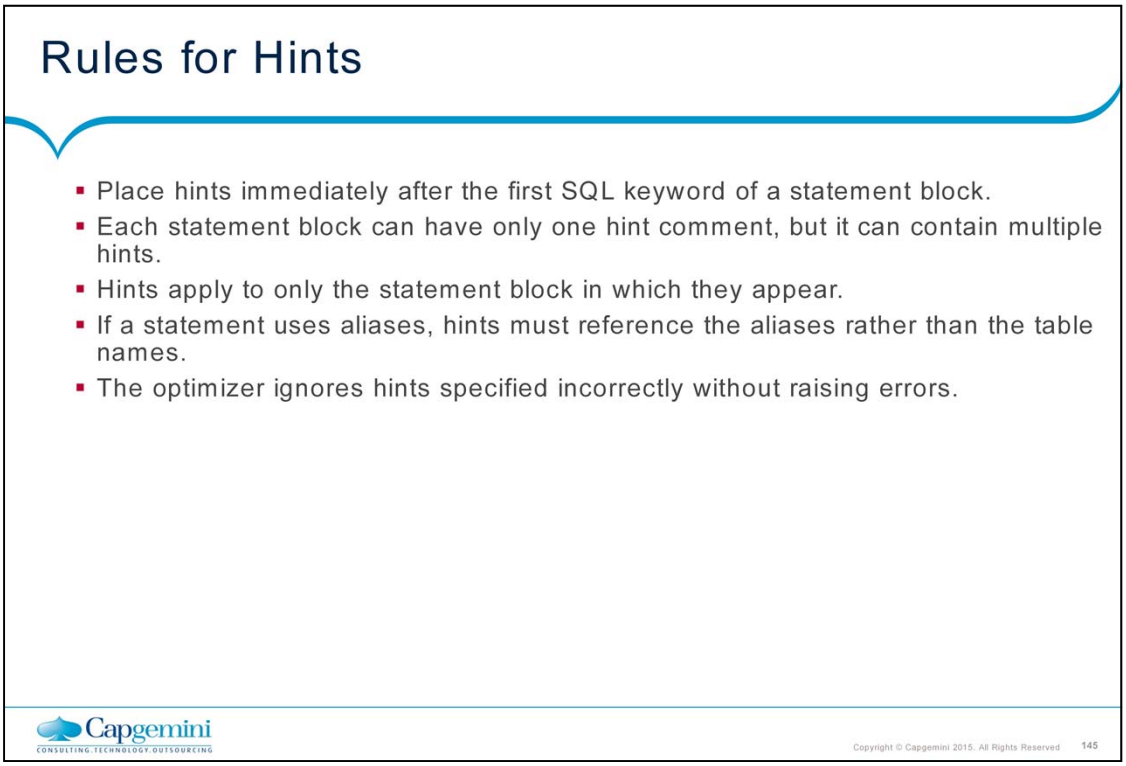


Specifying Hints

- Hints apply to the optimization of only the block of the statement in which they appear. A statement block is:
  - A simple MERGE, SELECT, INSERT, UPDATE, or DELETE statement
  - A parent statement or a subquery of a complex statement
  - A part of a compound query using set operators (UNION, MINUS, INTERSECT)
- For example, a compound query consisting of two component queries combined by the UNION operator has two blocks, one for each component query. For this reason, hints in the first component query apply only to its optimization, not to the optimization of the second component query.

Optimizer Hint Syntax

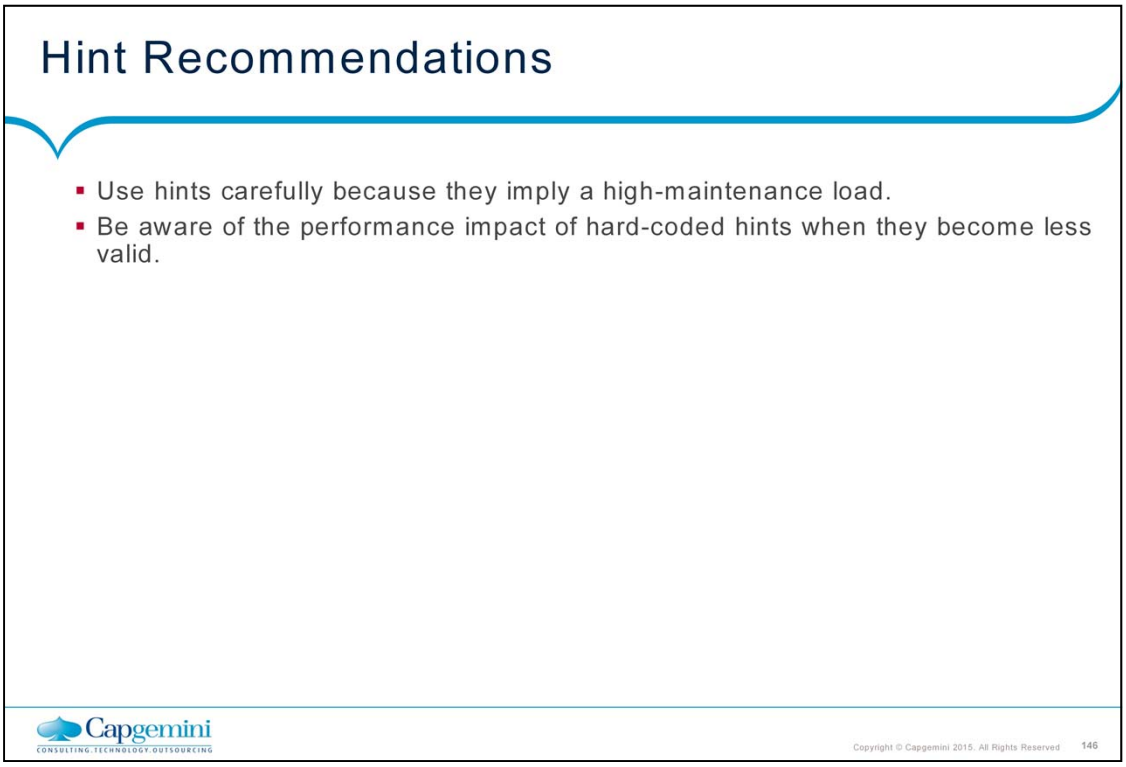
- Enclose hints within the comments of a SQL statement. You can use either style of comment. The hint delimiter (+) must come immediately after the comment delimiter. If you separate them by a space, the optimizer does not recognize that the comment contains hints.



Rules for Hints

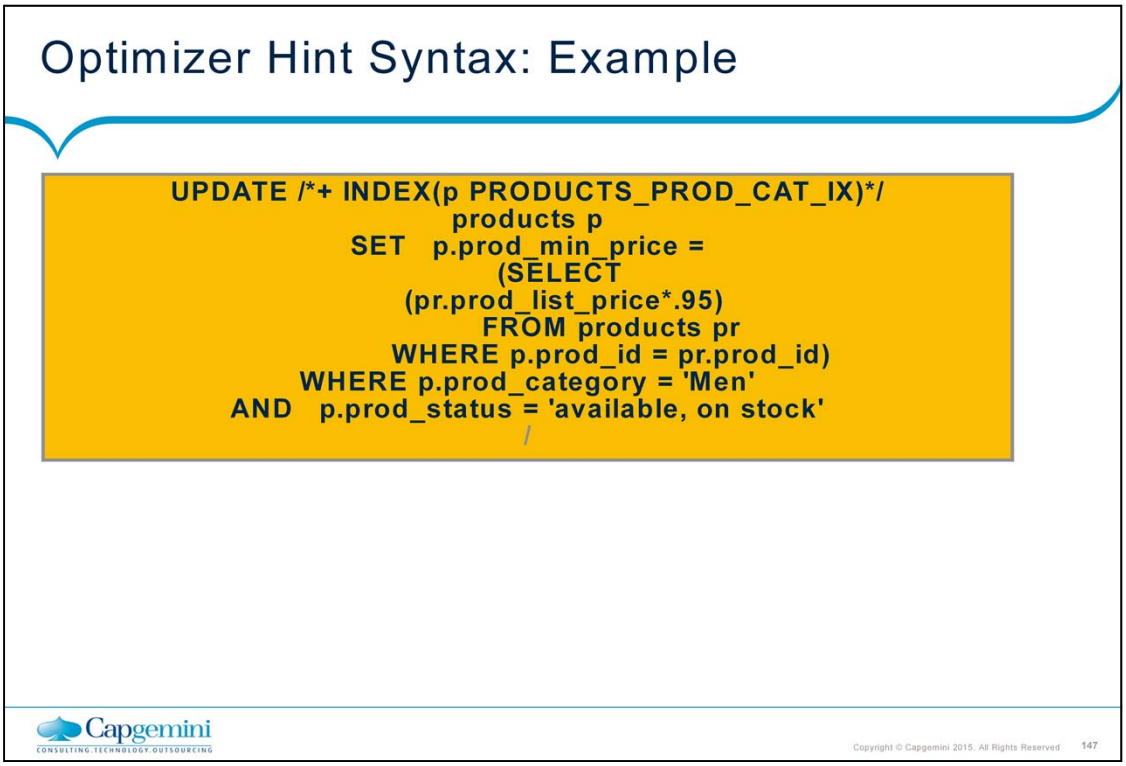
- You must place the hint comment immediately after the first keyword (MERGE, SELECT, INSERT, DELETE, or UPDATE) of a SQL statement block.
- A statement block can have only one comment containing hints, but it can contain many hints inside that comment separated by spaces.
- Hints apply to only the statement block in which they appear and override instance- or session-level parameters.
- If a SQL statement uses aliases, hints must reference the aliases rather than the table names.

- The Oracle optimizer ignores incorrectly specified hints. However, be aware of the following situations:
  - You never get an error message.
  - Other (correctly) specified hints in the same comment are considered.
  - The Oracle optimizer also ignores combinations of conflicting hints.



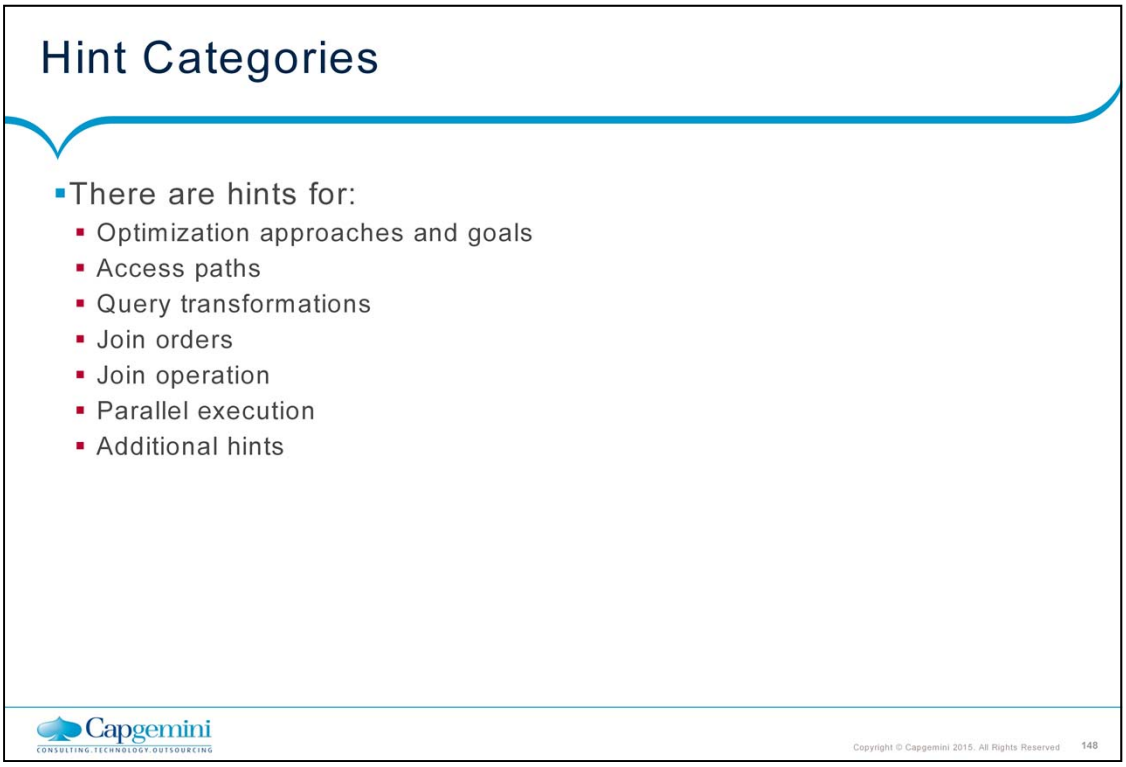
Hint Recommendations

- Use hints as a last remedy when tuning SQL statements.
- Hints may prevent the optimizer from using better execution plans.
- Hints may become less valid (or even invalid) when the database structure or contents change.



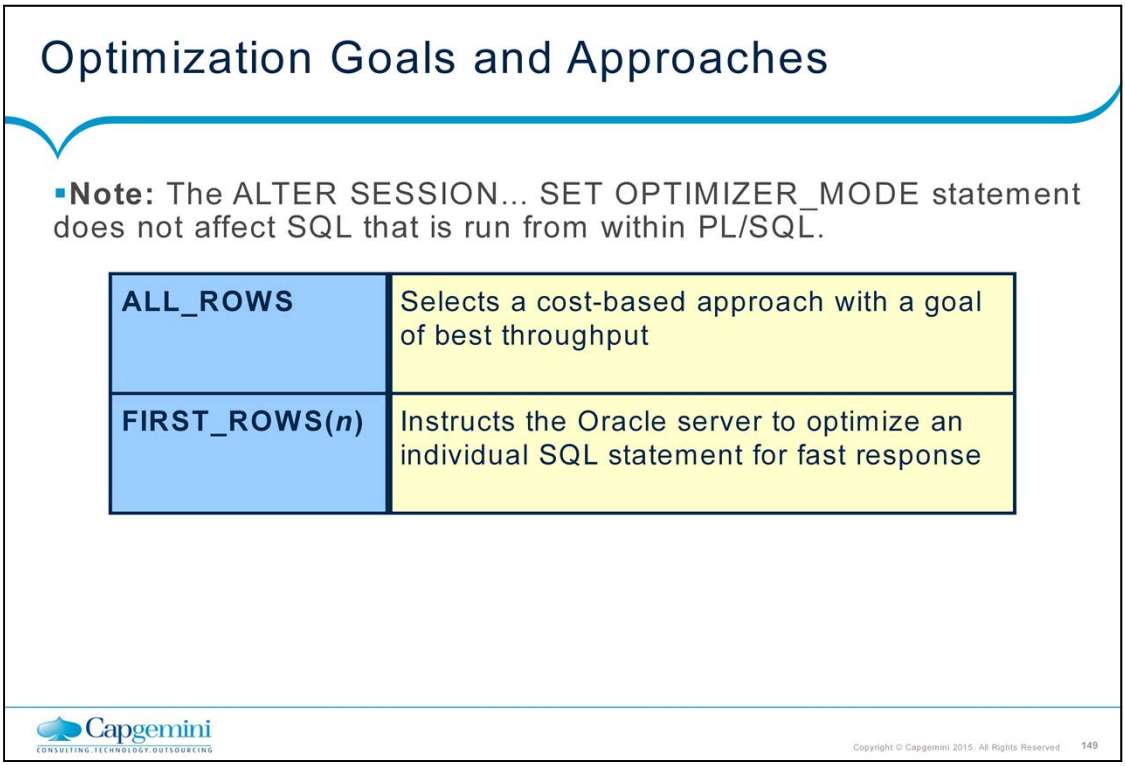
Optimizer Hint Syntax: Example

- The slide shows an example with a hint that advises the cost-based optimizer (CBO) to use the index. The execution plan is as follows:
- Execution Plan
- ----------
- 0 UPDATE STATEMENT Optimizer=ALL_ROWS (Cost=3 …)
- 1 0 UPDATE OF 'PRODUCTS'
- 2 1 TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS' (TABLE) (Cost…)
- 3 2 INDEX (RANGE SCAN) OF 'PRODUCTS_PROD_CAT_IX' (INDEX)
- (cost…)
- 4 1 TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS' (TABLE) (Cost…)
  5 4 INDEX (UNIQUE SCAN) OF 'PRODUCTS_PK' (INDEX (UNIQUE))
  (Cost=0 …)
- The hint shown in the example works only if an index called PRODUCTS_PROD_CAT_IX exists on the PRODUCTS table in the PROD_CATEGORY column.



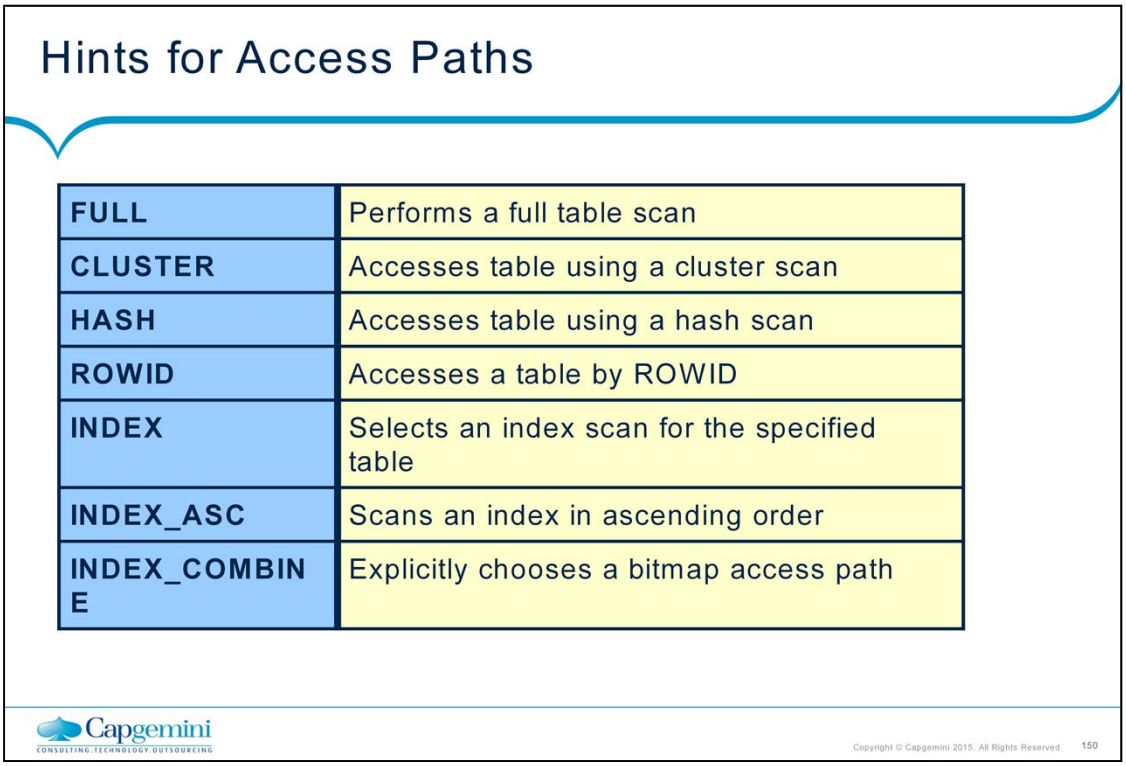
Hint Categories

- Most of these hints are discussed in the following slides. Many of these hints accept the table and index names as arguments.
- Note: Hints for parallel execution is not covered in this course.



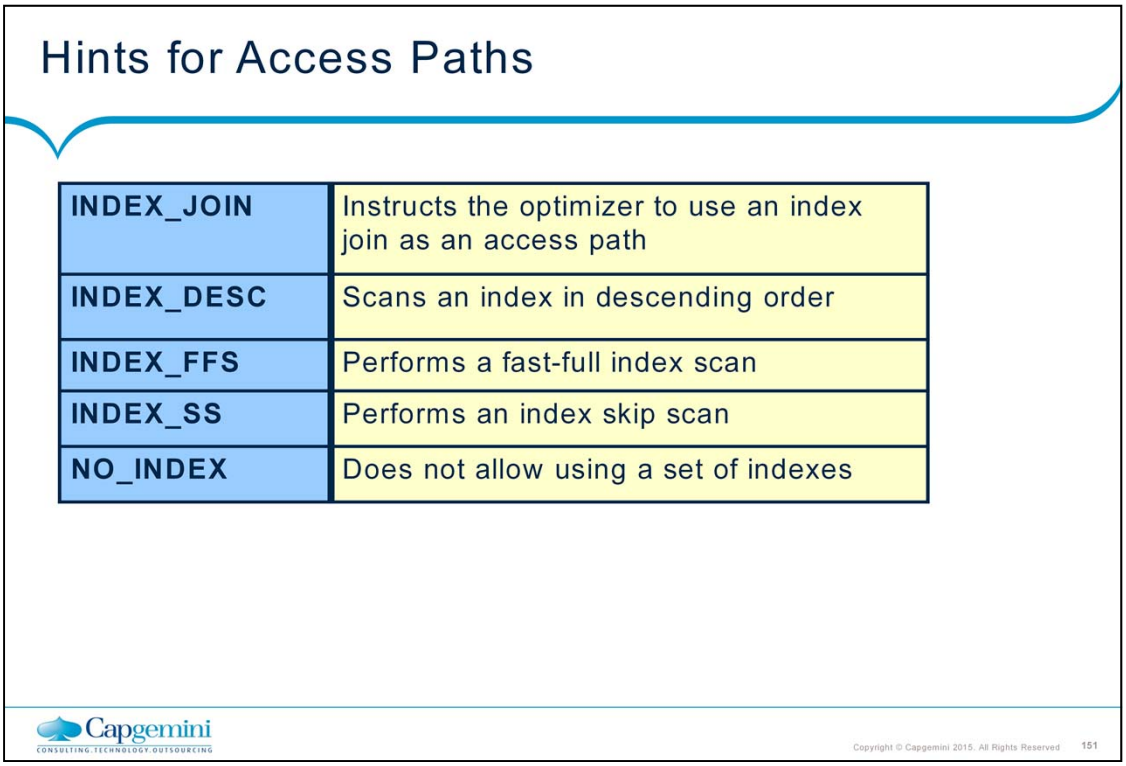
Optimization Goals and Approaches

- ALL_ROWS: The ALL_ROWS hint explicitly selects the cost-based approach to optimize a statement block with a goal of best throughput. That is, minimum total resource consumption.
- FIRST_ROWS(n): The FIRST_ROWS(n) hint (where n is any positive integer) instructs the Oracle server to optimize an individual SQL statement for fast response. It instructs the server to select the plan that returns the first n rows most efficiently. The FIRST_ROWS hint, which optimizes for the best plan to return the first single row, is retained for backward compatibility and plan stability. The optimizer ignores this hint SELECT statement blocks that include any blocking operations, such as sorts or groupings. Such statements cannot be optimized for best response time because Oracle Database must retrieve all rows accessed by the statement before returning the first row. If you specify this hint in any such statement, the database optimizes for best throughput.
- If you specify either the ALL_ROWS or the FIRST_ROWS(n) hint in a SQL statement, and if the data dictionary does not have statistics about tables accessed by the statement, then the optimizer uses default statistical values to estimate the missing statistics and to subsequently select an execution plan.
- If you specify hints for access paths or join operations along with either the ALL_ROWS or FIRST_ROWS(n) hint, the optimizer gives precedence to the access paths and join operations specified by the hints.
- Note: The FIRST_ROWS hints are probably the most useful hints.



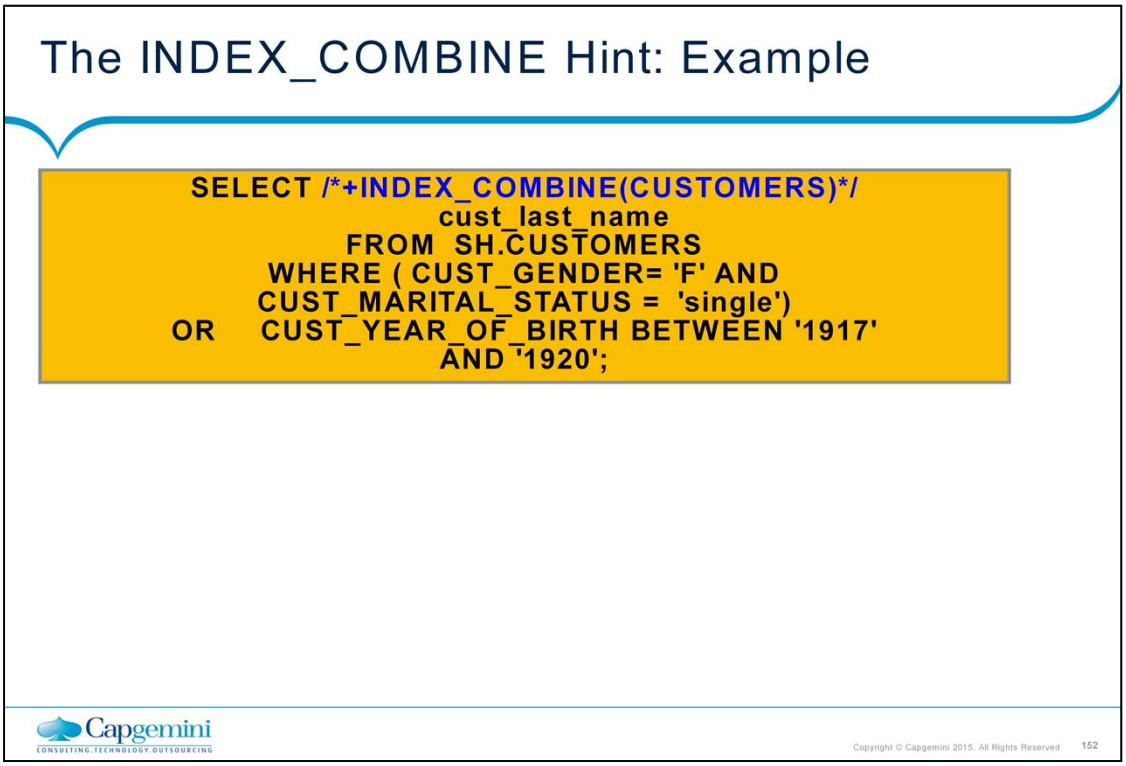
Hints for Access Paths

- Specifying one of these hints causes the optimizer to choose the specified access path only if the access path is available based on the existence of an index and on the syntactic constructs of the SQL statement. If a hint specifies an unavailable access path, the optimizer ignores it. You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, use the alias rather than the table name in the hint. The table name in the hint should not include the schema name if the schema name is present in the statement.
- FULL: The FULL hint explicitly selects a full table scan for the specified table. For example:
  - SELECT /*+ FULL(e) */ employee_id, last_name
  - FROM hr.employees e WHERE last_name LIKE 'K%';
- The Oracle server performs a full table scan on the employees table to execute this statement, even if there is an index on the last_name column that is made available by the condition in the WHERE clause.
- CLUSTER: The CLUSTER hint instructs the optimizer to use a cluster scan to access the specified table. This hint applies only to clustered tables.
- HASH: The HASH hint instructs the optimizer to use a hash scan to access the specified table. This hint applies only to tables stored in a table cluster.



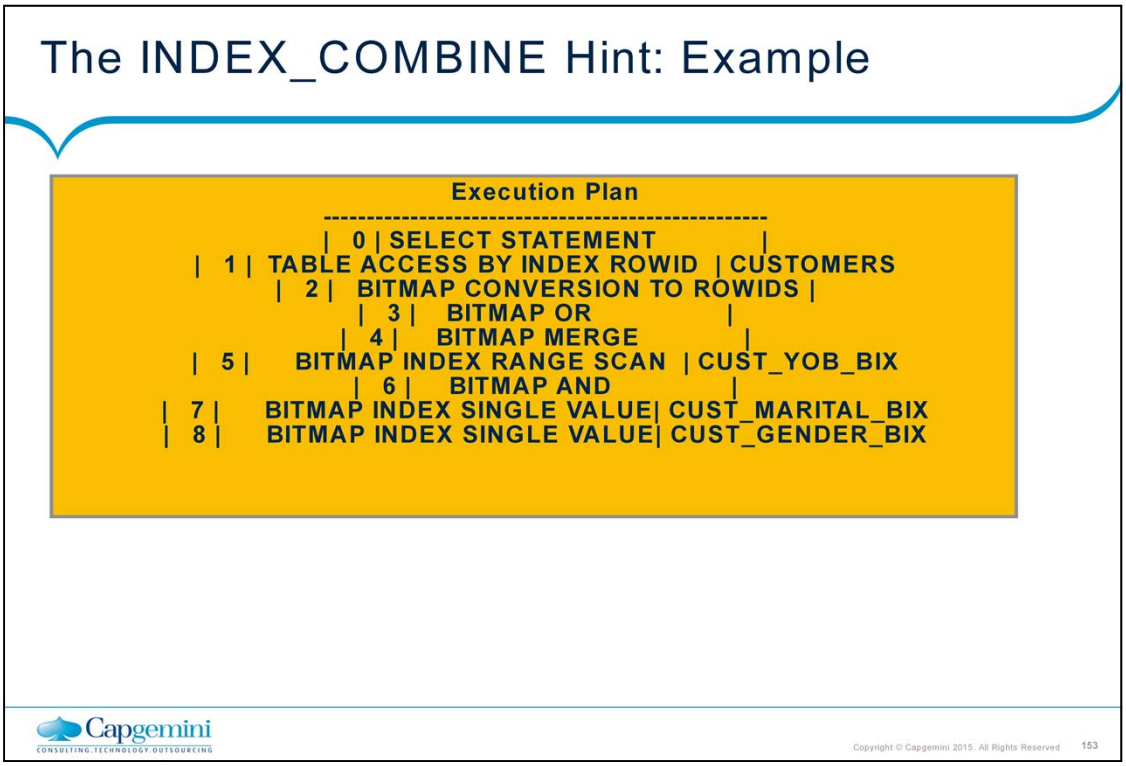
Hints for Access Paths (continued)

- INDEX_JOIN: The INDEX_JOIN hint explicitly instructs the optimizer to use an index join as an access path. For the hint to have a positive effect, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query.
- For example, the following query uses an index join to access the employee_id and department_id columns, both of which are indexed in the employees table:
  - SELECT /*+index_join(employees emp_emp_id_pk emp_department_ix)*/ employee_id, department_id
    FROM hr.employees WHERE department_id > 50;
- INDEX_DESC: The INDEX_DESC hint instructs the optimizer to use a descending index scan for the specified table. If the statement uses an index range scan and the index is ascending, the system scans the index entries in the descending order of their indexed values. In a partitioned index, the results are in the descending order within each partition. For a descending index, this hint effectively cancels out the descending order, resulting in a scan of the index entries in the ascending order. The INDEX_DESC hint explicitly chooses an index scan for the specified table.



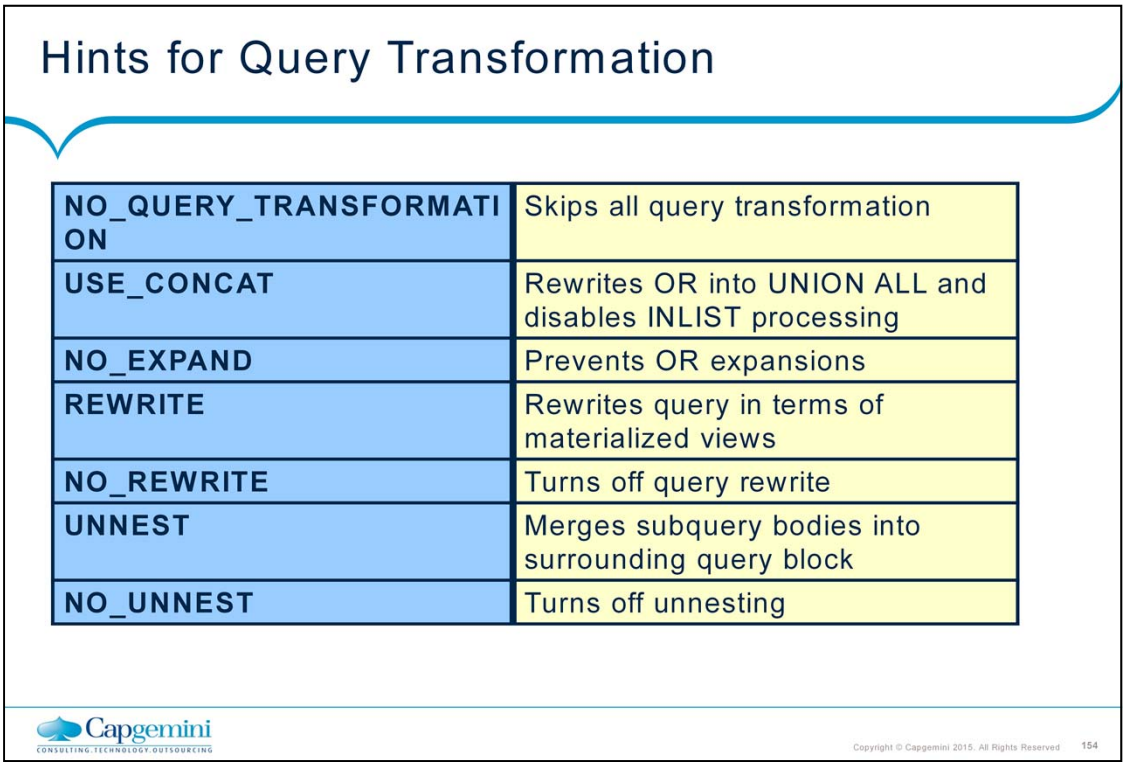
The INDEX_COMBINE Hint: Example

- The INDEX_COMBINE hint is designed for bitmap index operations. Remember the following:
  - If certain indexes are given as arguments for the hint, the optimizer tries to use some combination of those particular bitmap indexes.
  - If no indexes are named in the hint, all indexes are considered to be hinted.
  - The optimizer always tries to use hinted indexes, whether or not it considers them to be cost effective.
- In the example in the slide, suppose that all the three columns that are referenced in the WHERE predicate of the statement in the slide (CUST_MARITAL_STATUS, CUST_GENDER, and CUST_YEAR_OF_BIRTH) have a bitmap index. When you enable AUTOTRACE, the execution plan of the statement might appear as shown in the next slide.



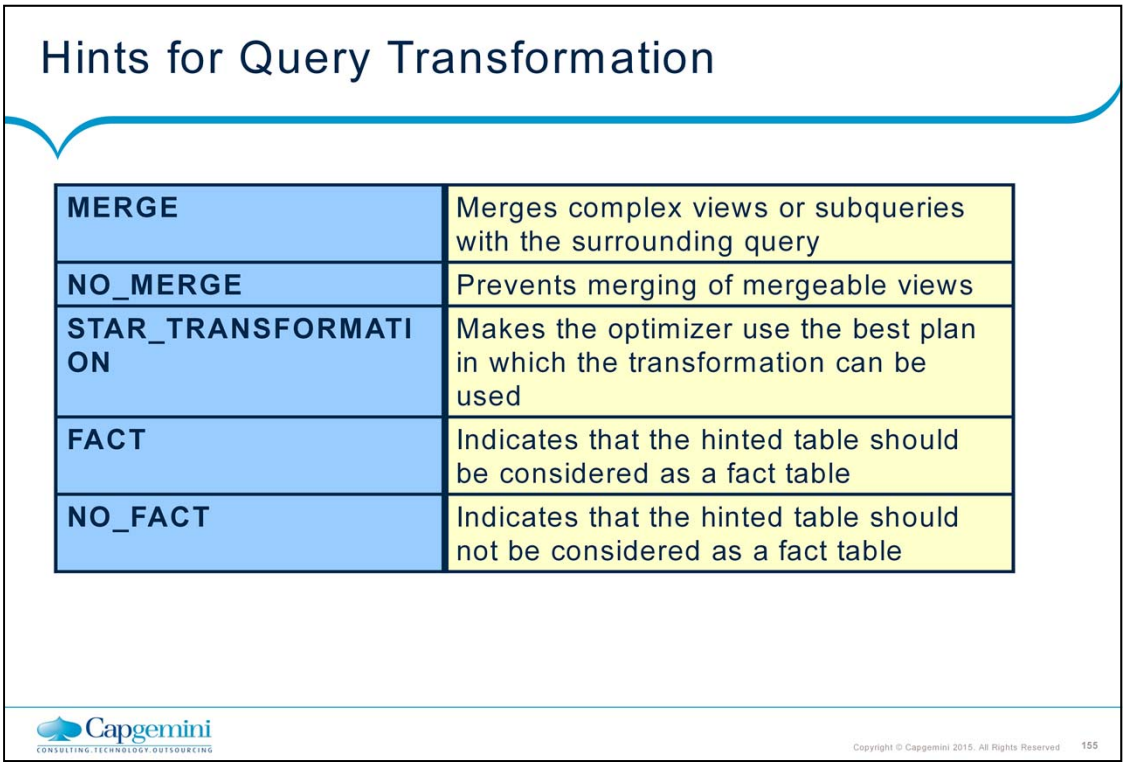
The INDEX_COMBINE Hint: Example (continued)

- In the example in the slide, the following bitmap row sources are used:
  - BITMAP CONVERSION TO ROWIDS: Converts bitmaps into ROWIDs to access a table
  - COUNT: Returns the number of entries if the actual values are not needed
  - BITMAP OR: Computes the bitwise OR of two bitmaps
  - BITMAP AND: Computes the bitwise AND of two bitmaps
  - BITMAP INDEX SINGLE VALUE: Looks up the bitmap for a single key
  - BITMAP INDEX RANGE SCAN: Retrieves bitmaps for a value range
  - BITMAP MERGE: Merges several bitmaps resulting from a range scan into one (using a bitwise AND operator)



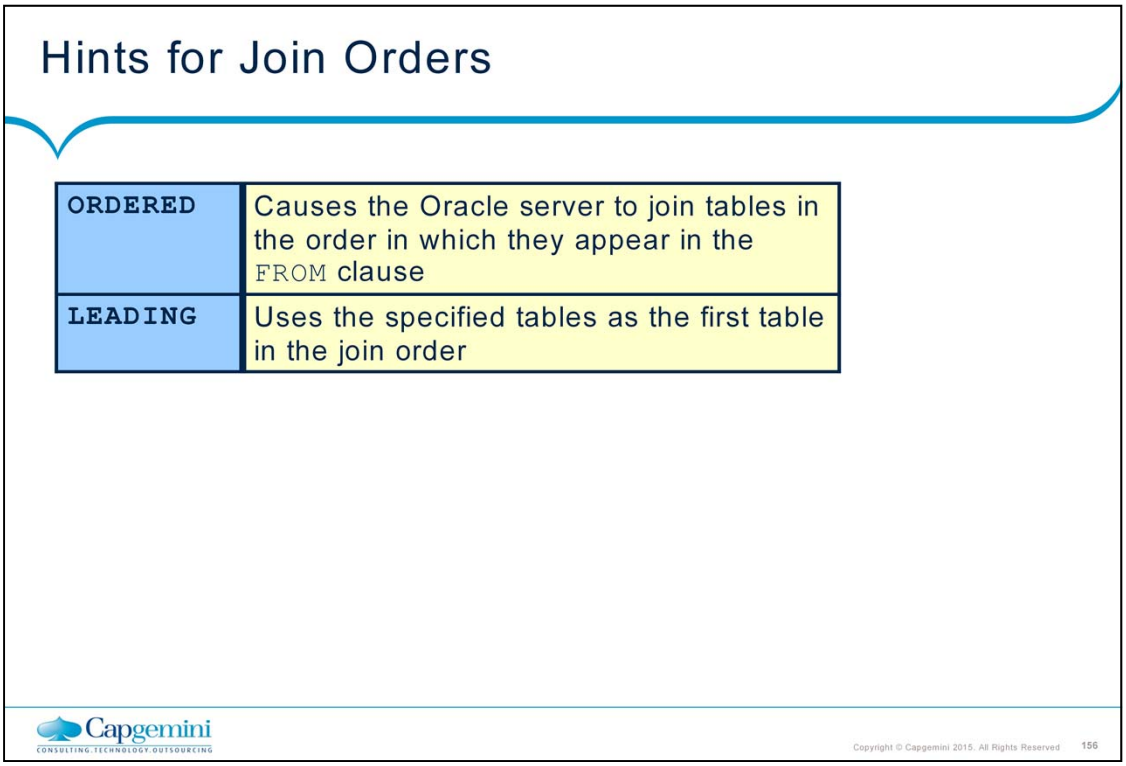
Hints for Query Transformation

- NO_QUERY_TRANSFORMATION: The NO_QUERY_TRANSFORMATION hint instructs the optimizer to skip all query transformations, including but not limited to OR-expansion, view merging, subquery unnesting, star transformation, and materialized view rewrite.
- USE_CONCAT: The USE_CONCAT hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator. Generally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them. The USE_CONCAT hint disables IN-list processing.
- NO_EXPAND: The NO_EXPAND hint prevents the cost-based optimizer from considering OR-expansion for queries having OR conditions or IN-lists in the WHERE clause. Usually, the optimizer considers using OR expansion and uses this method if it decides that the cost is lower than not using it.
- REWRITE: The REWRITE hint instructs the optimizer to rewrite a query in terms of materialized views, when possible, without cost consideration. Use the REWRITE hint with or without a view list. This course does not deal with Materialized Views.
- UNNEST: The UNNEST hint instructs the optimizer to unnest and merge the body of the subquery into the body of the query block that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.



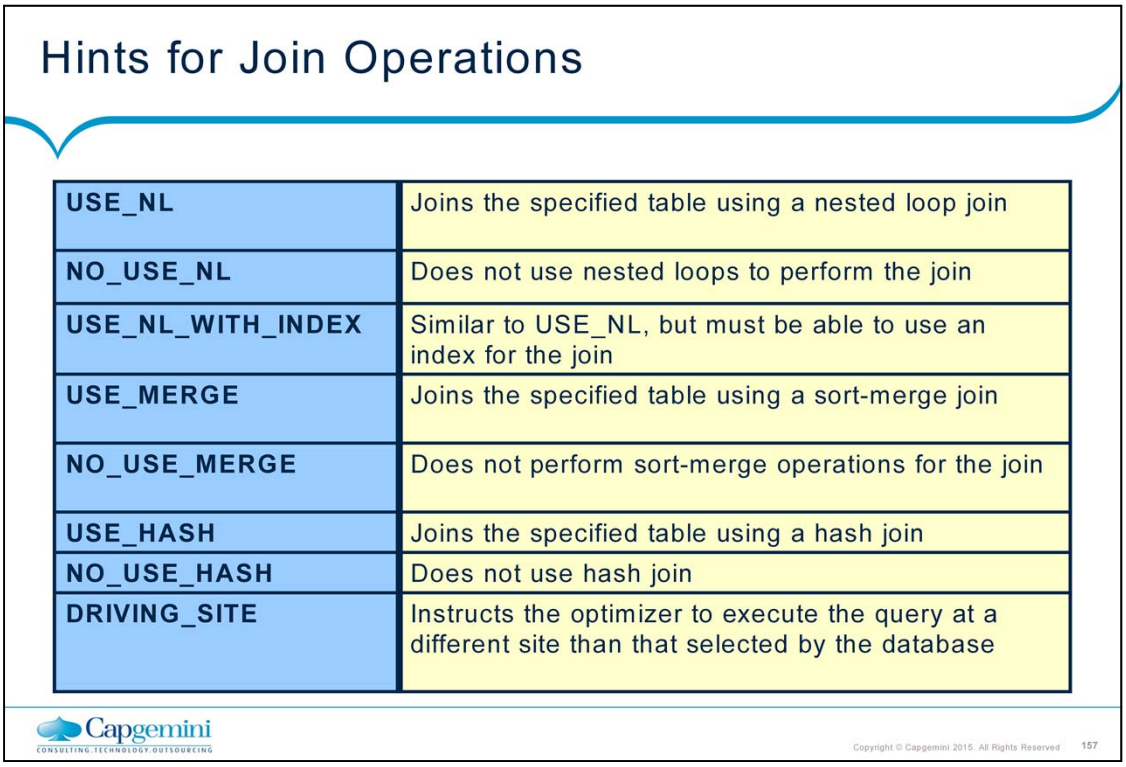
Hints for Query Transformation (continued)

- MERGE: The MERGE hint lets you merge a view for each query. If a view's query contains a GROUP BY clause or a DISTINCT operator in the SELECT list, then the optimizer can merge the view's query into the accessing statement only if complex view merging is enabled. This is the case by default, but you can disable this mechanism using the NO_MERGE hint. Complex merging can also be used to merge an IN subquery into the accessing statement if the subquery is not correlated.
- When the MERGE hint is used without an argument, it should be placed in the view query block. When MERGE is used with the view name as an argument, it should be placed in the surrounding query.
- NO_MERGE: The NO_MERGE hint causes the Oracle server not to merge views that can be merged. This hint gives the user more influence over the way in which the view is accessed. When the NO_MERGE hint is used without an argument, it should be placed in the view query block. When NO_MERGE is used with the view name as an argument, it should be placed in the surrounding query.



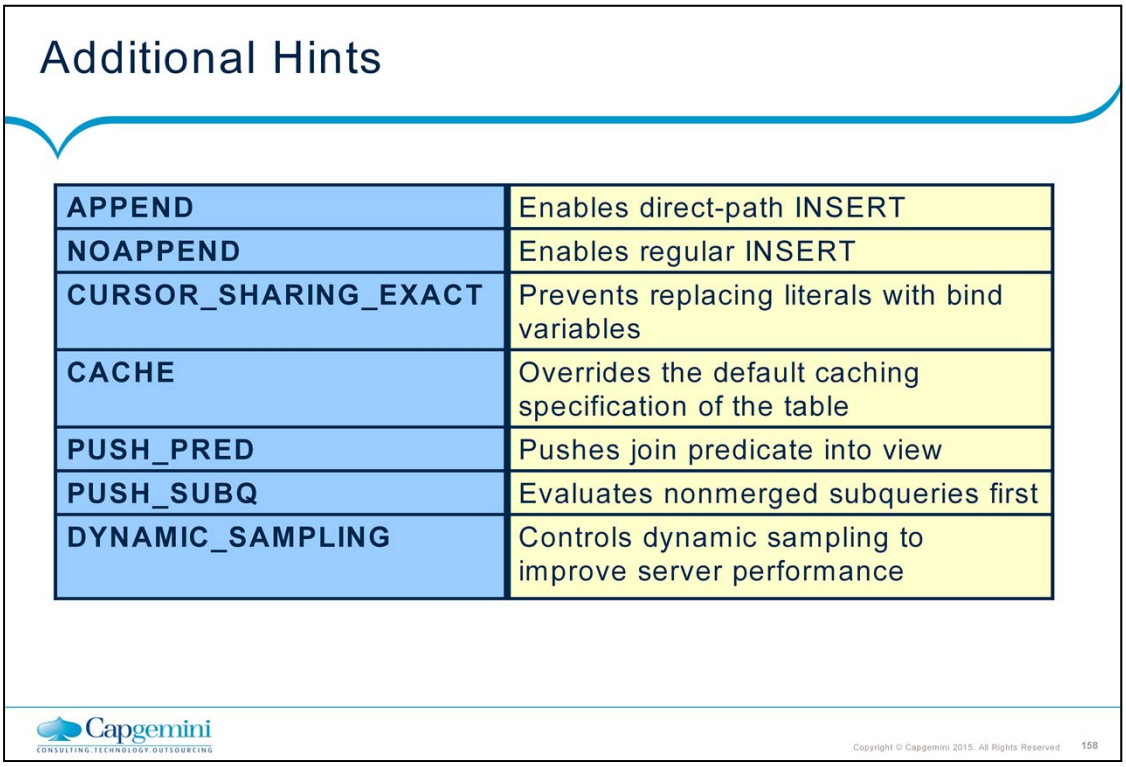
Hints for Join Orders

- The following hints are used to suggest join orders:
- ORDERED: The ORDERED hint causes the Oracle server to join tables in the order in which they appear in the FROM clause. If you omit the ORDERED hint from a SQL statement performing a join, the optimizer selects the order in which to join the tables. You might want to use the ORDERED hint to specify a join order if you know something that the optimizer does not know about the number of rows that are selected from each table. With a nested loops example, the most precise method is to order the tables in the FROM clause in the order of the keys in the index, with the large table at the end. Then use the following hints:
- /*+ ORDERED USE_NL(FACTS) INDEX(facts fact_concat) */
- Here, facts is the table and fact_concat is the index. A more general method is to use the STAR hint.
- LEADING: The LEADING hint instructs the optimizer to use the specified set of tables as the prefix in the execution plan. The LEADING hint is ignored if the tables specified cannot be joined first in the order specified because of dependencies in the join graph. If you specify two or more LEADING hints on different tables, all the hints are ignored. If you specify the ORDERED hint, it overrides all LEADING hints.



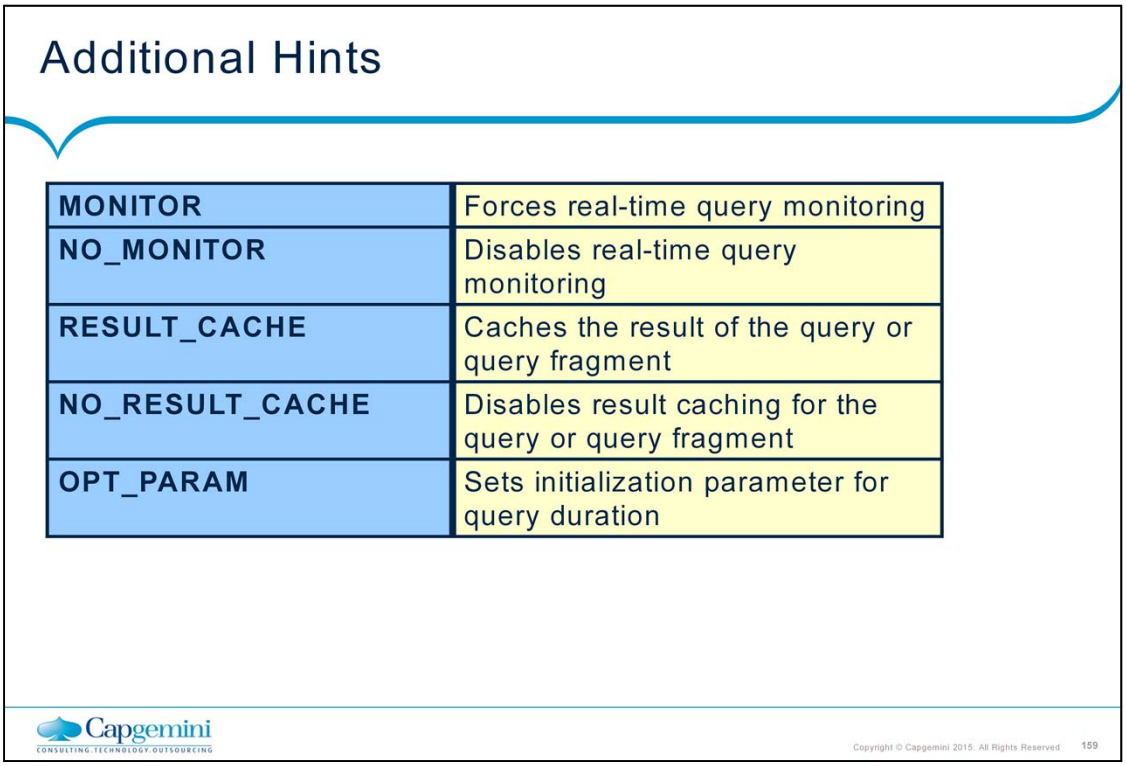
Hints for Join Operations

- Each hint described here suggests a join operation for a table. In the hint, you must specify a table exactly the same way as it appears in the statement. If the statement uses an alias for the table, you must use the alias rather than the table name in the hint. However, the table name in the hint should not include the schema name if the schema name is present in the statement. Use of the USE_NL and USE_MERGE hints is recommended with the ORDERED hint. The Oracle server uses these hints when the referenced table is forced to be the inner table of a join; the hints are ignored if the referenced table is the outer table.
- USE_NL: The USE_NL hint causes the Oracle server to join each specified table to another row source with a nested loops join, using the specified table as the inner table. If you want to optimize the statement for best response time or for the minimal elapsed time that is necessary to return the first row selected by the query, rather than for best throughput, then you can force the optimizer to select a nested loop join by using the USE_NL hint.
- USE_NL_WITH_INDEX: The USE_NL_WITH_INDEX hint is similar to the USE_NL hint. However, if no index is specified, the optimizer must be able to use some index with at least one join predicate as the index key. If an index is specified, the optimizer must be able to use that index with at least one join predicate as the index key.
- NO_USE_NL: The NO_USE_NL hint causes the optimizer to exclude the nested loops join.



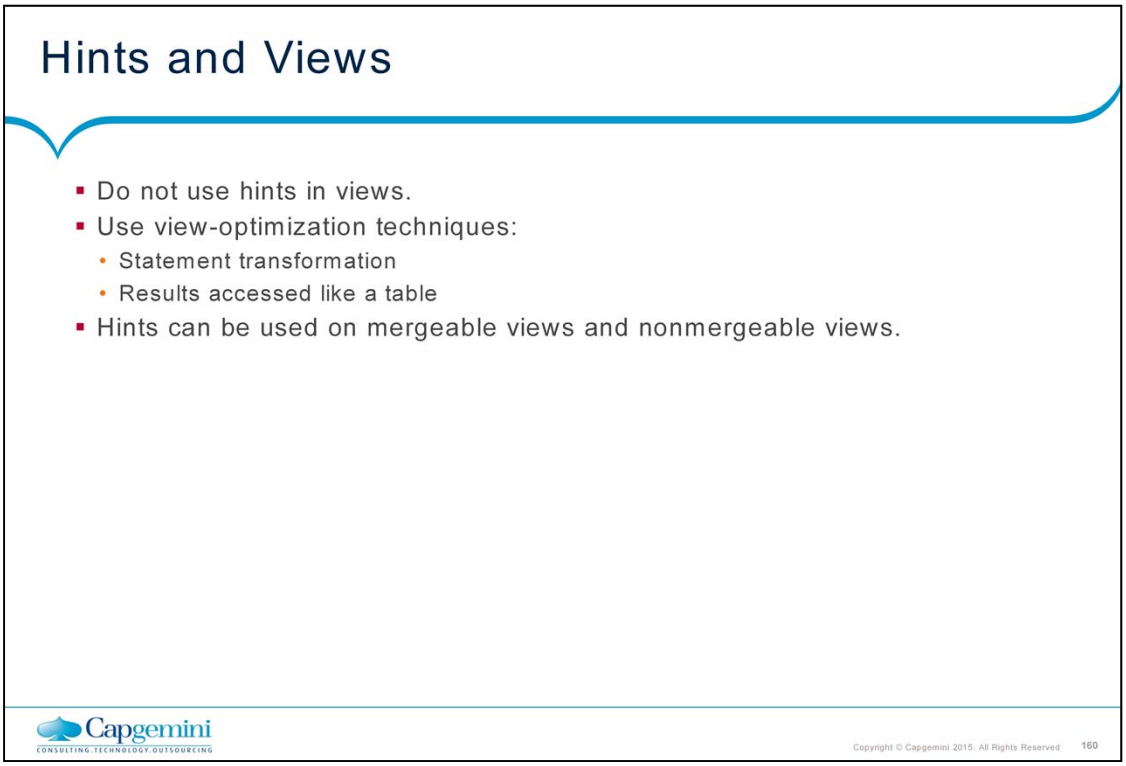
Additional Hints

- APPEND: The APPEND hint lets you enable direct-path INSERT if your database runs in serial mode. Your database is in serial mode if you are not using Enterprise Edition. Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode. In direct-path INSERT, data is appended to the end of the table rather than using existing space currently allocated to the table. As a result, direct-path INSERT can be considerably faster than the conventional INSERT. Note: In Enterprise Edition, a session must be placed in parallel mode for direct-path insert to be the default.
- NOAPPEND: The NOAPPEND hint disables direct-path INSERT by disabling parallel mode for the duration of the INSERT statement. (Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode.)
- CURSOR_SHARING_EXACT: The Oracle server can replace literals in SQL statements with bind variables if it is safe to do so. This is controlled with the CURSOR_SHARING startup parameter. The CURSOR_SHARING_EXACT hint causes this behavior to be disabled. In other words, the Oracle server executes the SQL statement without any attempt to replace literals with bind variables.



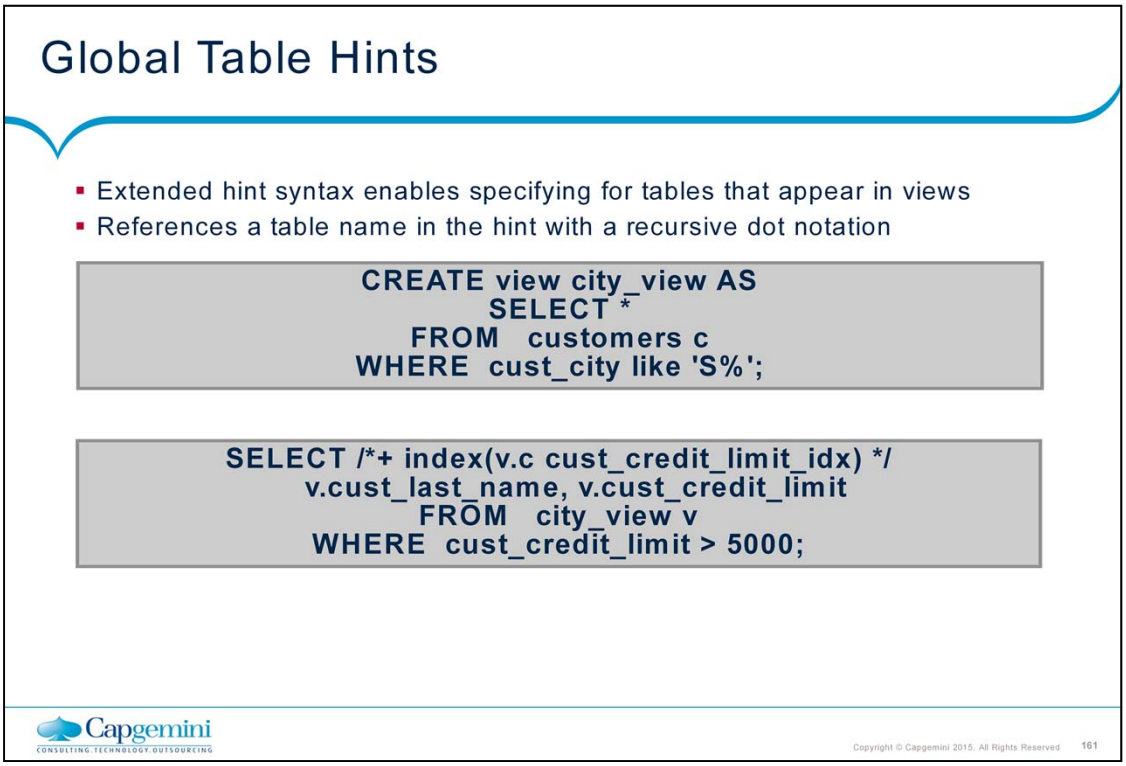
Additional Hints (continued)

- MONITOR: The MONITOR hint forces real-time SQL monitoring for the query, even if the statement is not long running. This hint is valid only when the CONTROL_MANAGEMENT_PACK_ACCESS parameter is set to DIAGNOSTIC+TUNING.
- NO_MONITOR: The NO_MONITOR hint disables real-time SQL monitoring for the query.
- RESULT_CACHE: The RESULT_CACHE hint instructs the database to cache the results of the current query or query fragment in memory and then to use the cached results in future executions of the query or query fragment.
- NO_RESULT_CACHE: The optimizer caches query results in the result cache if the RESULT_CACHE_MODE initialization parameter is set to FORCE. In this case, the NO_RESULT_CACHE hint disables such caching for the current query.
- OPT_PARAM: The OPT_PARAM hint lets you set an initialization parameter for the duration of the current query only. This hint is valid only for the following parameters: OPTIMIZER_DYNAMIC_SAMPLING, OPTIMIZER_INDEX_CACHING, OPTIMIZER_INDEX_COST_ADJ, OPTIMIZER_SECURE_VIEW_MERGING, and STAR_TRANSFORMATION_ENABLED



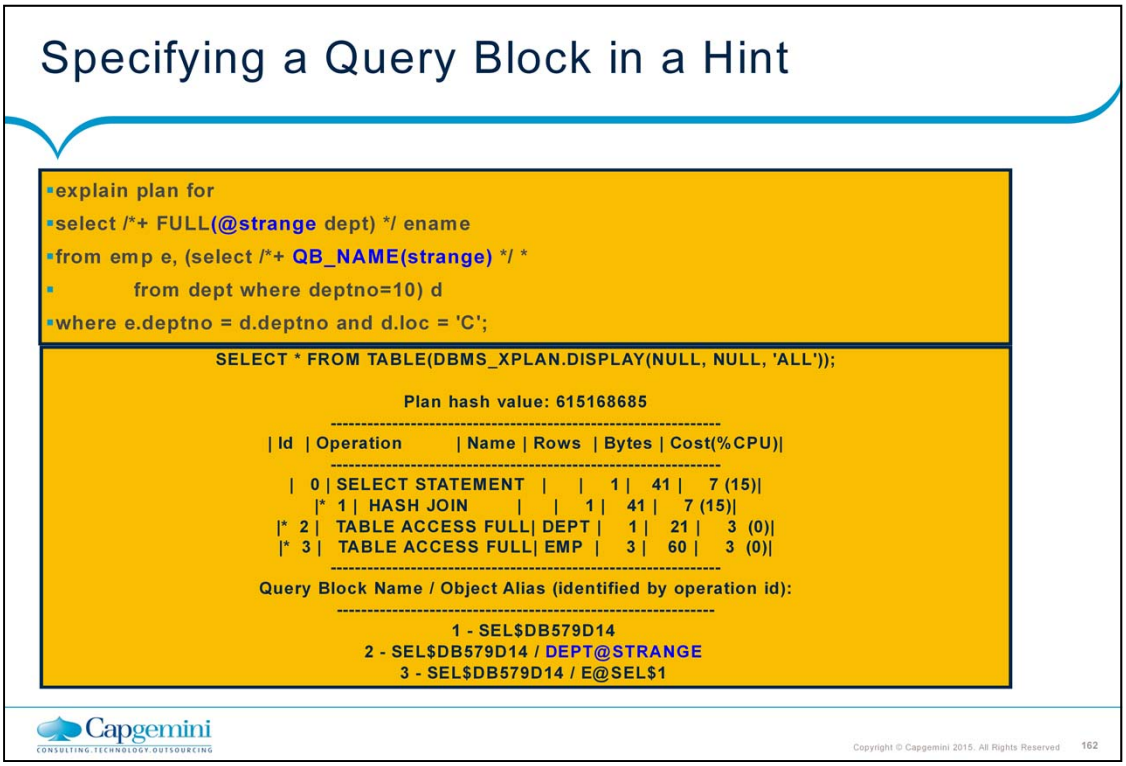
Hints and Views

- You should not use hints in or on views because views can be defined in one context and used in another; such hints can result in unexpected plans. In particular, hints in views are handled differently from hints on views depending on whether or not the view is mergeable into the top-level query.
- View Optimization
- The statement is normally transformed into an equivalent statement that accesses the view's base tables. The optimizer can use one of the following techniques to transform the statement:
    - Merge the view's query into the referencing query block in the accessing statement.
    - Push the predicate of the referencing query block inside the view.
- When these transformations are impossible, the view's query is executed and the result is accessed as if it were a table. This appears as a VIEW step in execution plans.



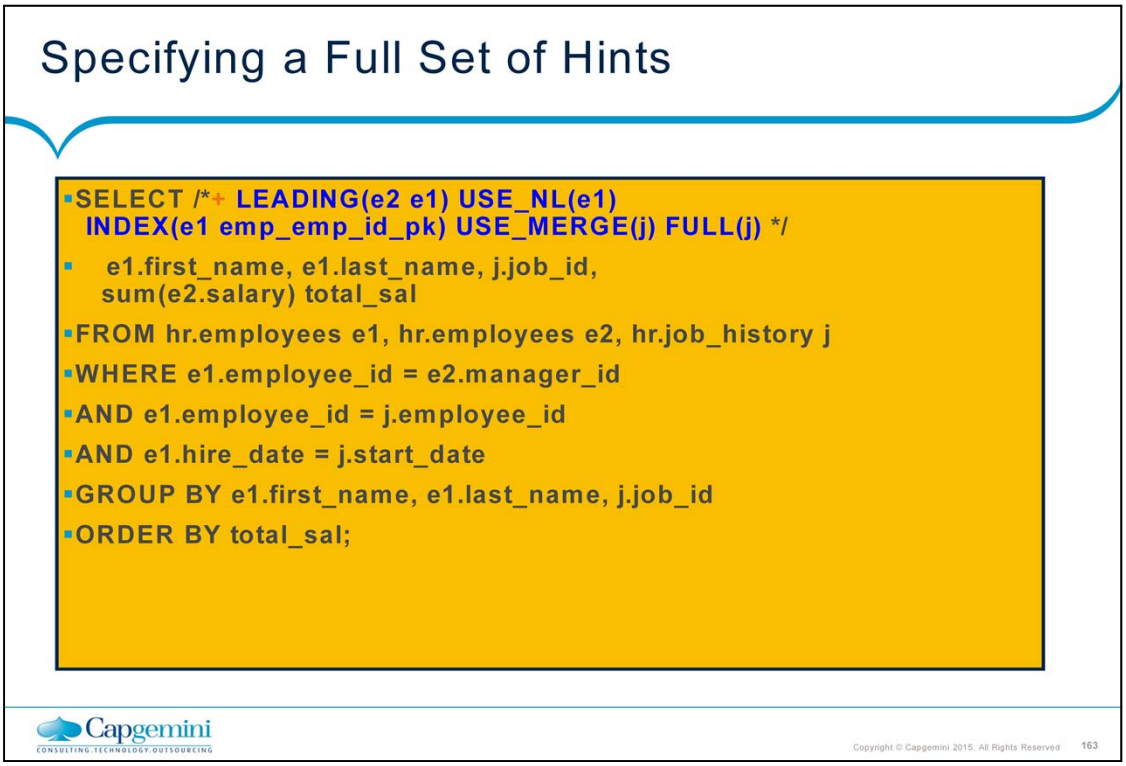
Global Table Hints

- Hints that specify a table generally refer to tables in the DELETE, SELECT, or UPDATE query block in which the hint occurs, rather than to tables inside any views that are referenced by the statement. When you want to specify hints for tables that appear inside views, it is recommended that you use global hints instead of embedding the hint in the view.
- The table hints can be transformed into global hints by using an extended table specification syntax that includes view names with the table name as shown in the slide. In addition, an optional query block name can precede the table specification.
- For example, by using the global hint structure, you can avoid the modification of a view with the specification of an index hint in the body of view.
- Note: If a global hint references a table name or alias that is used twice in the same query (for example, in a UNION statement), the hint applies to only the first instance of the table (or alias).



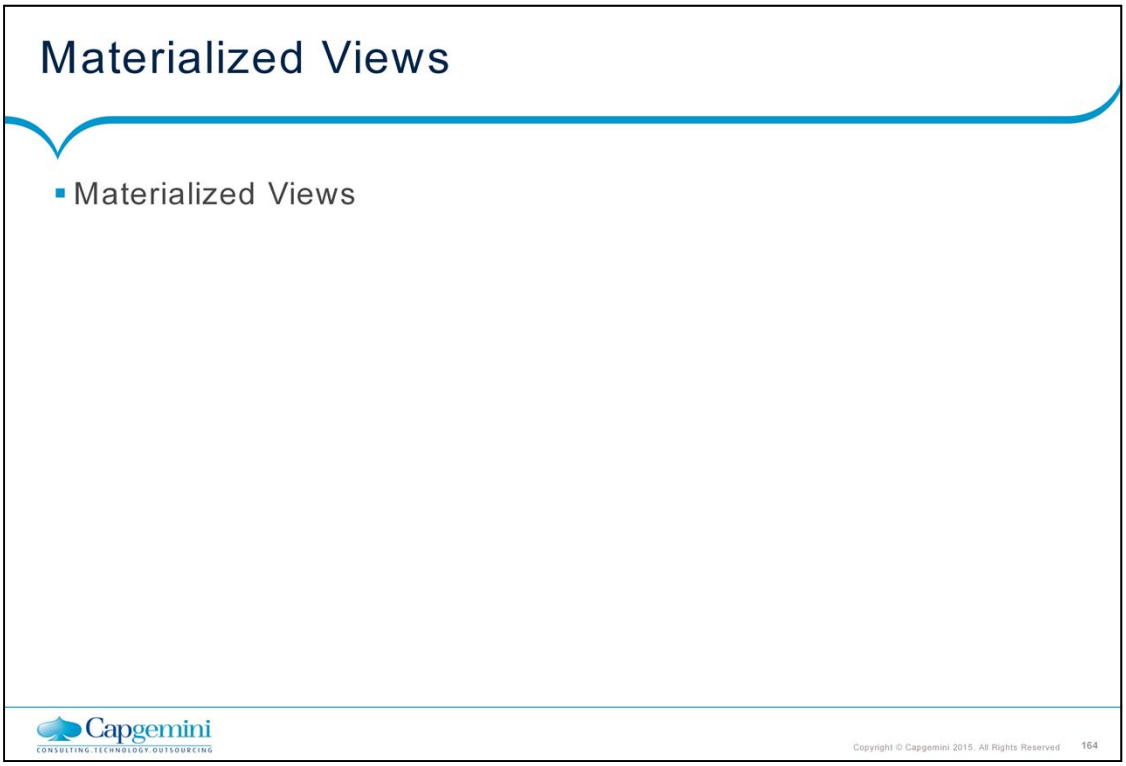
Specifying a Query Block in a Hint

- You can specify an optional query block name in many hints to specify the query block to which the hint applies. This syntax lets you specify in the outer query a hint that applies to an inline view.
- The syntax of the query block argument is of the @queryblock form, where queryblock is an identifier that specifies a query block in the query. The queryblock identifier can either be system-generated or user-specified. When you specify a hint in the query block itself to which the hint applies, you do not have to specify the @queryblock syntax.
- The slide gives you an example. You can see that the SELECT statement uses an inline view. The corresponding query block is given the name strange through the use of the QB_NAME hint.
- The example assumes that there is an index on the DEPTNO column of the DEPT table so that the optimizer would normally choose that index to access the DEPT table. However, because you specify the FULL hint to apply to the strange query block in the main query block, the optimizer does not use the index in question. You can see that the execution plan exhibits a full table scan on the DEPT table. In addition, the output of the plan clearly shows the system-generated names for each query block in the original query.



Specifying a Full Set of Hints

- When using hints, you might sometimes need to specify a full set of hints to ensure the optimal execution plan. For example, if you have a very complex query consisting of many table joins, and if you specify only the INDEX hint for a given table, then the optimizer needs to determine the remaining access paths to be used as well as the corresponding join methods. Therefore, even though you gave the INDEX hint, the optimizer might not necessarily use that hint because the optimizer might have determined that the requested index cannot be used due to the join methods and access paths that were selected by the optimizer.
- In the example, the LEADING hint specifies the exact join order to be used. The join methods to be used on the different tables are also specified.



Refresh Methods

- COMPLETE refreshes by recalculating the detail query of the materialized view. This can be accomplished by deleting the table or truncating it. Complete refresh reexecutes the materialized view query, thereby completely recalculating the contents of the materialized view from the detail tables.
- FAST refreshes by incrementally adding the new data that has been inserted or updated in the tables.
- FORCE applies fast refresh if possible; otherwise, it applies COMPLETE refresh.
- NEVER prevents the materialized view from being refreshed with any Oracle refresh mechanism or procedure.

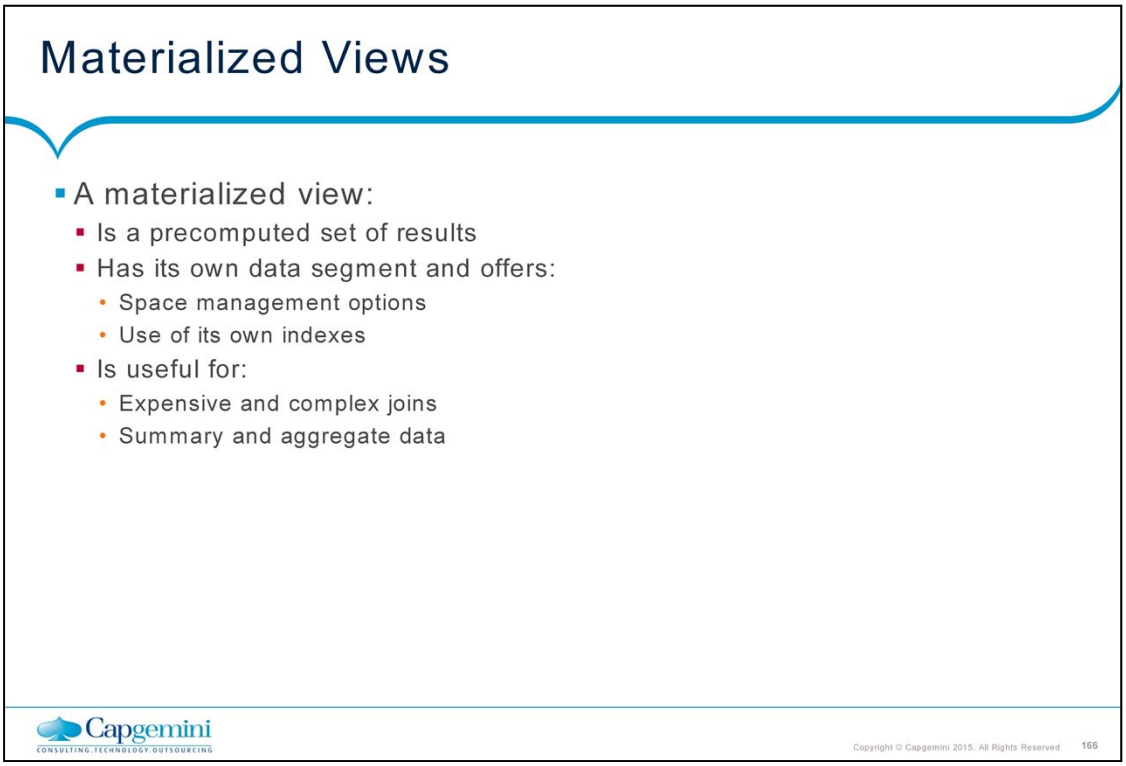
- Some types of nested materialized views cannot be fast refreshed. Use DBMS_MVIEW.EXPLAIN_MVIEW to identify those types of materialized views. You can refresh a tree of nested materialized views in the appropriate dependency order by specifying the nested = TRUE parameter with the DBMS_MVIEW.REFRESH method.

Objectives

- After completing this lesson, you should be able to do the following:
  - Identify the characteristics and benefits of materialized views
  - Use materialized views to enable query rewrites
  - Verify the properties of materialized views
  - Perform refreshes on materialized views

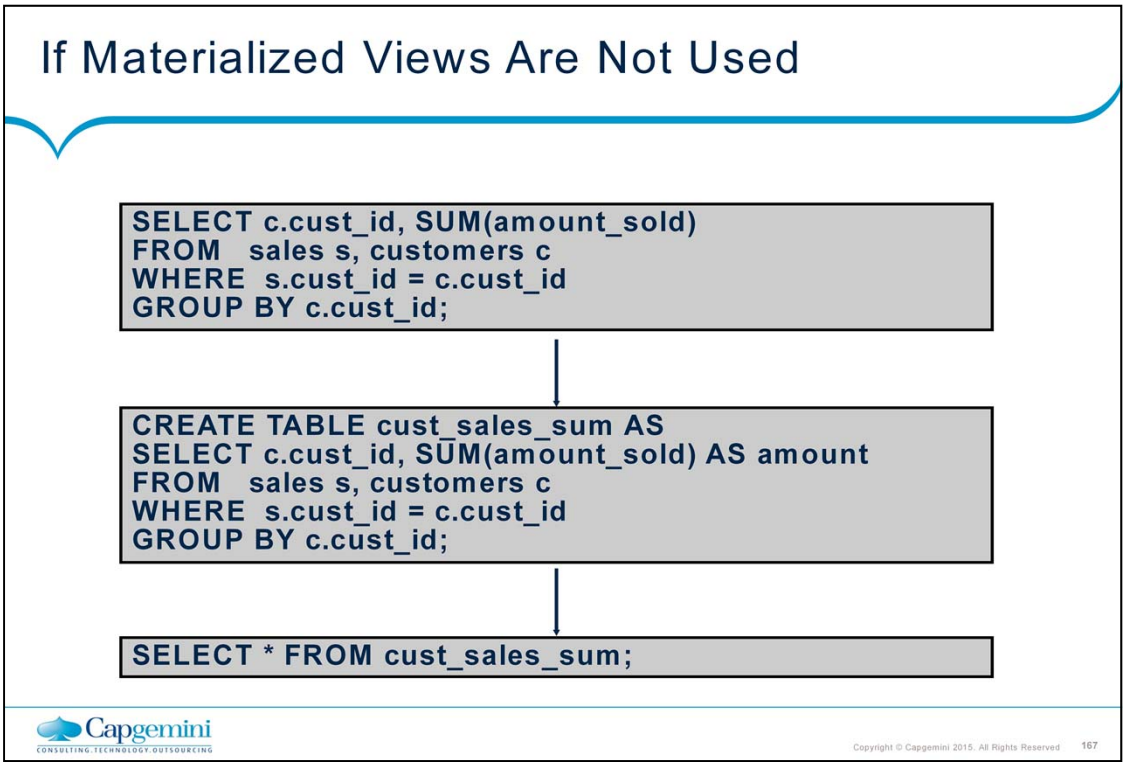
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 165



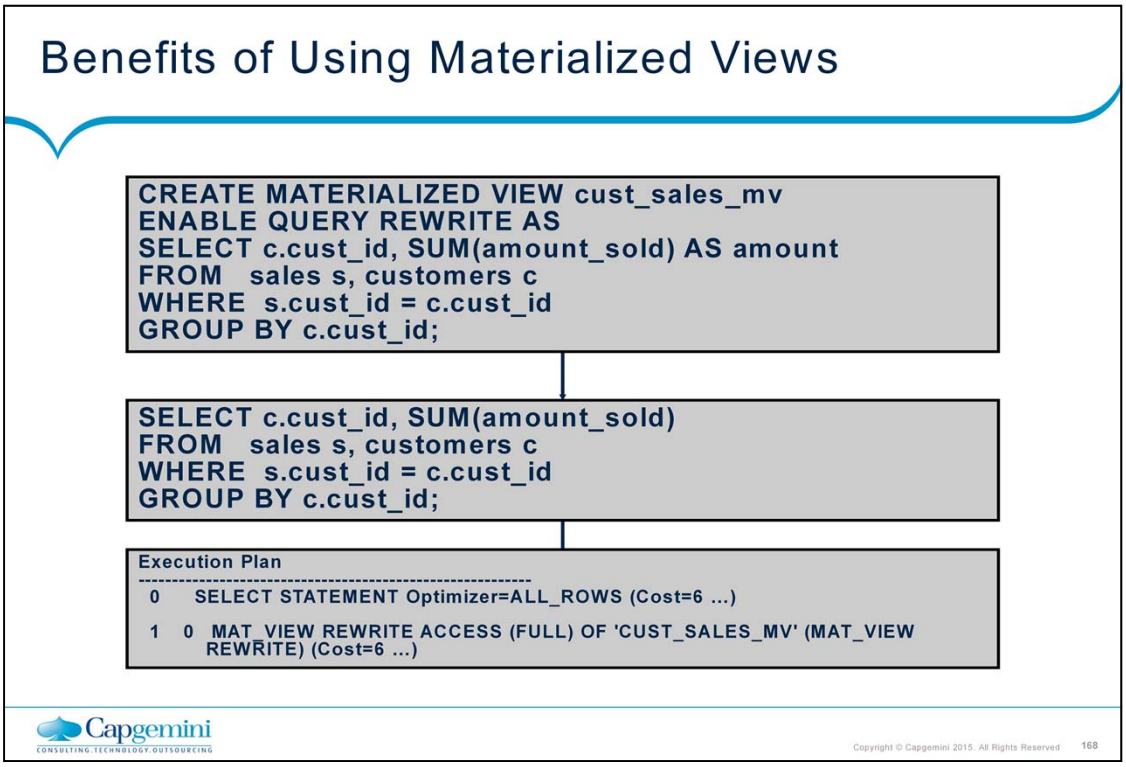
Materialized Views

- A materialized view stores both the definition of a view and the rows resulting from the execution of the view. Like a view, it uses a query as the basis. However, the query is executed at the time the view is created, and the results are stored in a table. You can define the table with the same storage parameters that any other table has, and you can place it in the tablespace of your choice. You can also index the materialized view table in the same way that you index other tables to improve the performance of queries executed against them.
- When a query can be satisfied with data in a materialized view, the Oracle Server transforms the query to reference the view rather than the base tables. By using a materialized view, expensive operations such as joins and aggregations do not need to be reexecuted.
- Note: The query is not always executed when a materialized view is created. It depends on the BUILD clause. If BUILD IMMEDIATE is coded, the data for the query is built when the view is created. If BUILD DEFERRED is coded, the query data is built at first refresh. The ON PREBUILT TABLE clause lets you register an existing table as a preinitialized materialized view. The table must have the same name and be in the same schema as the resulting materialized view. If the materialized view is dropped, then the preexisting table reverts to its identity as a table.



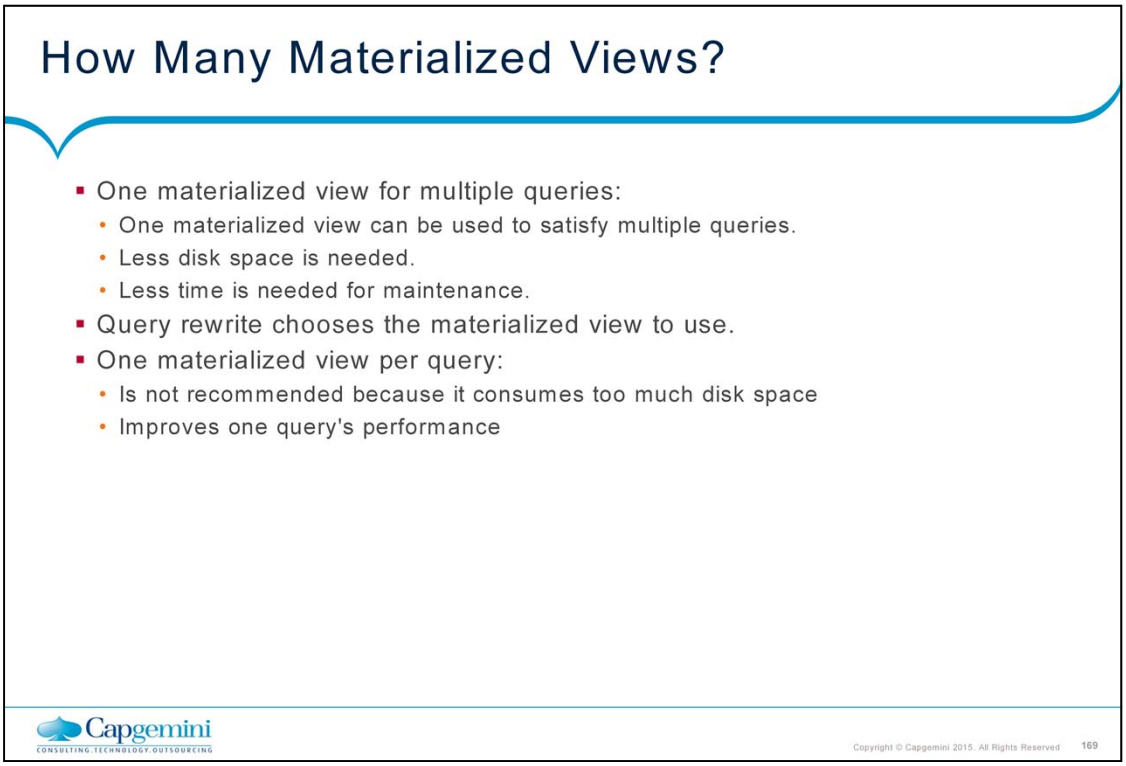
If Materialized Views Are Not Used

- Some organizations that use summaries spend a significant amount of time creating them manually, identifying which ones to create, indexing them, updating them, and advising their users about which ones to use.
- For example, to enhance performance for the SQL query of the application in the slide, you can create a summary table called CUST_SALES_SUM and inform users of its existence. In turn, users use this summary table instead of the original query.
- Obviously, the time required to execute the SQL query from the preceding summary table is minimal compared to the time required for the original SQL query.
- On the other hand, users must be aware of summary tables and need to rewrite their applications to use those tables.
- Also, you must manually refresh the summary tables to keep them up to date with the corresponding original tables.
- It can very quickly become difficult to maintain such a system.



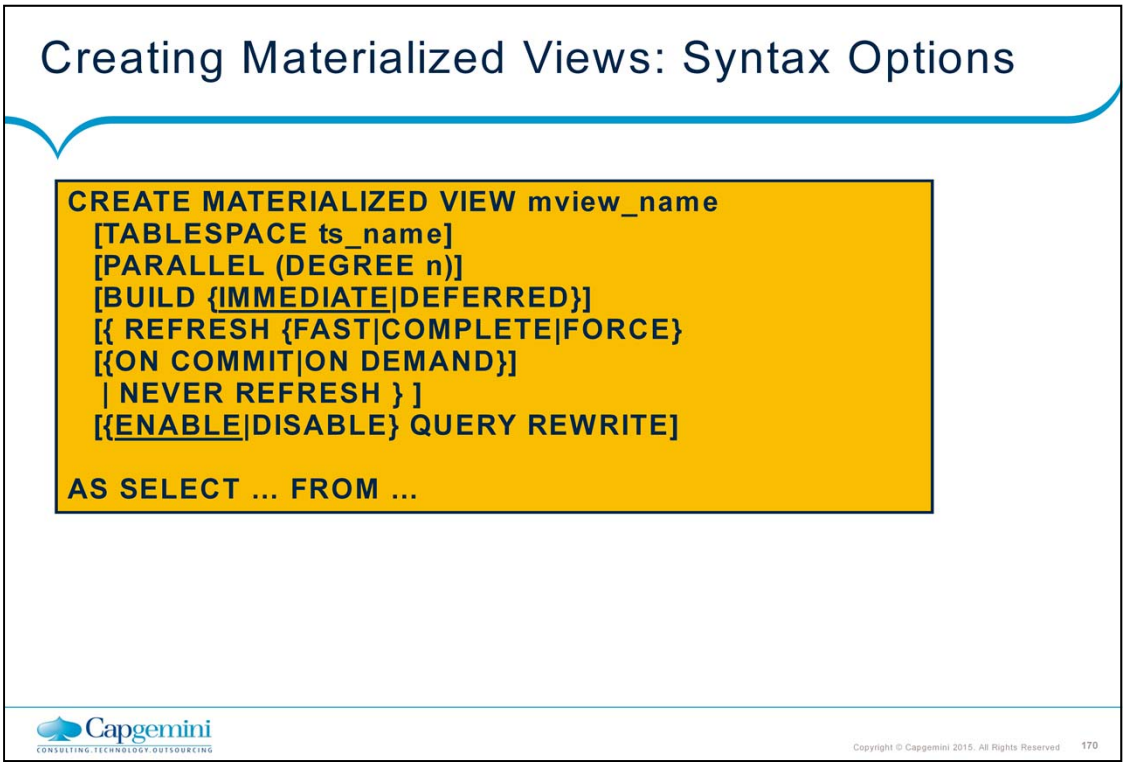
Benefits of Using Materialized Views

- With materialized views, end users no longer must be aware of the summaries that have been defined. You can create one or more materialized views that are the equivalent of summary tables. The advantage provided by creating a materialized view instead of a table is that a materialized view not only materializes the result of a query into a database table, but also generates metadata information used by the query rewrite engine to automatically rewrite the SQL query to use the summary tables. Furthermore, a materialized view optionally offers another important possibility: refreshing data automatically.
- The slide shows that using materialized views is transparent to the user. If your application must execute the same SQL query as in the previous slide, all you need to do is create the materialized view called cust_sales_mv. Then, whenever the application executes the SQL query, the Oracle Server automatically rewrites it to use the materialized view instead.
- Compared to the time that is required by the summary table approach, the query response time is the same. The primary difference is that the application need not be rewritten. The rewrite phase is automatically handled by the system. In addition, the SQL statement that defines the materialized view does not need to match the SQL statement of the query itself.



How Many Materialized Views?

- The query optimizer automatically recognizes when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries go directly to the materialized view and not to the underlying detail tables. In general, rewriting queries to use materialized views rather than detail tables improves queries response time.
- You thus need to decide which materialized view(s) to create. One materialized view can be used by the optimizer to rewrite any number of suitable queries. When using query rewrite, you should therefore create materialized views that satisfy the largest number of queries.
- You can also create nested materialized views. A nested materialized view is a materialized view whose definition is based on another materialized view. A nested materialized view can reference other relations in the database in addition to referencing materialized views. By using nested materialized views, you can create multiple single-table materialized views based on a joins-only materialized view; the join is performed just once. In addition, optimizations can be performed for this class of single-table aggregate materialized view, and thus refresh is very efficient.

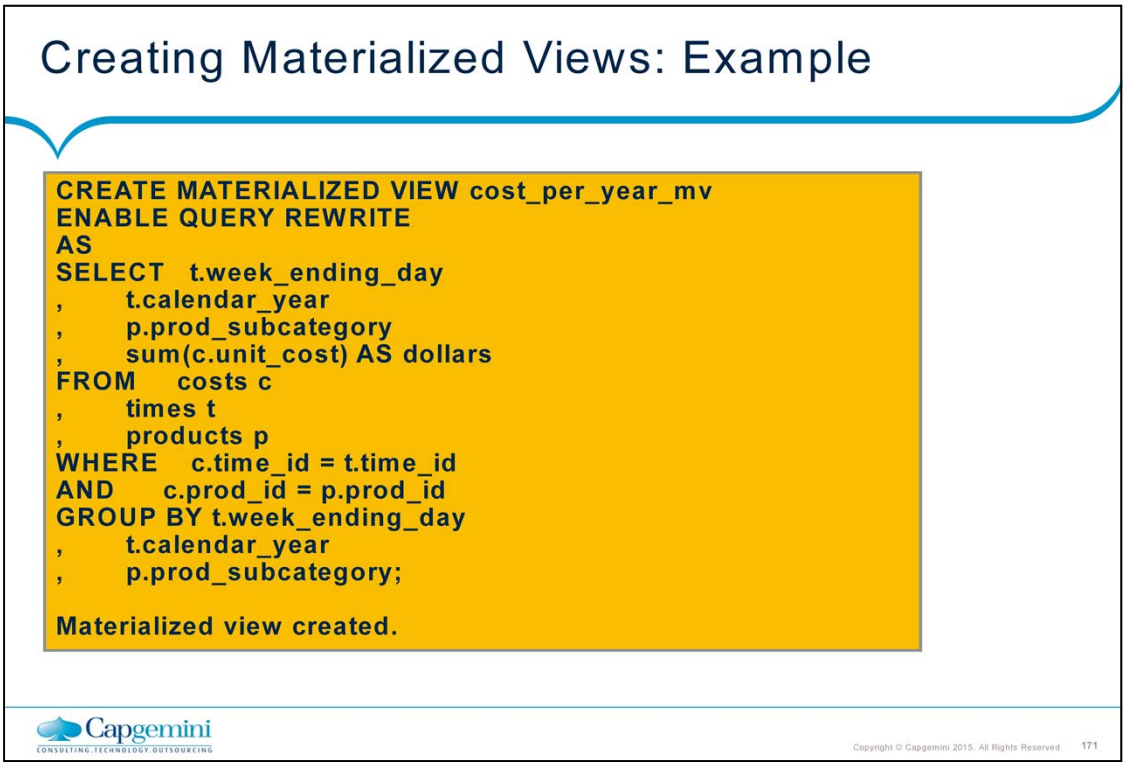


How Many Materialized Views? (continued)

- It is possible to create a materialized view for each incoming SQL query. Although it would be ideal for query response time, this technique is not recommended because you would need considerable disk space to store all the materialized views and because more time is required to create and maintain them.

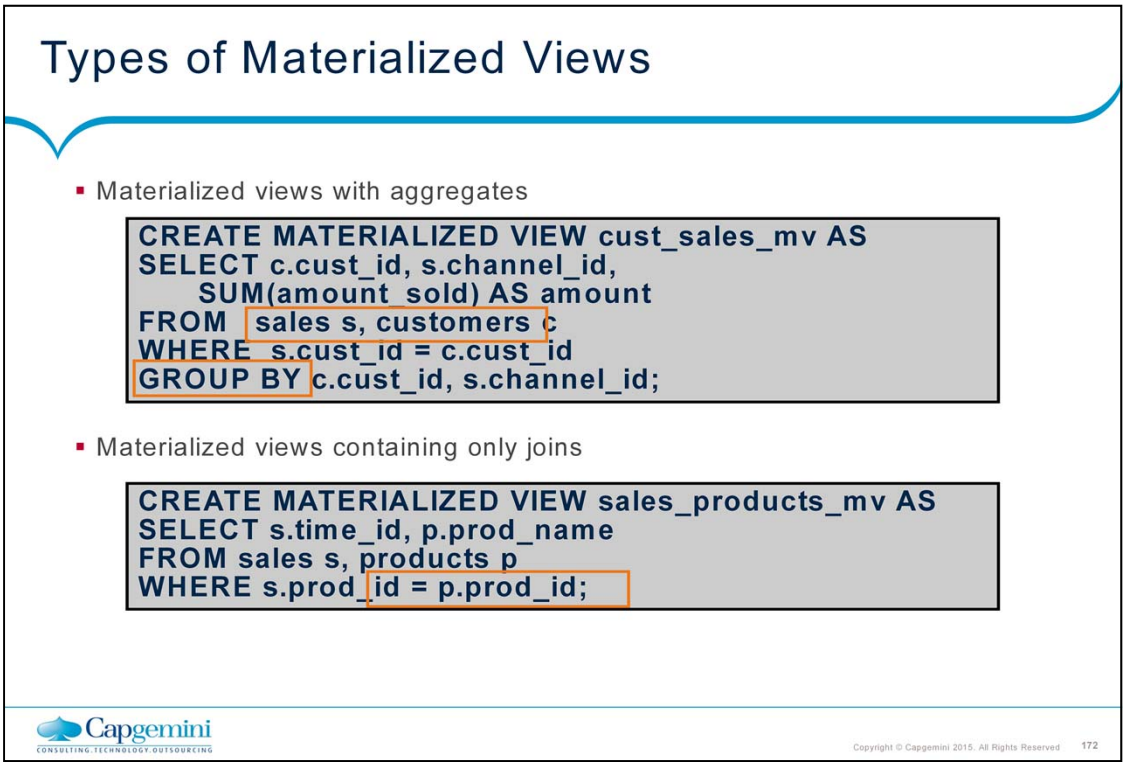
Creating Materialized Views: Syntax Options

- The CREATE MATERIALIZED VIEW syntax is similar to the CREATE SNAPSHOT command, which it replaces. There are some additional options:
  - The BUILD IMMEDIATE option causes the materialized view to be populated when the CREATE command is executed. This is the default behavior. You can choose the BUILD DEFERRED option, which creates the structure but does not populate it until the first refresh occurs.
  - Instead of the BUILD option, you can also specify ON PREBUILT TABLE when you want an existing summary table to be the source of a materialized view.
  - The ENABLE/DISABLE QUERY REWRITE clause determines whether query rewrites are automatically enabled for the materialized view. Query rewrite is enable by default in Oracle Database 10g.



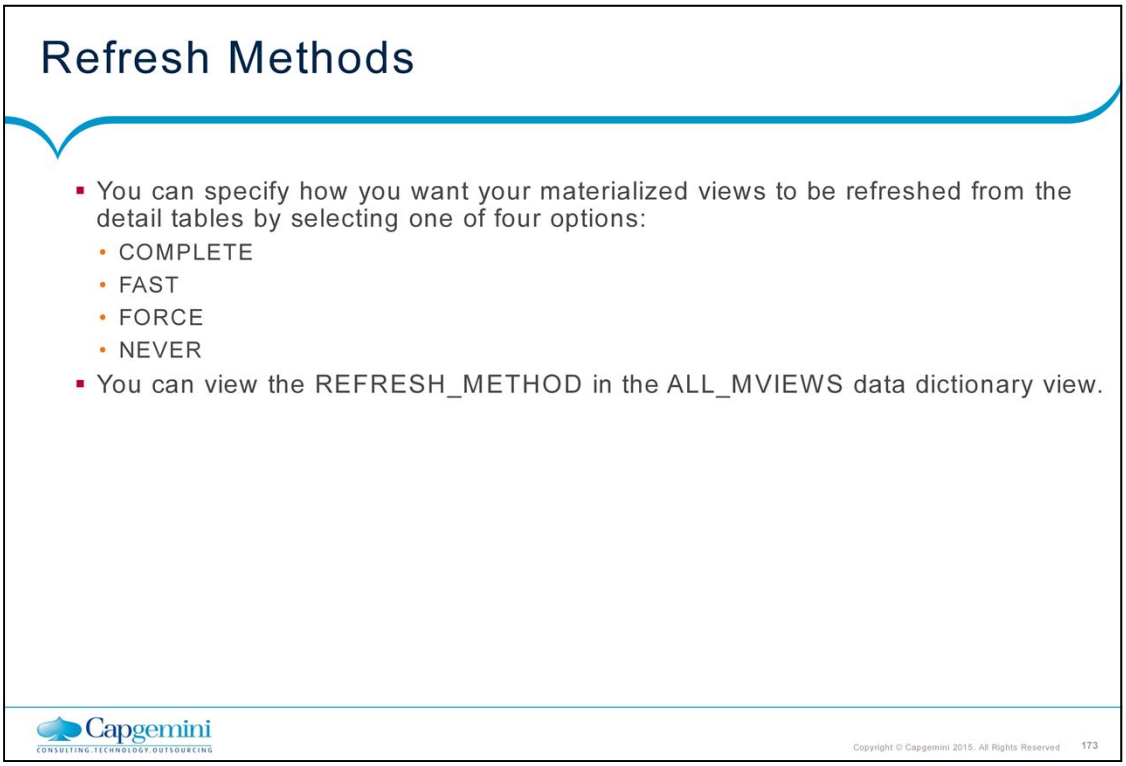
Creating Materialized Views: Example

- If a query involving aggregates, large or multiple joins, or both aggregates and joins is likely to be used multiple times, it can be more efficient to create a materialized view of the query results. This requires a single execution of the query and the storage space to preserve the results.
- If the queries are likely to be reused over time, you may also need a mechanism to update the materialized view as the base tables change. The performance and storage costs of maintaining the materialized view must be compared to the costs of reexecuting the original query whenever it is needed. This is discussed in more detail later in this lesson.
- When the optimizer chooses to use the view rather than the base table, this event is called a query rewrite. This is also discussed in more detail later in this lesson.
- Note: This example is a CREATE MATERIALIZED VIEW command. Details about optional command clauses are discussed later in this lesson.
- Note: In the previous page the default options in the syntax are underlined. This is not the complete syntax; see Oracle Database 10g SQL Reference for more details.



Types of Materialized Views

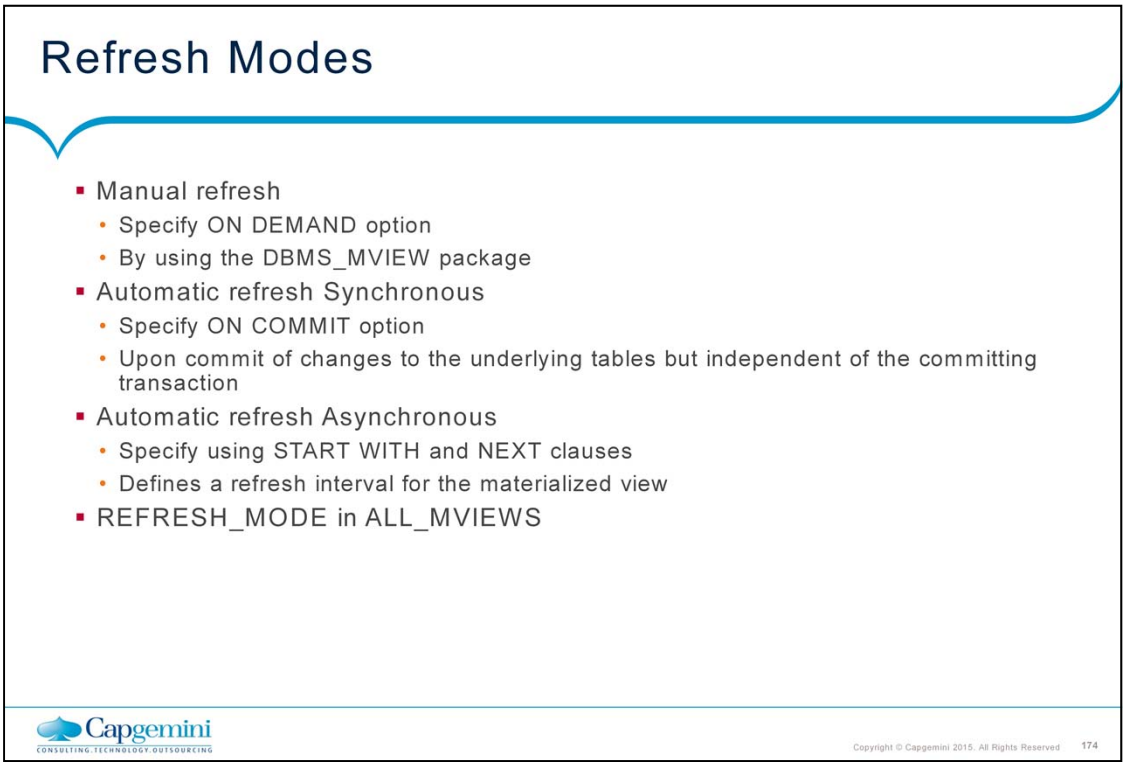
- The SELECT clause in the materialized view creation statement defines the data that the materialized view is to contain. Only a few restrictions limit what can be specified. Any number of tables can be joined together. However, they cannot be remote tables if you want to take advantage of query rewrite. In addition to tables, other elements such as views, inline views (subqueries in the FROM clause of a SELECT statement), subqueries in the WHERE clause, and materialized views can all be joined or referenced in the SELECT clause.
- In data warehouses, materialized views normally contain aggregates. This is the origin of the term summaries in data warehouses. Some materialized views contain only joins and no aggregates. The advantage of creating this type of materialized view is that expensive joins are precalculated.
- Note: The CREATE statements in the slide must include the ENABLE QUERY REWRITE clause. If they do not, then rewrite does not occur, and these materialized views are treated as replication materialized views.



Refresh Methods

- COMPLETE refreshes by recalculating the detail query of the materialized view. This can be accomplished by deleting the table or truncating it. Complete refresh reexecutes the materialized view query, thereby completely recalculating the contents of the materialized view from the detail tables.
- FAST refreshes by incrementally adding the new data that has been inserted or updated in the tables.
- FORCE applies fast refresh if possible; otherwise, it applies COMPLETE refresh.
- NEVER prevents the materialized view from being refreshed with any Oracle refresh mechanism or procedure.

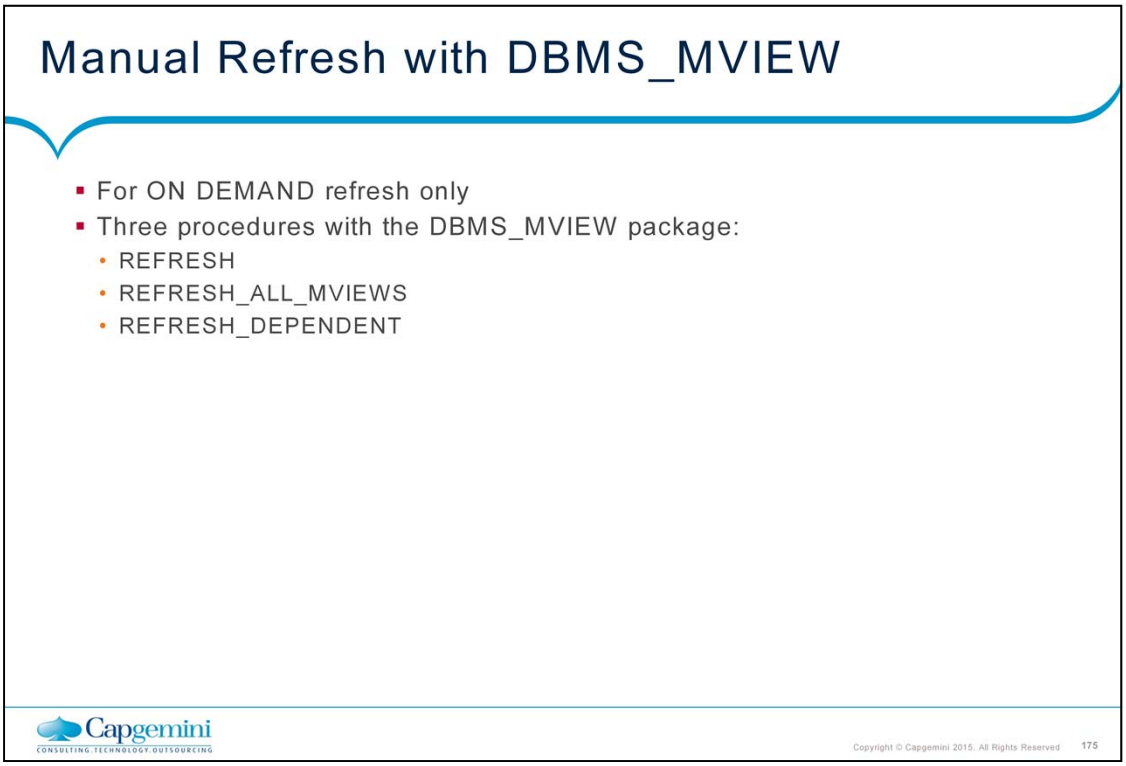
- Some types of nested materialized views cannot be fast refreshed. Use DBMS_MVIEW.EXPLAIN_MVIEW to identify those types of materialized views. You can refresh a tree of nested materialized views in the appropriate dependency order by specifying the nested = TRUE parameter with the DBMS_MVIEW.REFRESH method.



Refresh Modes

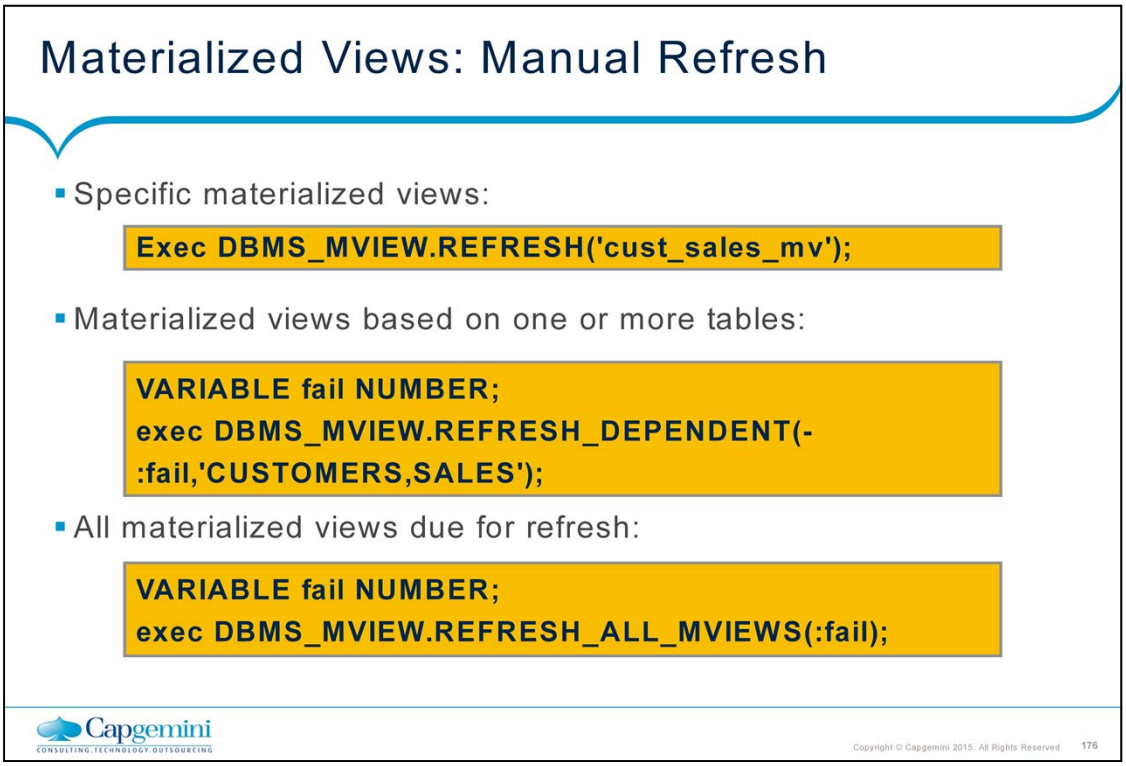
- ON DEMAND (default): Refresh occurs on user demand by using the DBMS_MVIEW package. This package provides several procedures and functions to manage materialized views, including the REFRESH, REFRESH_DEPENDENT, and REFRESH_ALL_MVIEWS procedures.
- ON COMMIT: Refresh occurs automatically when a transaction that modified one of the detail tables of the materialized view commits. This mode can be specified as long as the materialized view is fast refreshable. If a materialized view fails during refresh at commit time, the user must explicitly invoke the refresh procedure using the DBMS_MVIEW package after addressing the errors specified in the trace files. Until this is done, the view is no longer refreshed automatically at commit time.
- At a specified time: Refresh of a materialized view can be scheduled to occur at a specified time (for example, it can be refreshed every Monday at 9:00 a.m. by using the START WITH and NEXT clauses). You can create jobs using DBMS_SCHEDULER to do this.

- Note: If you specify ON COMMIT or ON DEMAND, you cannot also specify START WITH or NEXT.



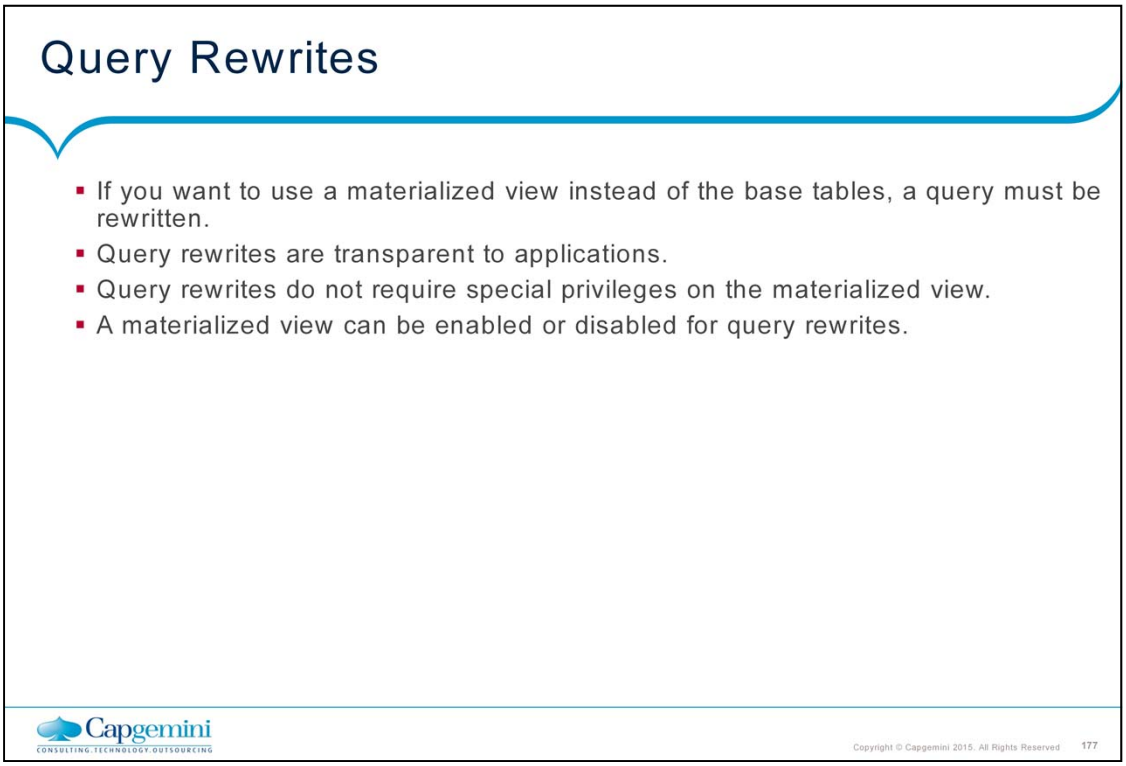
Manual Refresh with DBMS_MVIEW

- You can control the time at which refresh of the materialized views occurs by specifying ON DEMAND refreshes. In this case, the materialized view can be refreshed only by calling one of the procedures in the DBMS_MVIEW package. DBMS_MVIEW provides three different types of refresh operations:
  - DBMS_MVIEW.REFRESH: Refresh one or more materialized views
  - DBMS_MVIEW.REFRESH_ALL_MVIEWS: Refresh all materialized views
  - DBMS_MVIEW.REFRESH_DEPENDENT: Refresh all table-based materialized views that depend on a specified detail table or list of detail tables



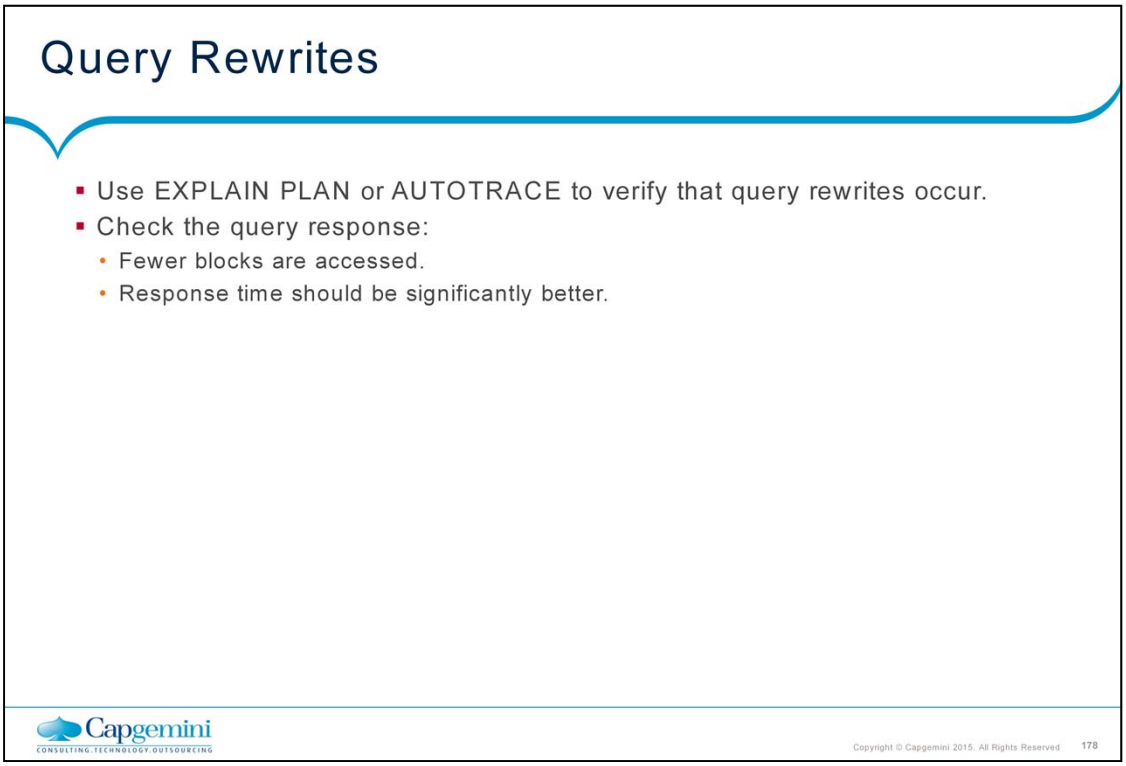
Materialized Views: Manual Refresh

- Possible refresh scenarios for materialized views include the following:
  - Use the REFRESH procedure to refresh specific materialized views.
  - Use the REFRESH_DEPENDENT procedure to refresh all materialized views that depend on a given set of base tables.
  - Use the REFRESH_ALL_MVIEWS procedure to refresh all materialized views that have not been refreshed since the last bulk load to one or more detail tables. This accepts an out parameter that is used to return the number of failures.
- The procedures in the package use several parameters to specify the following:
  - Refresh method
  - Whether to proceed if an error is encountered
  - Whether to use a single transaction (consistent refresh)
  - Rollback segment to use



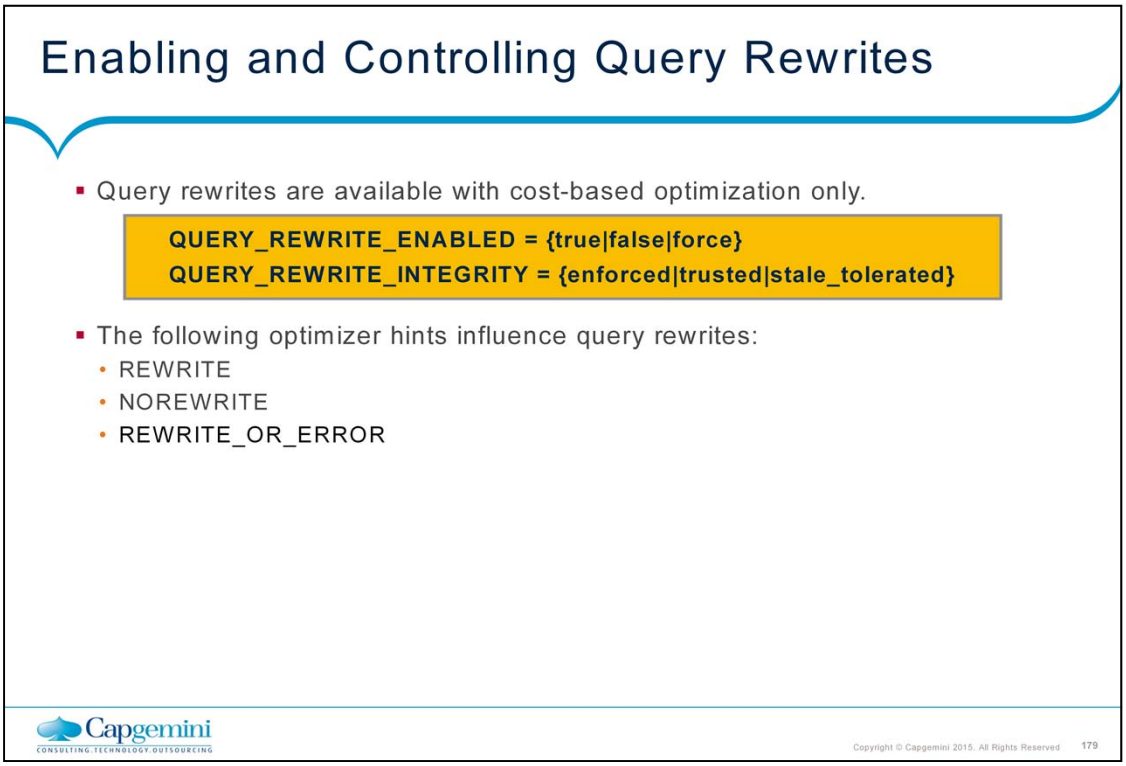
Query Rewrites

- Because accessing a materialized view may be significantly faster than accessing the underlying base tables, the optimizer rewrites a query to access the view when the query allows it. The query rewrite activity is transparent to applications. In this respect, their use is similar to the use of an index.
- Users do not need explicit privileges on materialized views to use them. Queries that are executed by any user with privileges on the underlying tables can be rewritten to access the materialized view.
- A materialized view can be enabled or disabled. A materialized view that is enabled is available for query rewrites, as in the following example:
    - ALTER MATERIALIZED VIEW cust_sales_mv DISABLE QUERY REWRITE;



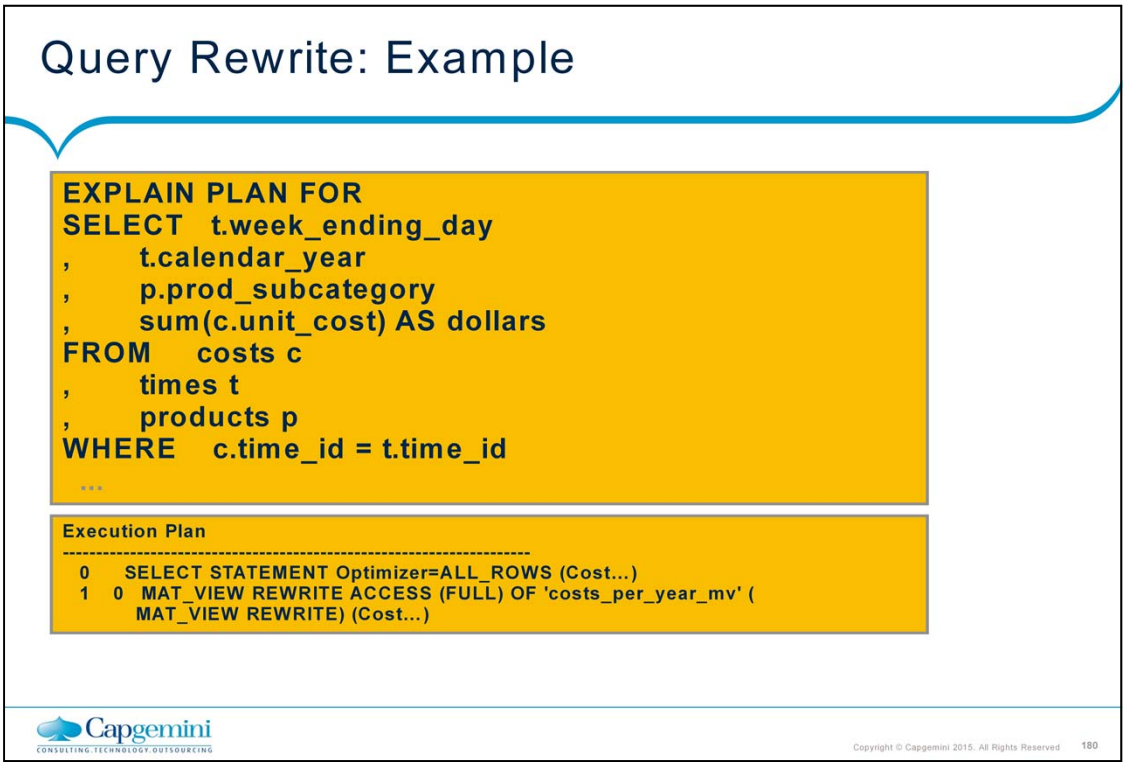
Query Rewrites (continued)

- The best way to determine whether query rewrites occur is to use the EXPLAIN PLAN command or the AUTOTRACE setting in SQL*Plus. The execution plan shows whether a rewrite has taken place that uses the materialized view. You should also notice improved response time if a materialized view is used by the optimizer. You can also use the DBMS_MVIEW.EXPLAIN_REWRITE procedure to verify whether rewrites can occur.
- Note: There are several system privileges that control whether you are allowed to create materialized views and modify them, and whether query rewrites are enabled for you. Contact your local database administrator to set this up properly.
- There are also many data dictionary views that contain information about materialized views. For details, see Oracle Database 10g Performance Guide and Reference and Oracle Database 10g Reference.



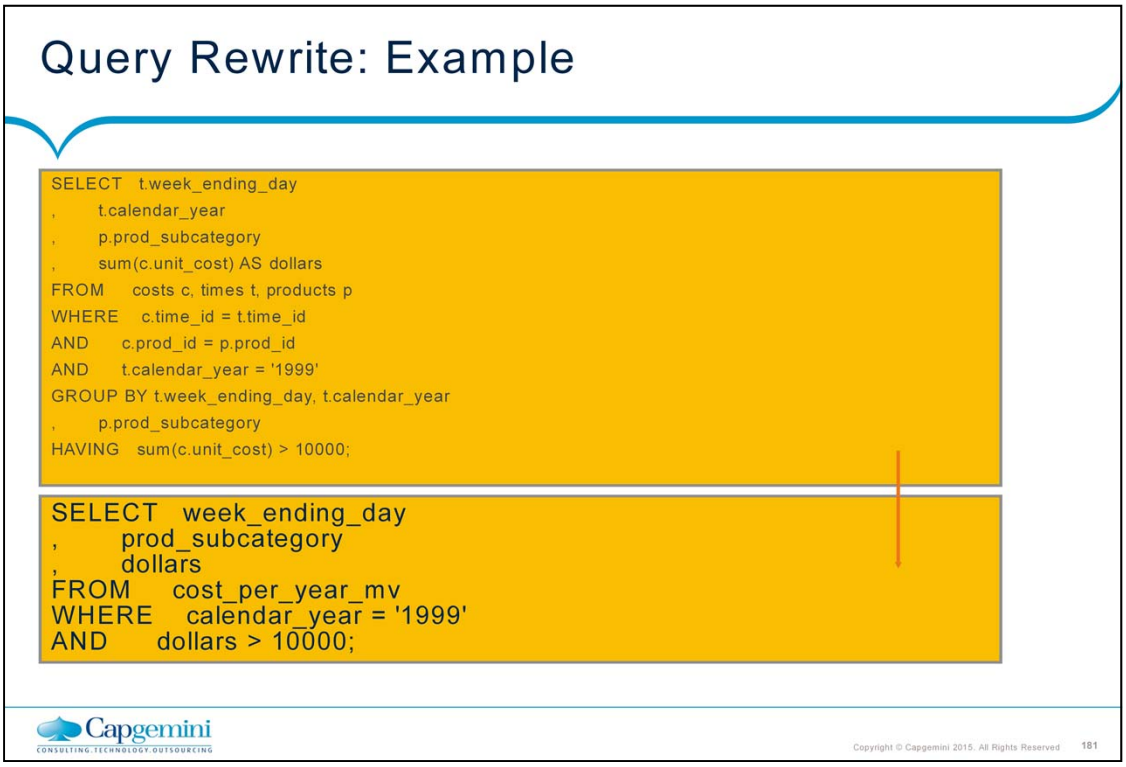
Enabling and Controlling Query Rewrites

- OPTIMIZER_MODE: Query rewrites are available with cost-based optimization only.
- QUERY_REWRITE_ENABLED: This is a dynamic instance/session parameter that can be set to the following values:
  - TRUE: Cost-based rewrite
  - FALSE: No rewrite
  - FORCE: Forced rewrite
- QUERY_REWRITE_INTEGRITY: This is also a dynamic instance/session parameter. It accepts the following values:
  - ENFORCED (default) enables query rewrites only if the optimizer can guarantee consistency. Only fresh materialized views and enabled validated constraints are used for query rewrites.
  - TRUSTED allows query rewrites based on declared (not necessarily enforced) relationships. All fresh materialized views and constraints with the RELY flag are used for query rewrites.
  - STALE_TOLERATED allows stale materialized views that do not contain the latest data to be used.
- Note: If query rewrite does not occur, try STALE_TOLERATED mode first. If the optimizer does not rewrite in this mode, it will never rewrite.



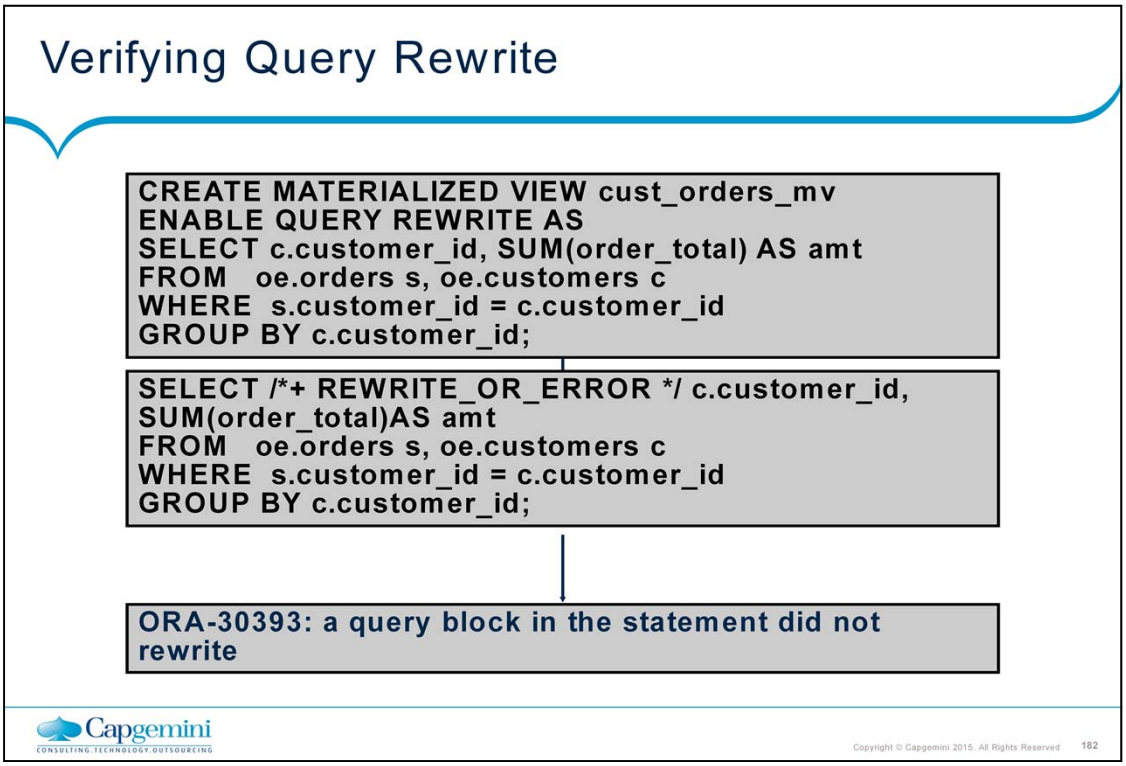
Query Rewrite: Example

- The execution plan shows that the materialized view is accessed instead of joining all four base tables to produce the result.
- A REWRITE or NOREWRITE hint overrides a materialized view's definition that is set in the CREATE or ALTER MATERIALIZED VIEW command with the QUERY REWRITE clause.
- This example shows a transparent query rewrite in which the query exactly matches the materialized view definition. The next slide shows an example of a query that does not match the materialized view definition.



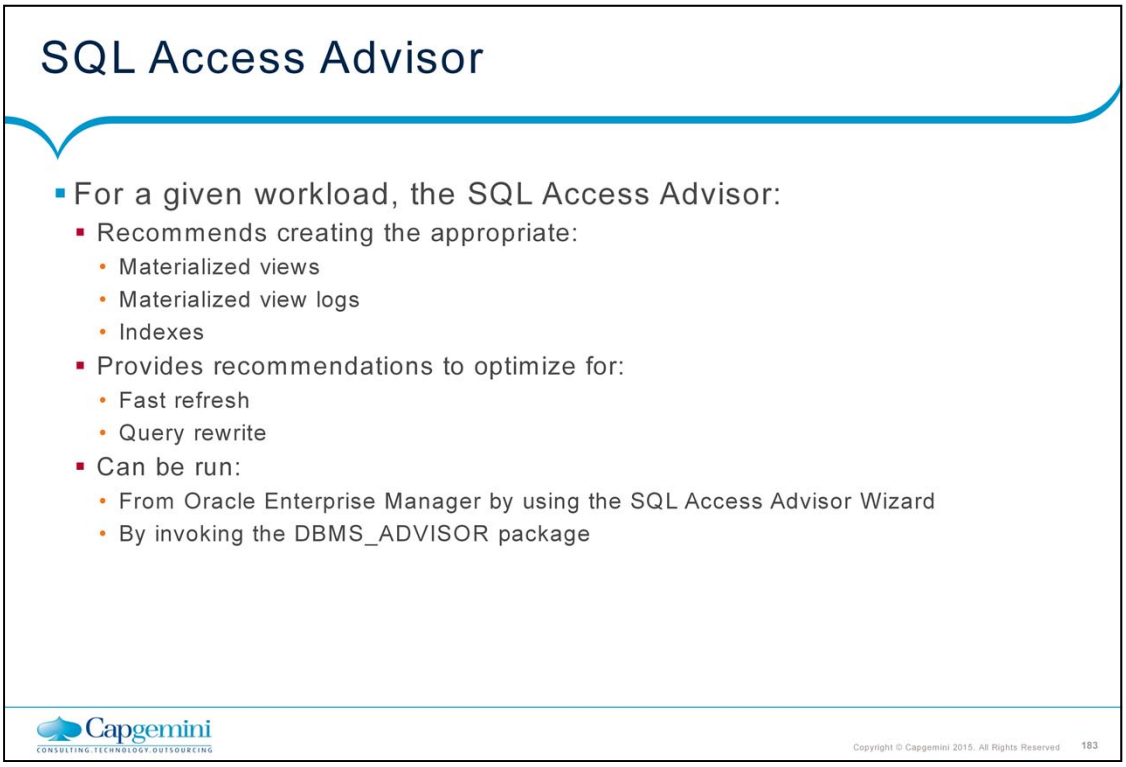
Query Rewrite: Example (continued)

- The optimizer can use the materialized view created earlier to satisfy the query.
- Note: The query in the slide does not exactly match the materialized view definition. You have added a nonjoin predicate on line 8 and a HAVING clause on the last line. The nonjoin predicate is merged into the rewritten query against the materialized view, and the HAVING clause is translated into a second component of the WHERE clause.



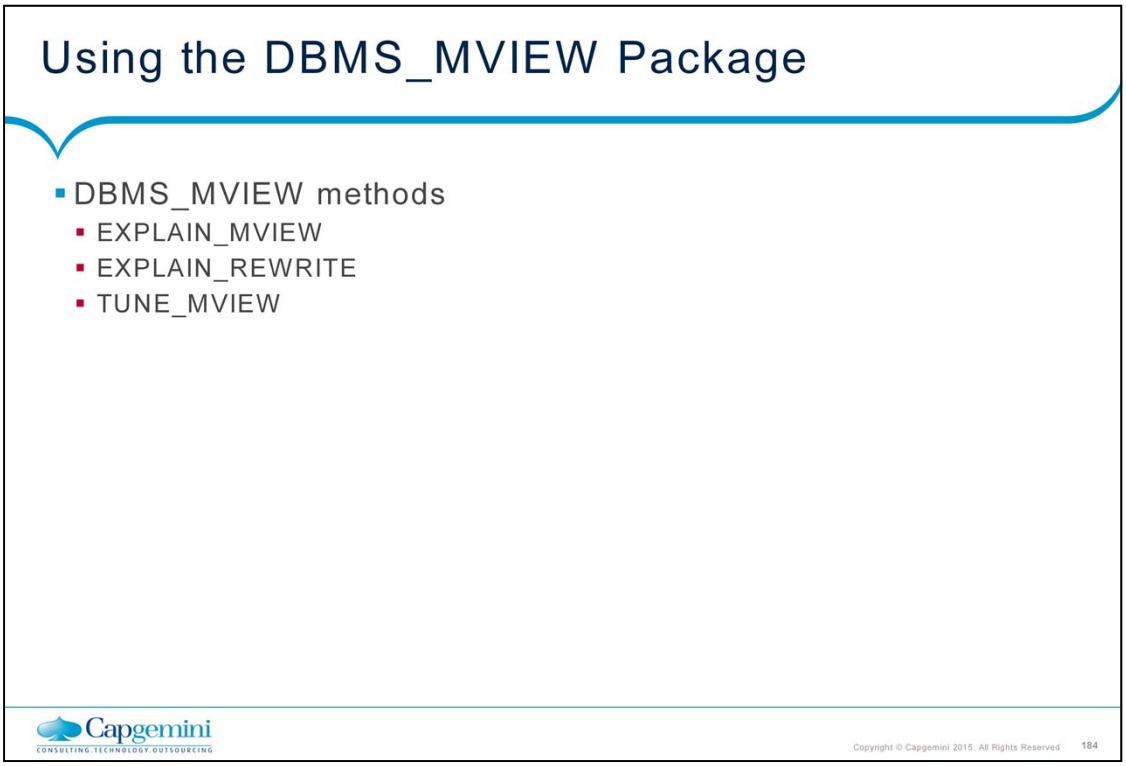
Verifying Query Rewrite

- The example in the slide shows a situation in which creating a materialized view is not useful. Using the REWRITE_OR_ERROR hint in a query causes the following error if the query fails to rewrite:
  - ORA-30393: a query block in the statement did not rewrite
- You can then decide if such a materialized view is worth retaining, based on testing with other queries.



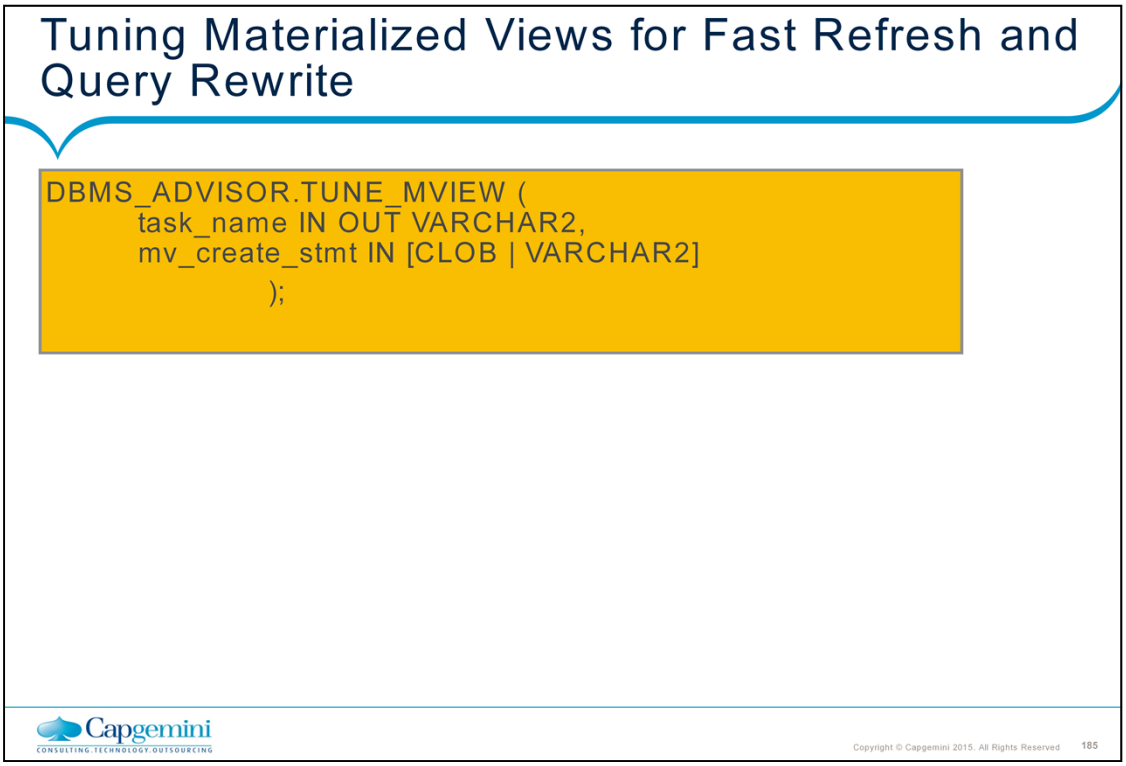
SQL Access Advisor

- The SQL Access Advisor (SAA) helps you achieve your performance goals by recommending the proper set of materialized views, materialized view logs, and indexes for a given workload. Understanding and using these structures is essential when optimizing SQL, because they can result in significant performance improvements in data retrieval. The advantages, however, come with a cost. Creation and maintenance of these objects can be time consuming, and space requirements can be significant. The SAA also recommends how to optimize materialized views so that they can be fast refreshable and take advantage of general query rewrite.
- The SAA can be run from Oracle Enterprise Manager by using the SQL Access Advisor Wizard or by invoking the DBMS_ADVISOR package. The DBMS_ADVISOR package consists of a collection of analysis and advisory functions and procedures that can be called from any PL/SQL program. If a workload is not provided, the SAA can generate and use a hypothetical workload also.



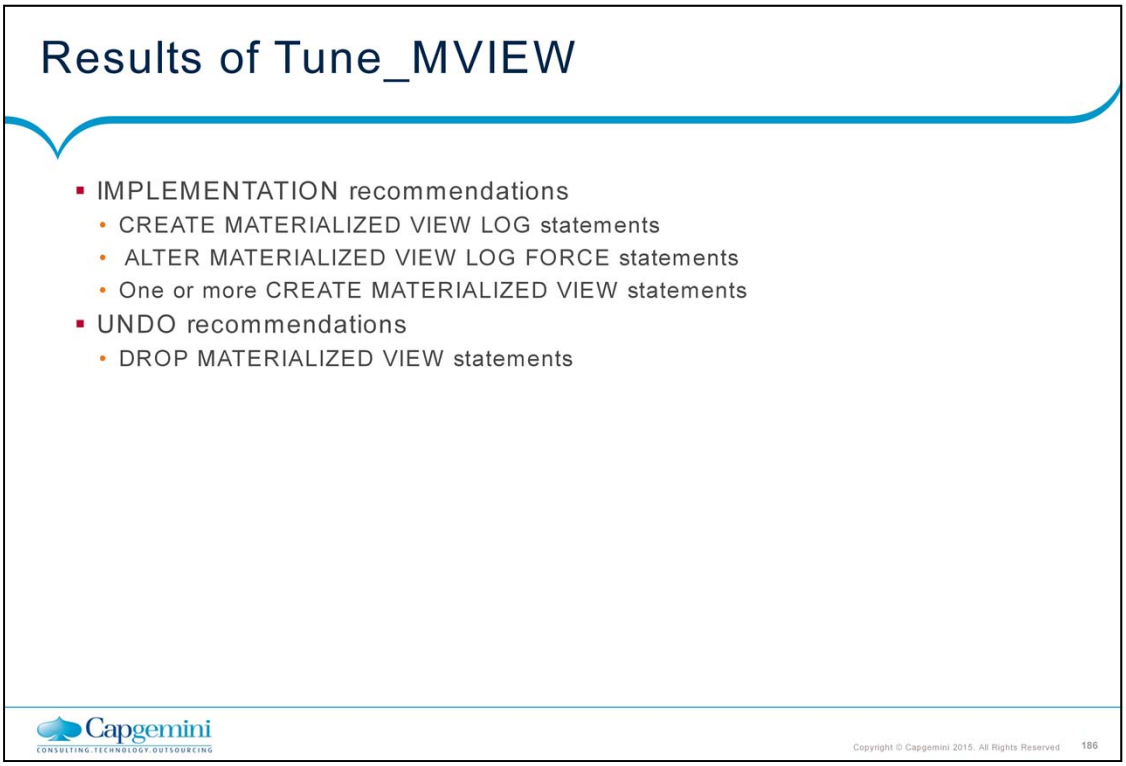
Using the DBMS_MVIEW Package

- Several DBMS_MVIEW procedures help you with materialized view fast refresh and query rewrite.
- The EXPLAIN_MVIEW procedure tells you whether a materialized view is fast refreshable or eligible for general query rewrite. EXPLAIN_REWRITE tells you whether query rewrite will occur. However, neither tells you how to achieve fast refresh or query rewrite.
- To further facilitate the use of materialized views, the TUNE_MVIEW procedure shows how to optimize your CREATE MATERIALIZED VIEW statement and how to meet other requirements (such as materialized view log for fast refresh and general query rewrite).



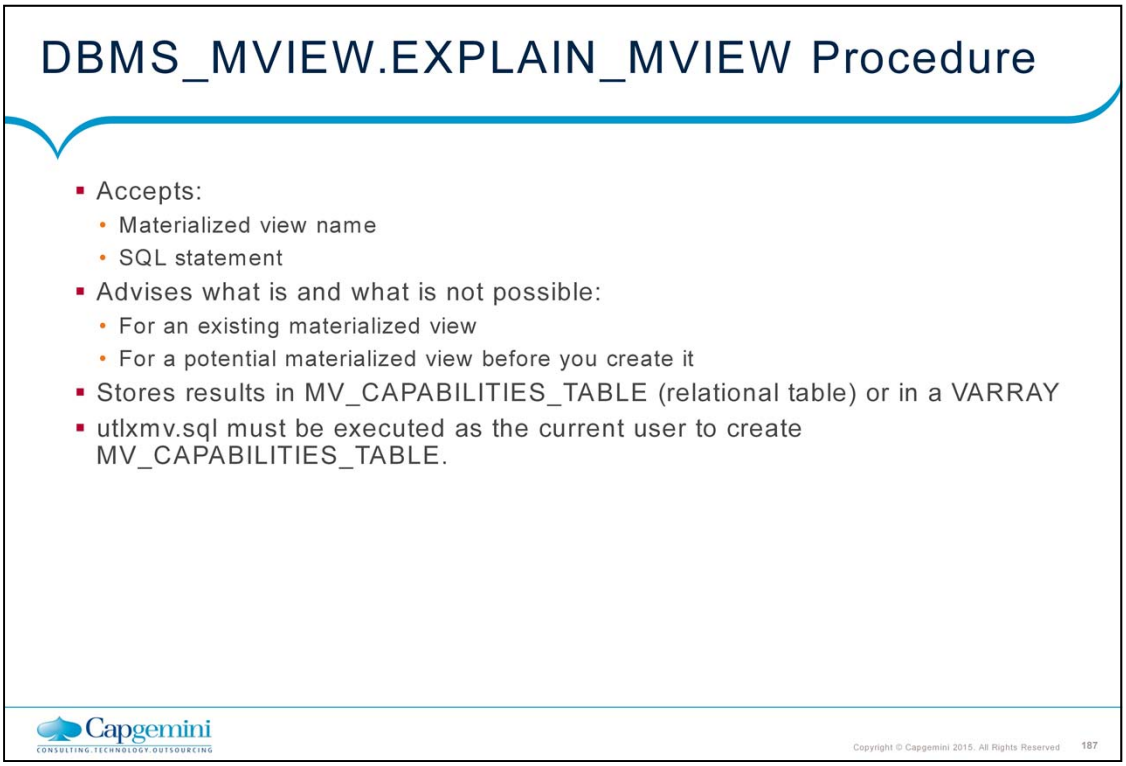
Tuning Materialized Views for Fast Refresh and Query Rewrite

- The DBMS_MVIEW.TUNE_MVIEW method analyzes and processes the input statement and generates two sets of output results: one for the materialized view implementation and the other for undoing the create materialized view operations. The two sets of output results can be accessed through Oracle views or can be stored in external script files created by the SAA. These external script files are ready to execute to implement the materialized view.
- The TUNE_MVIEW procedure takes two input parameters: task_name and mv_create_stmt. Task_name is a user-provided task identifier used to access the output results. Mv_create_stmt is a complete CREATE MATERIALIZED VIEW statement that is to be tuned. If the input CREATE MATERIALIZED VIEW statement does not have the clauses REFRESH FAST or ENABLE QUERY REWRITE, or both, TUNE_MVIEW will use the default clauses REFRESH FORCE and DISABLE QUERY REWRITE to tune the statement to be fast refreshable (if possible) or only complete refreshable.



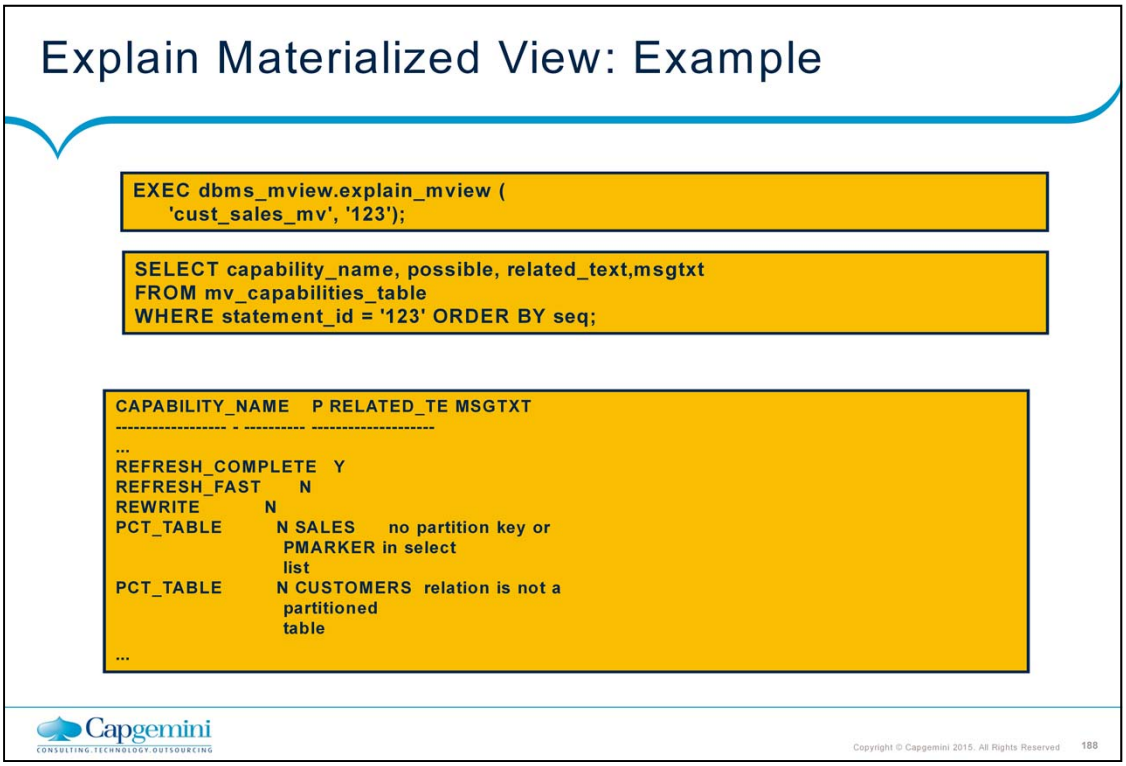
Results of Tune_MVIEW

- When the ENABLE QUERY REWRITE clause is specified, TUNE_MVIEW also fixes the statement by using a process similar to REFRESH FAST, which redefines the materialized view so that many of the advanced forms of query rewrite are possible.
- The TUNE_MVIEW procedure generates two sets of output results as executable statements. One set of output results, IMPLEMENTATION, is for implementing materialized views and required components such as materialized view logs to achieve fast refreshability and query rewritability. The other set of output results, UNDO, is for dropping the materialized views (if you decide that they are not required).
- The output statements for the IMPLEMENTATION process include:
  - CREATE MATERIALIZED VIEW LOG statements, which create any missing materialized view logs required for fast refresh
  - ALTER MATERIALIZED VIEW LOG FORCE statements, which fix any materialized view log–related requirements (such as missing filter columns, sequence, and so on) that are required for fast refresh



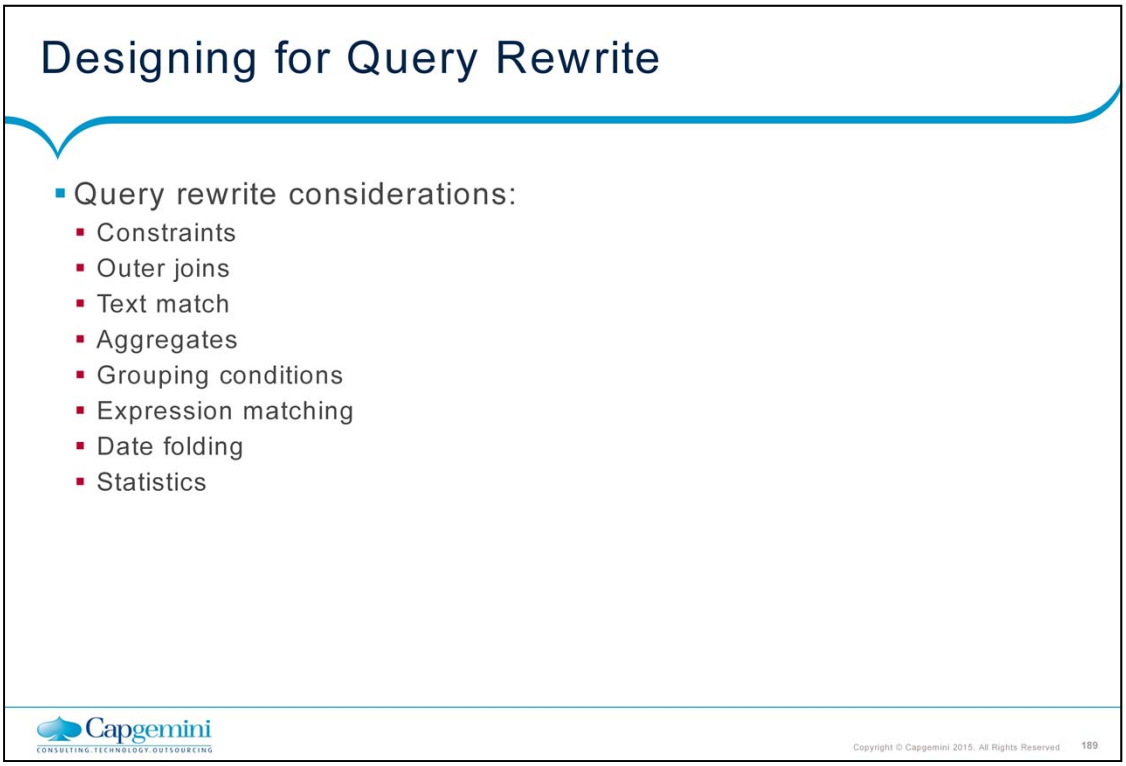
DBMS_MVIEW.EXPLAIN_MVIEW Procedure

- The purpose of the Explain Materialized View procedure is to advise what is and is not possible with a given materialized view or potential materialized view. This package advises the user by providing answers to the following questions:
  - Is this materialized view fast refreshable?
  - What are the types of query rewrite that can be done with this materialized view?
- The process for using this package is very simple. The procedure DBMS_MVIEW.EXPLAIN_MVIEW is called, passing in as parameters the schema and materialized view name for an existing materialized view. Alternatively, you can specify the select string for a potential materialized view. The materialized view or potential materialized view is then analyzed, and the results are written either to a table called MV_CAPABILITIES_TABLE (the default) or to a VARRAY of type ExplainMVArrayType called MSG_ARRAY.
- Note: Except when using VARRAYS, you must run the utlxmv.sql script prior to calling EXPLAIN_MVIEW. The script creates the MV_CAPABILITIES_TABLE in the current schema. The script is found in the admin directory.



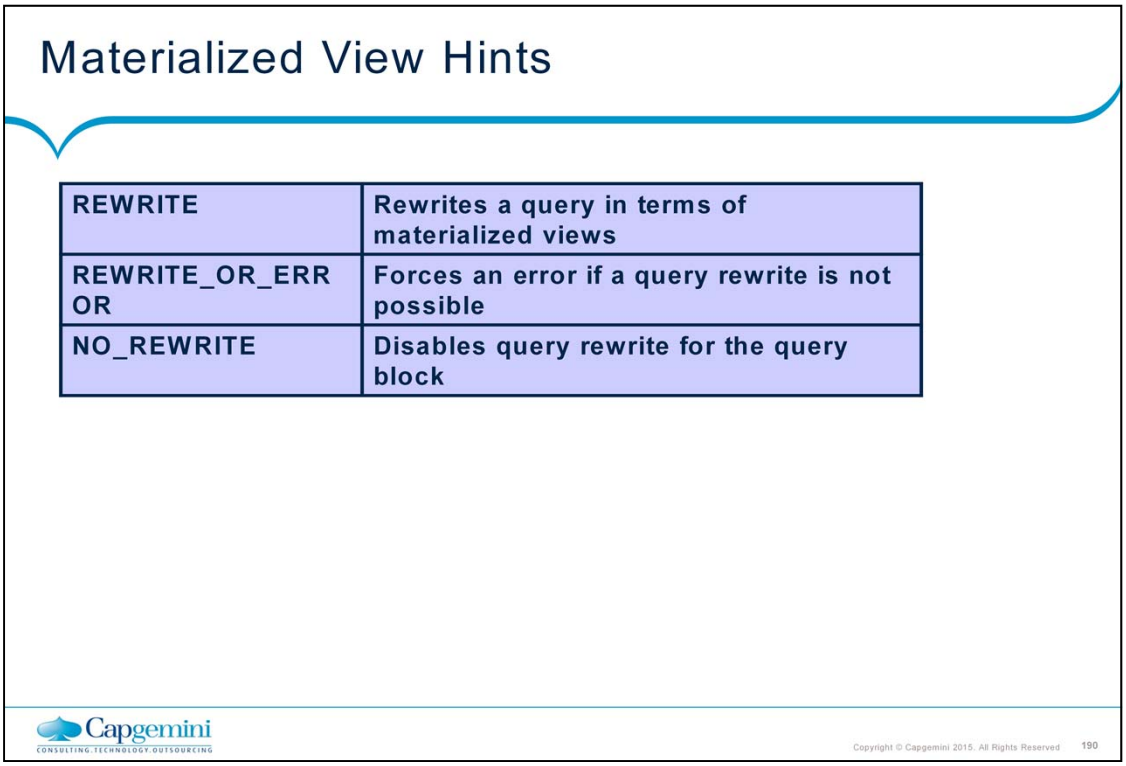
Explain Materialized View: Example

- In the example in the slide, suppose that the MV_CAPABILITIES_TABLE has already been created. You want to analyze an already-existing materialized view called CUST_SALES_MV that was created in the SH schema. You need to assign an ID for this analysis so that we can retrieve it subsequently in the MV_CAPABILITIES_TABLE. You also need to use the SEQ column in an ORDER BY clause so that the rows appear in a logical order.
- If a capability is not possible, N appears in the P column and an explanation appears in the MSGTXT column. If a capability is not possible for more than one reason, a row is displayed for each reason.



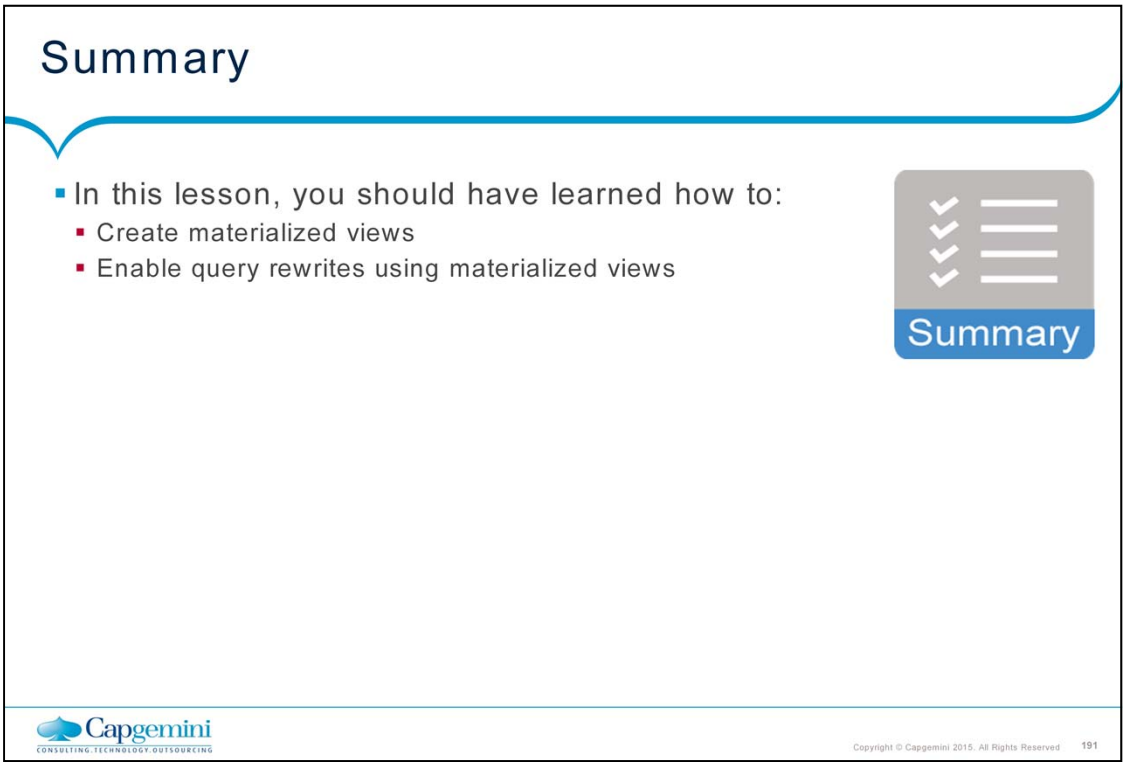
Designing for Query Rewrite

- Constraints: Make sure that all inner joins that are referred to in a materialized view have referential integrity (foreign key and primary key constraints) with additional NOT NULL constraints on the foreign key columns. Because constraints tend to impose a large overhead, you could make them NO VALIDATE and RELY and set the parameter QUERY_REWRITE_INTEGRITY to STALE_TOLERATED or TRUSTED. However, if you set QUERY_REWRITE_INTEGRITY to ENFORCED, all constraints must be enabled, enforced, and validated to obtain maximum rewritability. You should avoid using the ON DELETE clause because it can lead to unexpected results.
- Outer joins: Another way of avoiding constraints is to use outer joins in the materialized view. Query rewrite will be able to derive an inner join in the query, such as (A.a = B.b), from an outer join in the materialized view (A.a = B.b(+)) as long as the row ID of B or column B.b is available in the materialized view. Most of the support for rewrites with outer joins is provided for materialized views with joins only. To exploit it, a materialized view with outer joins should store the row ID or primary key of the inner table of an outer join.



Materialized View Hints

- REWRITE, NOREWRITE, and REWRITE_OR_ERROR are hints that are used with materialized views to control query rewrites.
- REWRITE
- The REWRITE hint forces the cost-based optimizer to rewrite a query in terms of materialized views (when possible) without cost consideration. Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, then the Oracle Server uses that view regardless of its cost.
- The server does not consider views outside of the list. If you do not specify a view list, then the server searches for an eligible materialized view and always uses it regardless of its cost.
- REWRITE_OR_ERROR
- This hint forces an error if a query rewrite is not possible.
- NOREWRITE
- The NOREWRITE hint disables query rewrite for the query block, overriding the setting of the QUERY_REWRITE_ENABLED parameter. Use the NOREWRITE hint on any query block of a request.



Summary

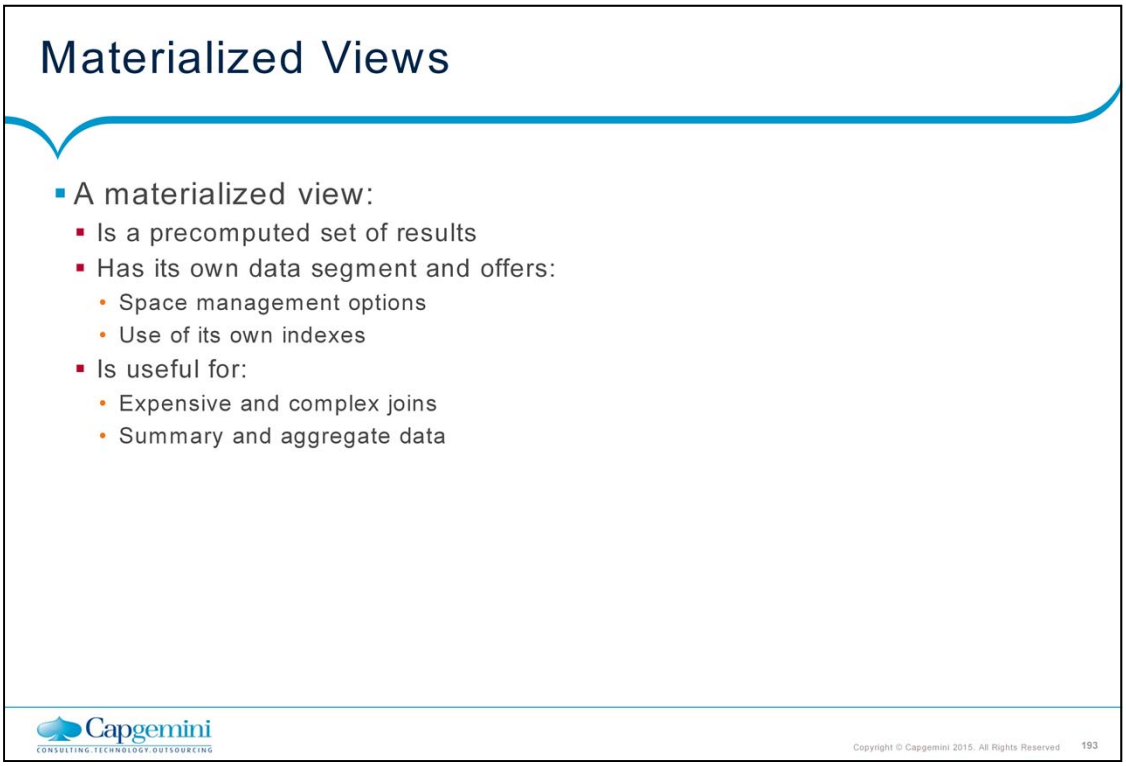
- This lesson introduced you to materialized views and query rewrites. A materialized view stores both the definition of a view and the rows resulting from the execution of the view. Like a view, it uses a query as its basis; however, the query is executed at the time the view is created and the results are stored in a table.
- Because accessing a materialized view can be significantly faster than accessing the underlying base tables, the optimizer rewrites a query to access the view when the query allows it. The query rewrite activity is transparent to applications.

Objectives

- After completing this lesson, you should be able to do the following:
  - Identify the characteristics and benefits of materialized views
  - Use materialized views to enable query rewrites
  - Verify the properties of materialized views
  - Perform refreshes on materialized views

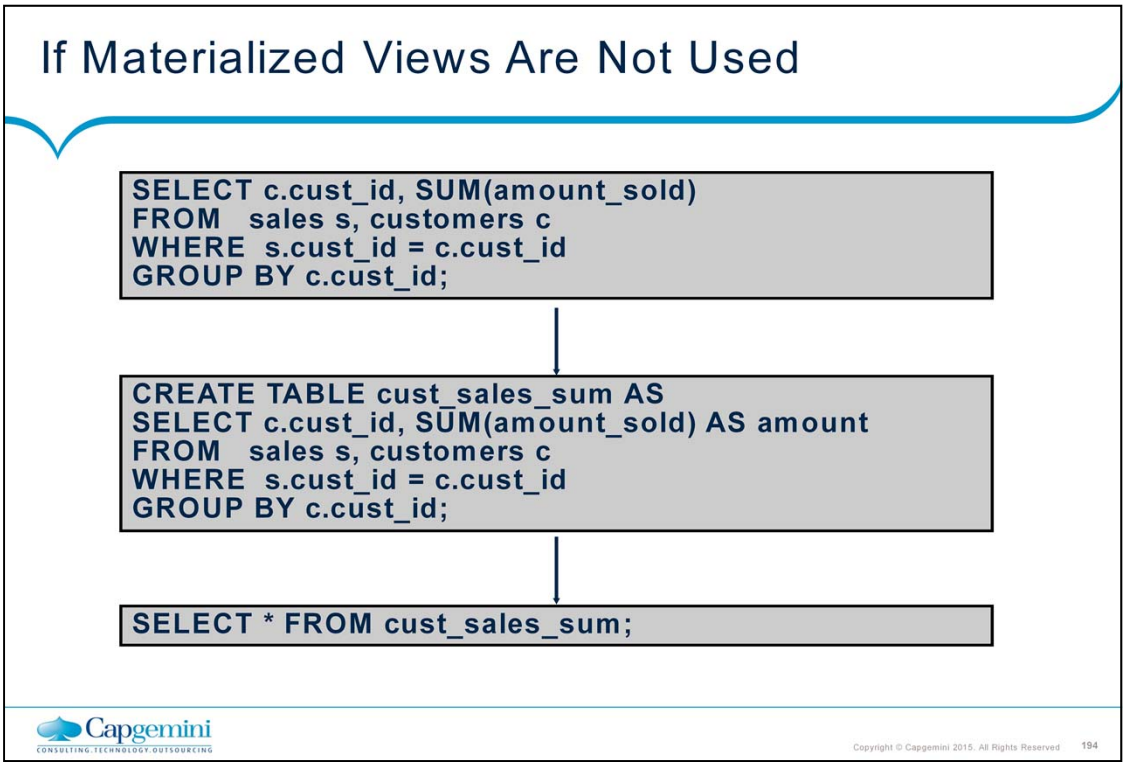
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 192



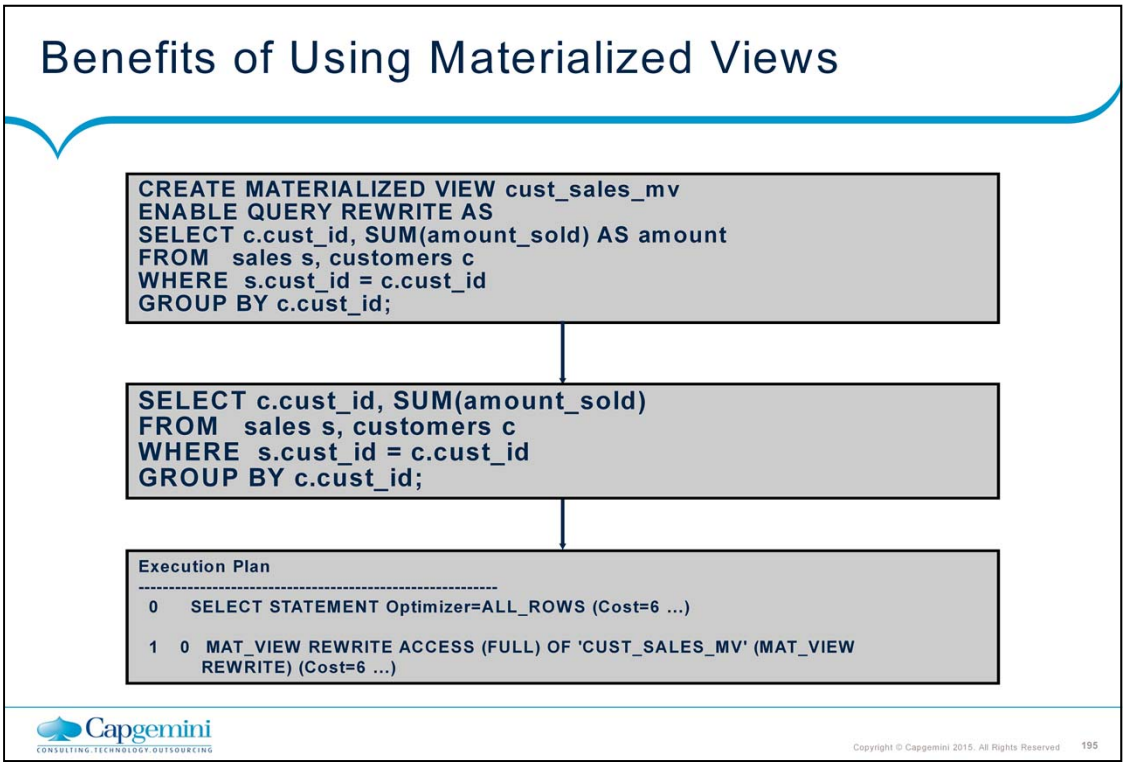
Materialized Views

- A materialized view stores both the definition of a view and the rows resulting from the execution of the view. Like a view, it uses a query as the basis. However, the query is executed at the time the view is created, and the results are stored in a table. You can define the table with the same storage parameters that any other table has, and you can place it in the tablespace of your choice. You can also index the materialized view table in the same way that you index other tables to improve the performance of queries executed against them.
- When a query can be satisfied with data in a materialized view, the Oracle Server transforms the query to reference the view rather than the base tables. By using a materialized view, expensive operations such as joins and aggregations do not need to be reexecuted.
- Note: The query is not always executed when a materialized view is created. It depends on the BUILD clause. If BUILD IMMEDIATE is coded, the data for the query is built when the view is created. If BUILD DEFERRED is coded, the query data is built at first refresh. The ON PREBUILT TABLE clause lets you register an existing table as a preinitialized materialized view. The table must have the same name and be in the same schema as the resulting materialized view. If the materialized view is dropped, then the preexisting table reverts to its identity as a table.



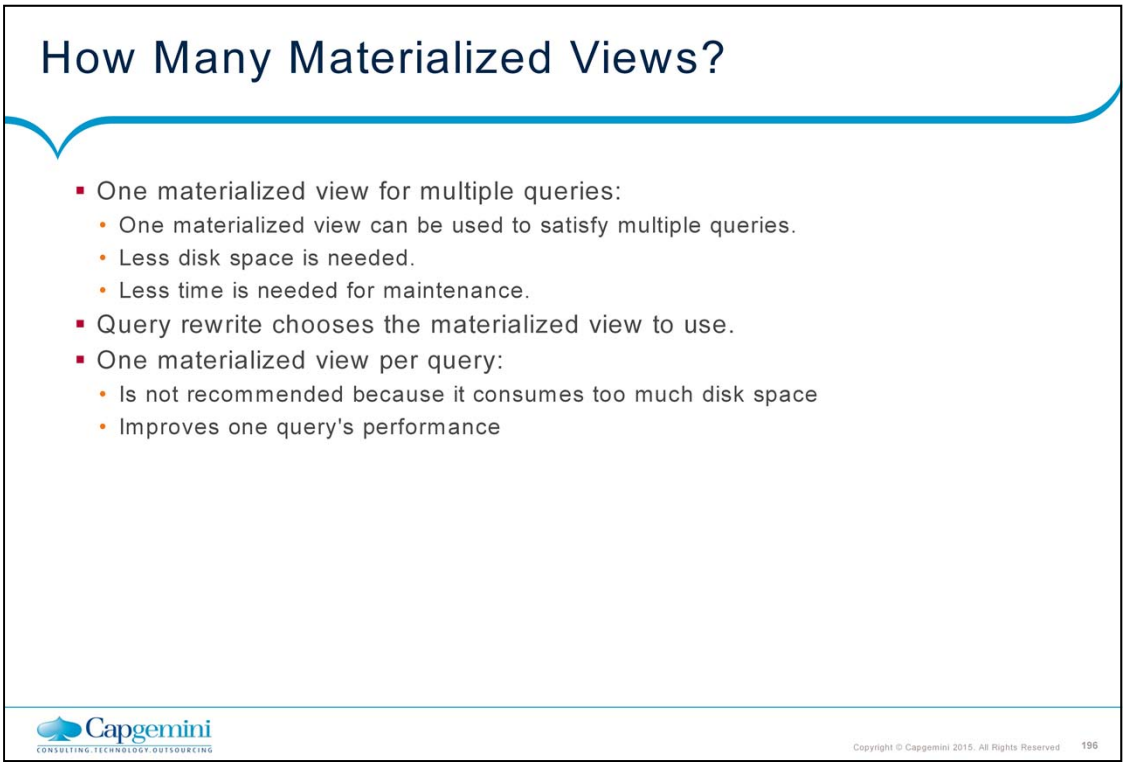
If Materialized Views Are Not Used

- Some organizations that use summaries spend a significant amount of time creating them manually, identifying which ones to create, indexing them, updating them, and advising their users about which ones to use.
- For example, to enhance performance for the SQL query of the application in the slide, you can create a summary table called CUST_SALES_SUM and inform users of its existence. In turn, users use this summary table instead of the original query.
- Obviously, the time required to execute the SQL query from the preceding summary table is minimal compared to the time required for the original SQL query.
- On the other hand, users must be aware of summary tables and need to rewrite their applications to use those tables.
- Also, you must manually refresh the summary tables to keep them up to date with the corresponding original tables.
- It can very quickly become difficult to maintain such a system.



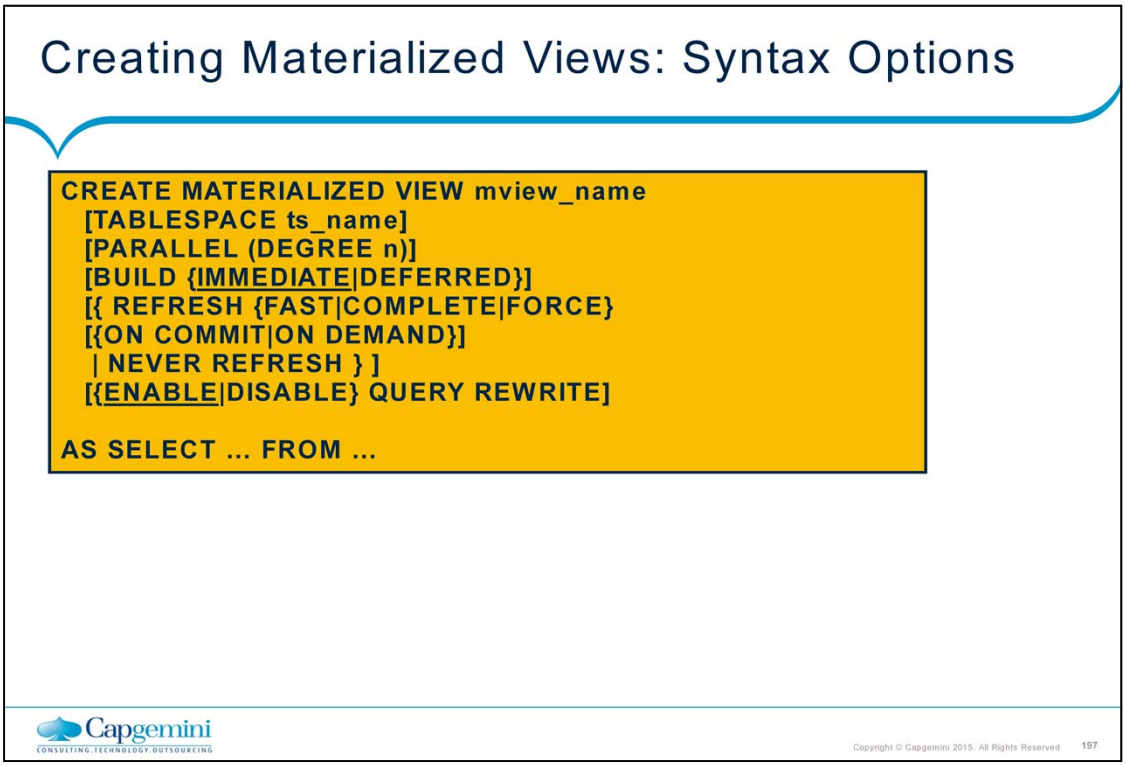
Benefits of Using Materialized Views

- With materialized views, end users no longer must be aware of the summaries that have been defined. You can create one or more materialized views that are the equivalent of summary tables. The advantage provided by creating a materialized view instead of a table is that a materialized view not only materializes the result of a query into a database table, but also generates metadata information used by the query rewrite engine to automatically rewrite the SQL query to use the summary tables. Furthermore, a materialized view optionally offers another important possibility: refreshing data automatically.
- The slide shows that using materialized views is transparent to the user. If your application must execute the same SQL query as in the previous slide, all you need to do is create the materialized view called cust_sales_mv. Then, whenever the application executes the SQL query, the Oracle Server automatically rewrites it to use the materialized view instead.
- Compared to the time that is required by the summary table approach, the query response time is the same. The primary difference is that the application need not be rewritten. The rewrite phase is automatically handled by the system. In addition, the SQL statement that defines the materialized view does not need to match the SQL statement of the query itself.



How Many Materialized Views?

- The query optimizer automatically recognizes when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries go directly to the materialized view and not to the underlying detail tables. In general, rewriting queries to use materialized views rather than detail tables improves queries response time.
- You thus need to decide which materialized view(s) to create. One materialized view can be used by the optimizer to rewrite any number of suitable queries. When using query rewrite, you should therefore create materialized views that satisfy the largest number of queries.
- You can also create nested materialized views. A nested materialized view is a materialized view whose definition is based on another materialized view. A nested materialized view can reference other relations in the database in addition to referencing materialized views. By using nested materialized views, you can create multiple single-table materialized views based on a joins-only materialized view; the join is performed just once. In addition, optimizations can be performed for this class of single-table aggregate materialized view, and thus refresh is very efficient.

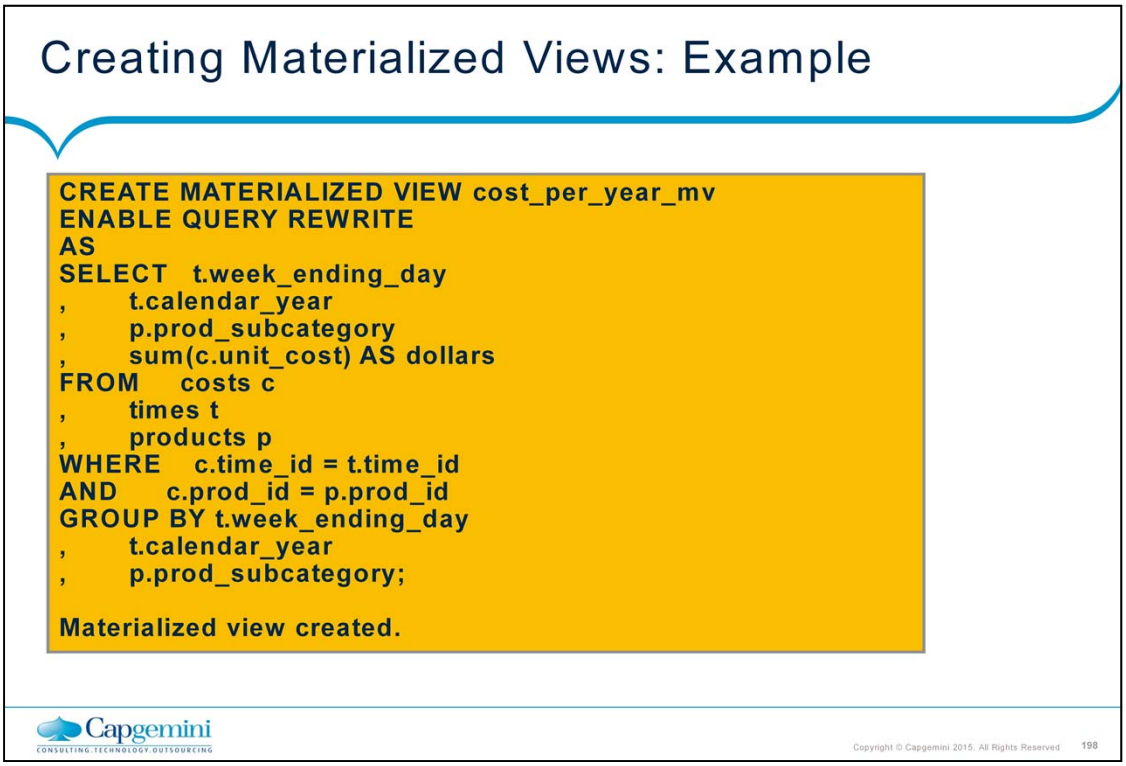


How Many Materialized Views? (continued)

- It is possible to create a materialized view for each incoming SQL query. Although it would be ideal for query response time, this technique is not recommended because you would need considerable disk space to store all the materialized views and because more time is required to create and maintain them.

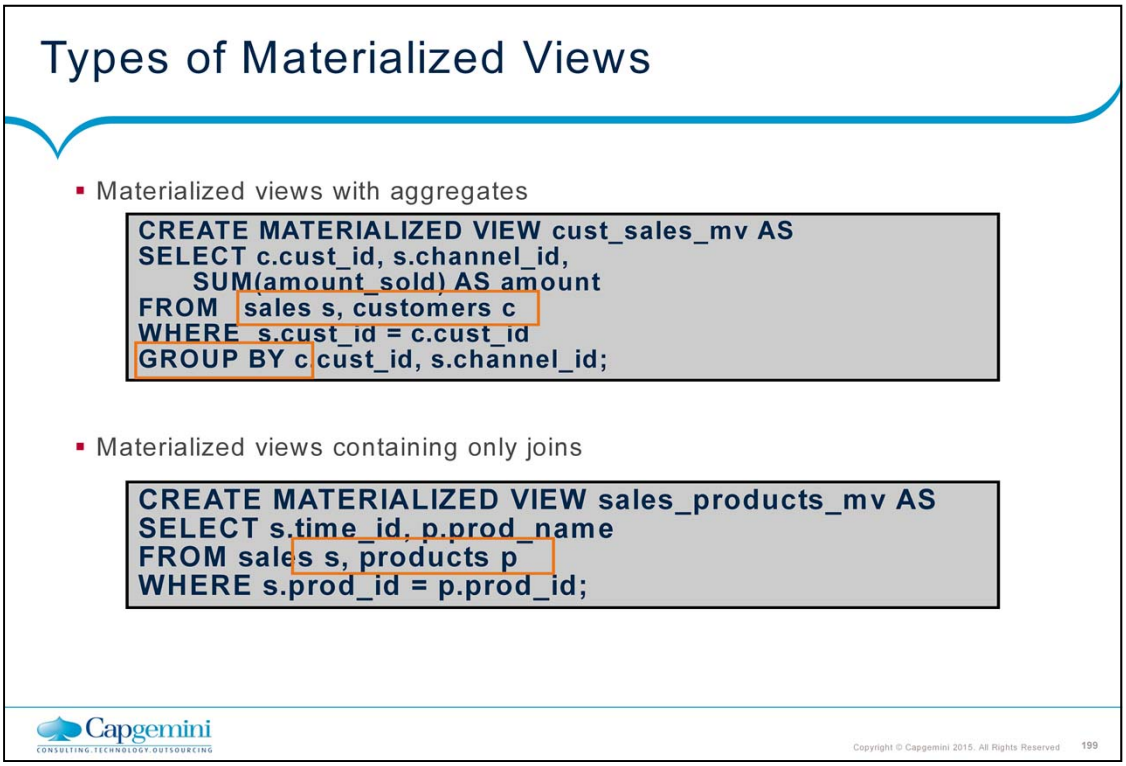
Creating Materialized Views: Syntax Options

- The CREATE MATERIALIZED VIEW syntax is similar to the CREATE SNAPSHOT command, which it replaces. There are some additional options:
  - The BUILD IMMEDIATE option causes the materialized view to be populated when the CREATE command is executed. This is the default behavior. You can choose the BUILD DEFERRED option, which creates the structure but does not populate it until the first refresh occurs.
  - Instead of the BUILD option, you can also specify ON PREBUILT TABLE when you want an existing summary table to be the source of a materialized view.
  - The ENABLE/DISABLE QUERY REWRITE clause determines whether query rewrites are automatically enabled for the materialized view. Query rewrite is enable by default in Oracle Database 10g.



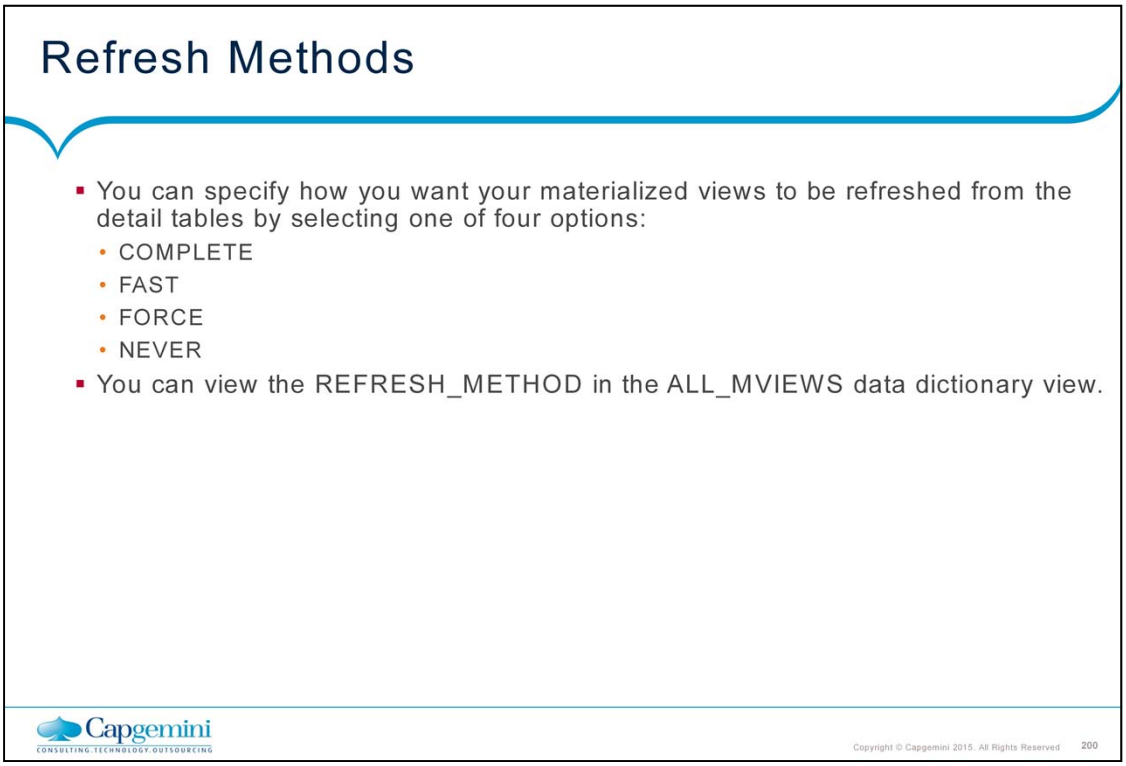
Creating Materialized Views: Example

- If a query involving aggregates, large or multiple joins, or both aggregates and joins is likely to be used multiple times, it can be more efficient to create a materialized view of the query results. This requires a single execution of the query and the storage space to preserve the results.
- If the queries are likely to be reused over time, you may also need a mechanism to update the materialized view as the base tables change. The performance and storage costs of maintaining the materialized view must be compared to the costs of reexecuting the original query whenever it is needed. This is discussed in more detail later in this lesson.
- When the optimizer chooses to use the view rather than the base table, this event is called a query rewrite. This is also discussed in more detail later in this lesson.
- Note: This example is a CREATE MATERIALIZED VIEW command. Details about optional command clauses are discussed later in this lesson.
- Note: In the previous page the default options in the syntax are underlined. This is not the complete syntax; see Oracle Database 10g SQL Reference for more details.



Types of Materialized Views

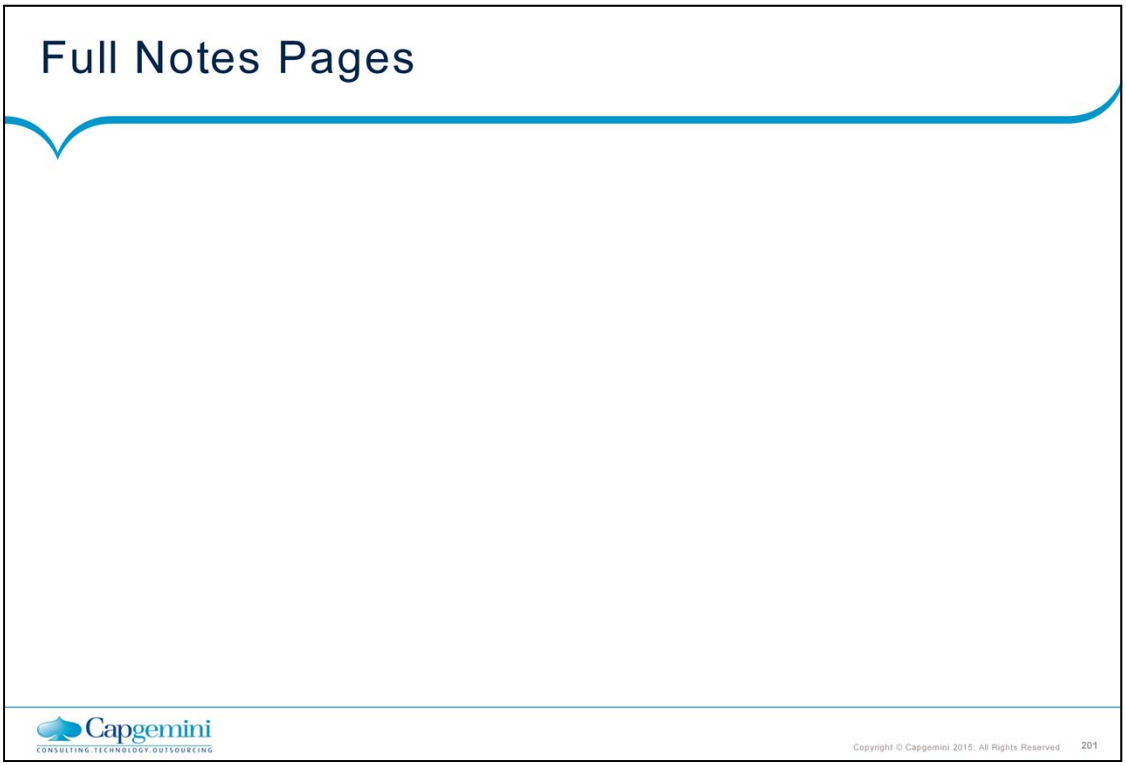
- The SELECT clause in the materialized view creation statement defines the data that the materialized view is to contain. Only a few restrictions limit what can be specified. Any number of tables can be joined together. However, they cannot be remote tables if you want to take advantage of query rewrite. In addition to tables, other elements such as views, inline views (subqueries in the FROM clause of a SELECT statement), subqueries in the WHERE clause, and materialized views can all be joined or referenced in the SELECT clause.
- In data warehouses, materialized views normally contain aggregates. This is the origin of the term summaries in data warehouses. Some materialized views contain only joins and no aggregates. The advantage of creating this type of materialized view is that expensive joins are precalculated.
- Note: The CREATE statements in the slide must include the ENABLE QUERY REWRITE clause. If they do not, then rewrite does not occur, and these materialized views are treated as replication materialized views.



Refresh Methods

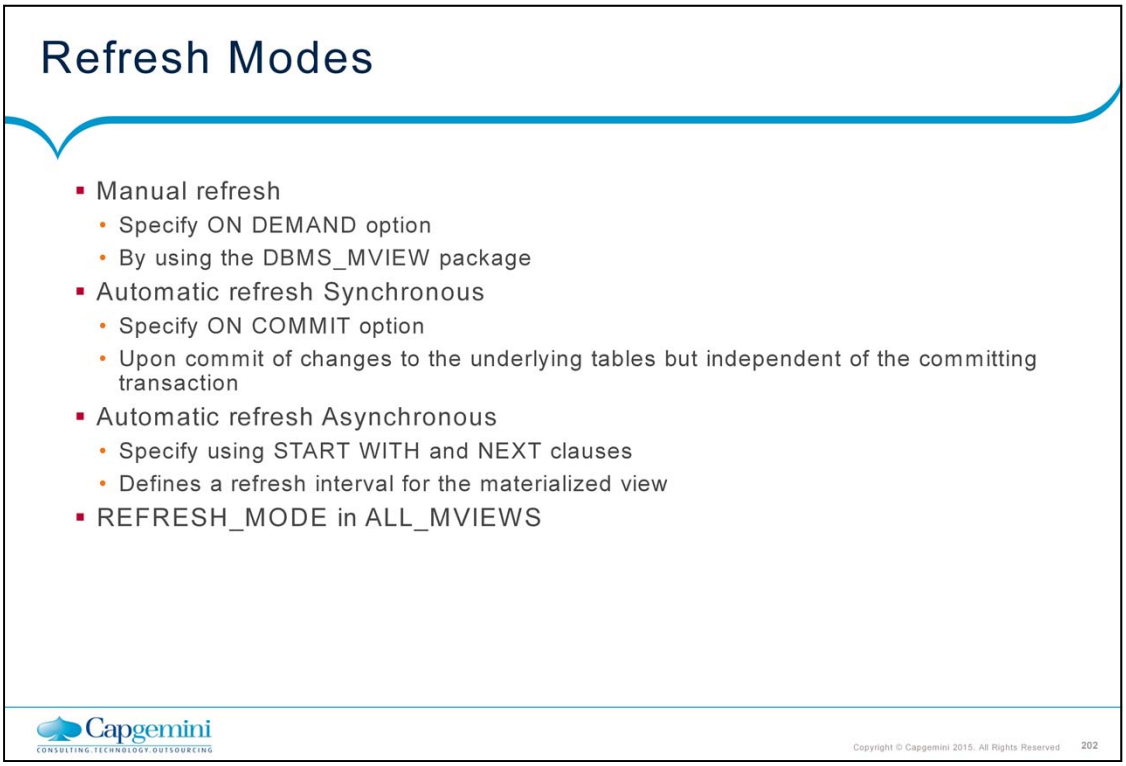
- COMPLETE refreshes by recalculating the detail query of the materialized view. This can be accomplished by deleting the table or truncating it. Complete refresh reexecutes the materialized view query, thereby completely recalculating the contents of the materialized view from the detail tables.
- FAST refreshes by incrementally adding the new data that has been inserted or updated in the tables.
- FORCE applies fast refresh if possible; otherwise, it applies COMPLETE refresh.
- NEVER prevents the materialized view from being refreshed with any Oracle refresh mechanism or procedure.

- Some types of nested materialized views cannot be fast refreshed. Use DBMS_MVIEW.EXPLAIN_MVIEW to identify those types of materialized views. You can refresh a tree of nested materialized views in the appropriate dependency order by specifying the nested = TRUE parameter with the DBMS_MVIEW.REFRESH method.



Refresh Methods (continued)

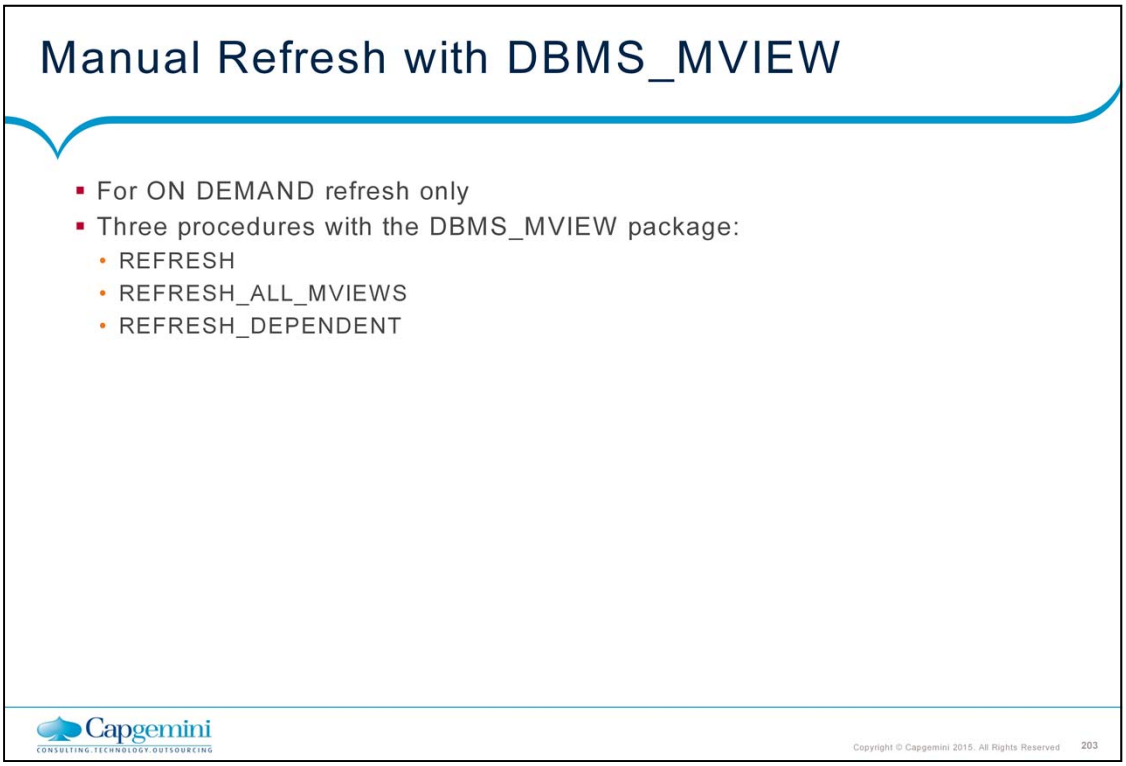
- Remember the following when deciding whether to use nested materialized views:
  - If you want to use fast refresh, you should fast refresh all the materialized views along any chain.
  - If you want the highest-level materialized view to be fresh with respect to the detail tables, you need to ensure that all materialized views in a tree are refreshed in the correct dependency order before refreshing the highest-level view. You can automatically refresh intermediate materialized views in a nested hierarchy using the nested = TRUE parameter.
  - When refreshing materialized views, you need to ensure that all materialized views in a tree are refreshed. If you refresh only the highest-level materialized view, the materialized views under it will be stale and you must explicitly refresh them. If you use the REFRESH procedure with the nested parameter value set to TRUE, only specified materialized views and their child materialized views in the tree are refreshed, and not their top-level materialized views.
  - Use the REFRESH_DEPENDENT procedure with the nested parameter value set to TRUE if you want to ensure that all materialized views in a tree are refreshed



Refresh Modes

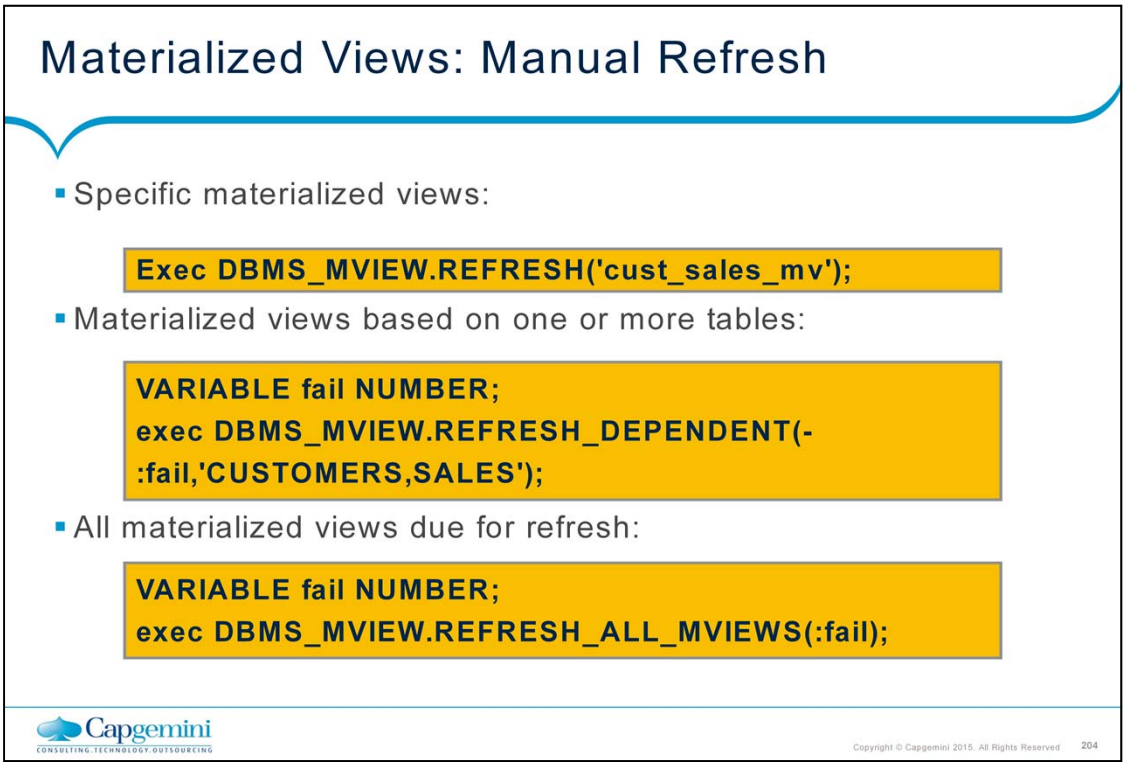
- ON DEMAND (default): Refresh occurs on user demand by using the DBMS_MVIEW package. This package provides several procedures and functions to manage materialized views, including the REFRESH, REFRESH_DEPENDENT, and REFRESH_ALL_MVIEWS procedures.
- ON COMMIT: Refresh occurs automatically when a transaction that modified one of the detail tables of the materialized view commits. This mode can be specified as long as the materialized view is fast refreshable. If a materialized view fails during refresh at commit time, the user must explicitly invoke the refresh procedure using the DBMS_MVIEW package after addressing the errors specified in the trace files. Until this is done, the view is no longer refreshed automatically at commit time.
- At a specified time: Refresh of a materialized view can be scheduled to occur at a specified time (for example, it can be refreshed every Monday at 9:00 a.m. by using the START WITH and NEXT clauses). You can create jobs using DBMS_SCHEDULER to do this.

- Note: If you specify ON COMMIT or ON DEMAND, you cannot also specify START WITH or NEXT.



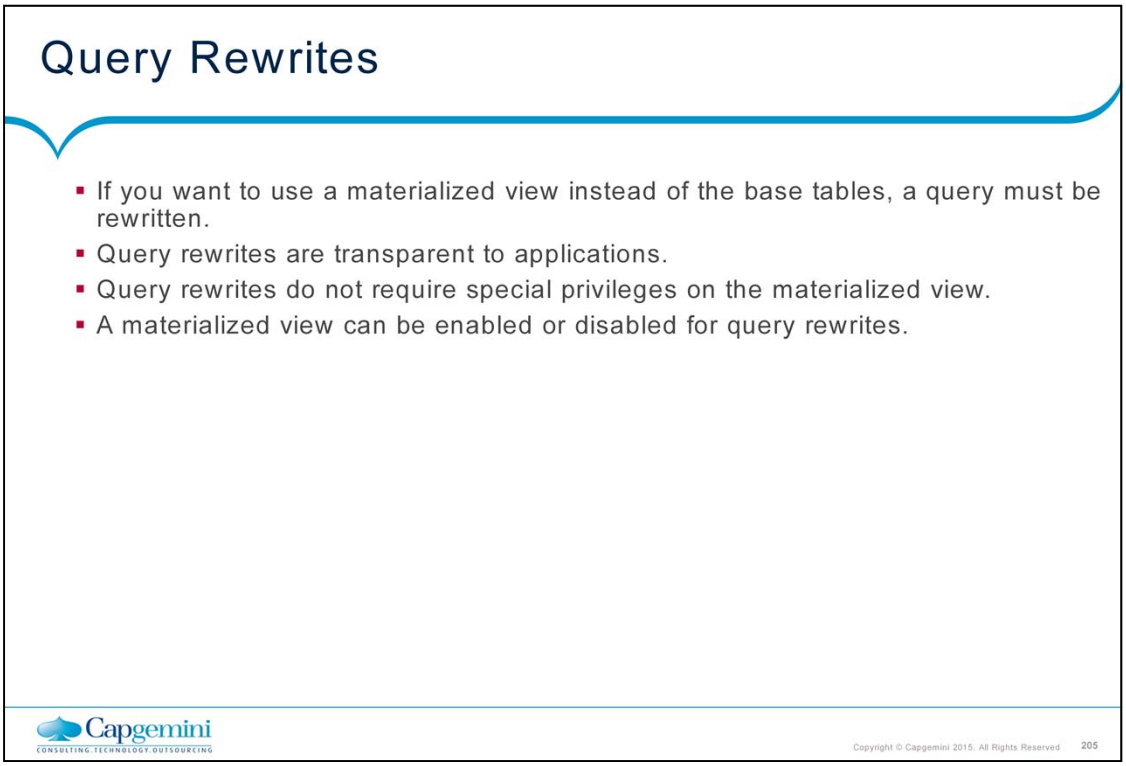
Manual Refresh with DBMS_MVIEW

- You can control the time at which refresh of the materialized views occurs by specifying ON DEMAND refreshes. In this case, the materialized view can be refreshed only by calling one of the procedures in the DBMS_MVIEW package. DBMS_MVIEW provides three different types of refresh operations:
  - DBMS_MVIEW.REFRESH: Refresh one or more materialized views
  - DBMS_MVIEW.REFRESH_ALL_MVIEWS: Refresh all materialized views
  - DBMS_MVIEW.REFRESH_DEPENDENT: Refresh all table-based materialized views that depend on a specified detail table or list of detail tables



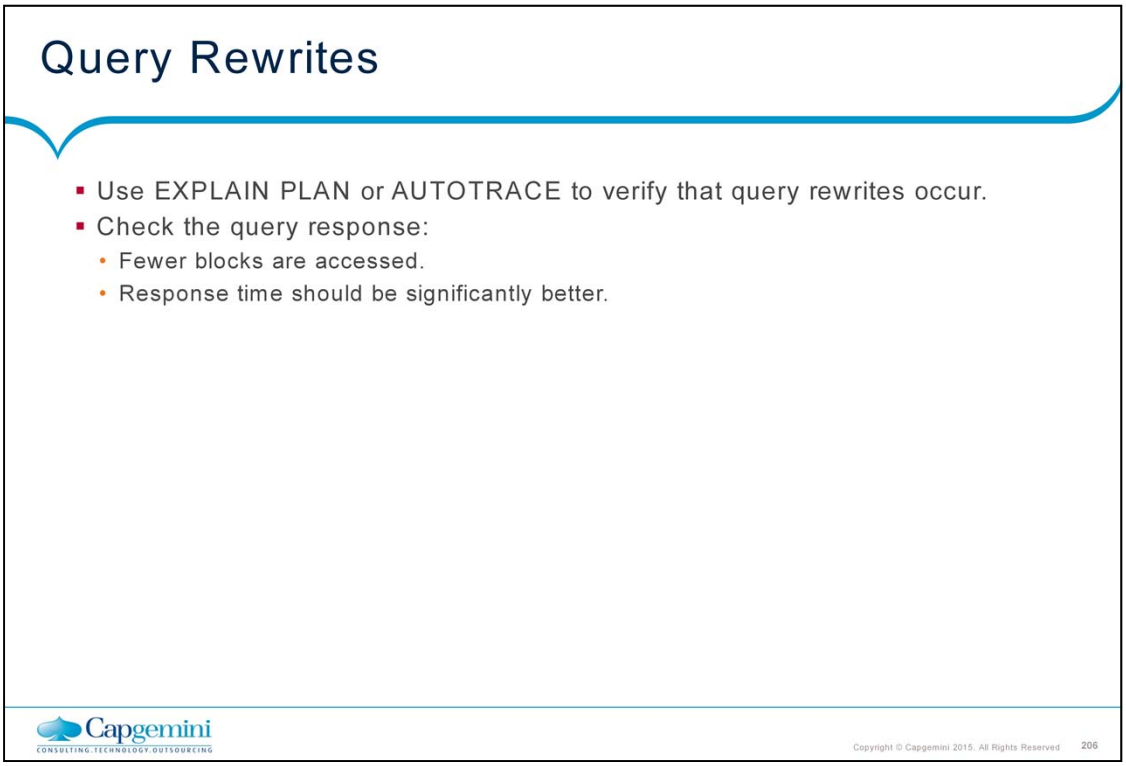
Materialized Views: Manual Refresh

- Possible refresh scenarios for materialized views include the following:
  - Use the REFRESH procedure to refresh specific materialized views.
  - Use the REFRESH_DEPENDENT procedure to refresh all materialized views that depend on a given set of base tables.
  - Use the REFRESH_ALL_MVIEWS procedure to refresh all materialized views that have not been refreshed since the last bulk load to one or more detail tables. This accepts an out parameter that is used to return the number of failures.
- The procedures in the package use several parameters to specify the following:
  - Refresh method
  - Whether to proceed if an error is encountered
  - Whether to use a single transaction (consistent refresh)
  - Rollback segment to use



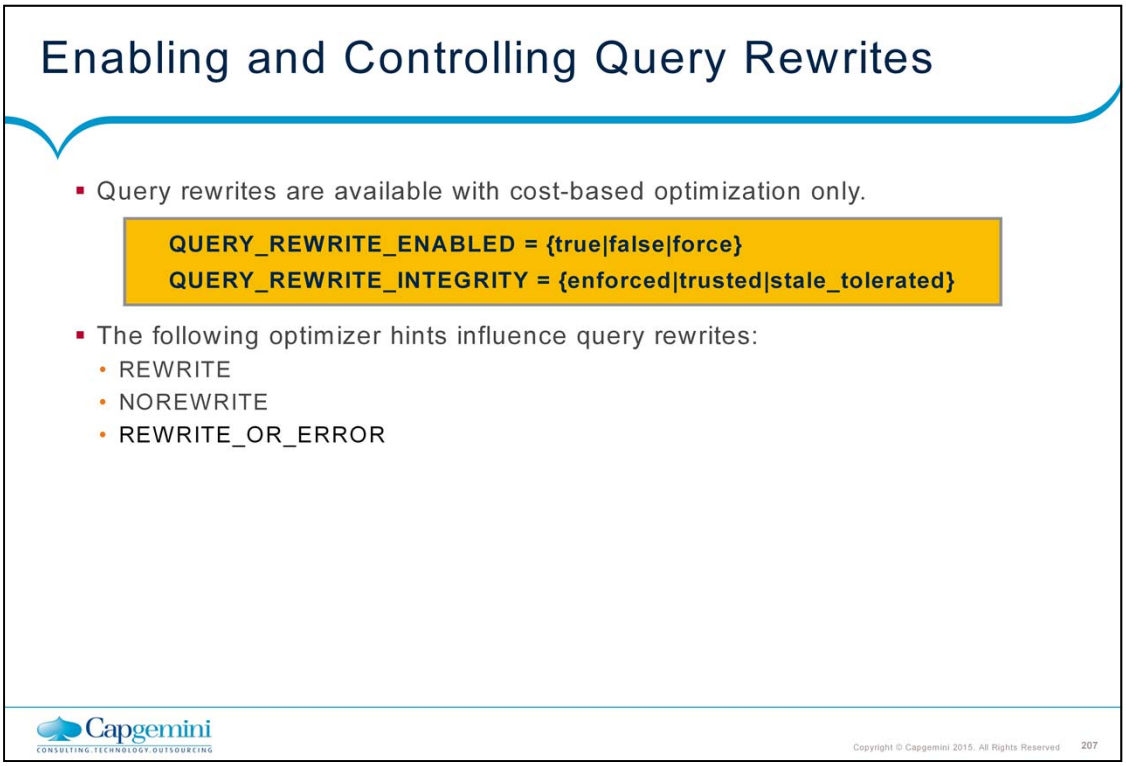
Query Rewrites

- Because accessing a materialized view may be significantly faster than accessing the underlying base tables, the optimizer rewrites a query to access the view when the query allows it. The query rewrite activity is transparent to applications. In this respect, their use is similar to the use of an index.
- Users do not need explicit privileges on materialized views to use them. Queries that are executed by any user with privileges on the underlying tables can be rewritten to access the materialized view.
- A materialized view can be enabled or disabled. A materialized view that is enabled is available for query rewrites, as in the following example:
    - ALTER MATERIALIZED VIEW cust_sales_mv DISABLE QUERY REWRITE;



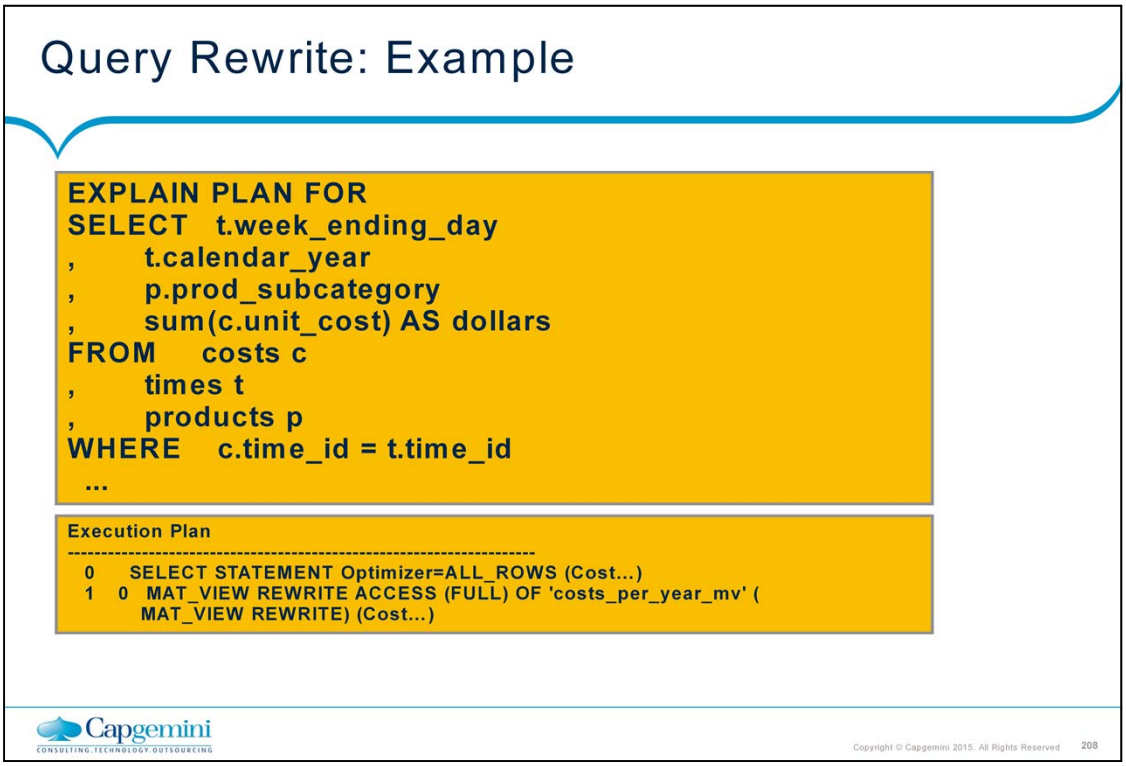
Query Rewrites (continued)

- The best way to determine whether query rewrites occur is to use the EXPLAIN PLAN command or the AUTOTRACE setting in SQL*Plus. The execution plan shows whether a rewrite has taken place that uses the materialized view. You should also notice improved response time if a materialized view is used by the optimizer. You can also use the DBMS_MVIEW.EXPLAIN_REWRITE procedure to verify whether rewrites can occur.
- Note: There are several system privileges that control whether you are allowed to create materialized views and modify them, and whether query rewrites are enabled for you. Contact your local database administrator to set this up properly.
- There are also many data dictionary views that contain information about materialized views. For details, see Oracle Database 10g Performance Guide and Reference and Oracle Database 10g Reference.



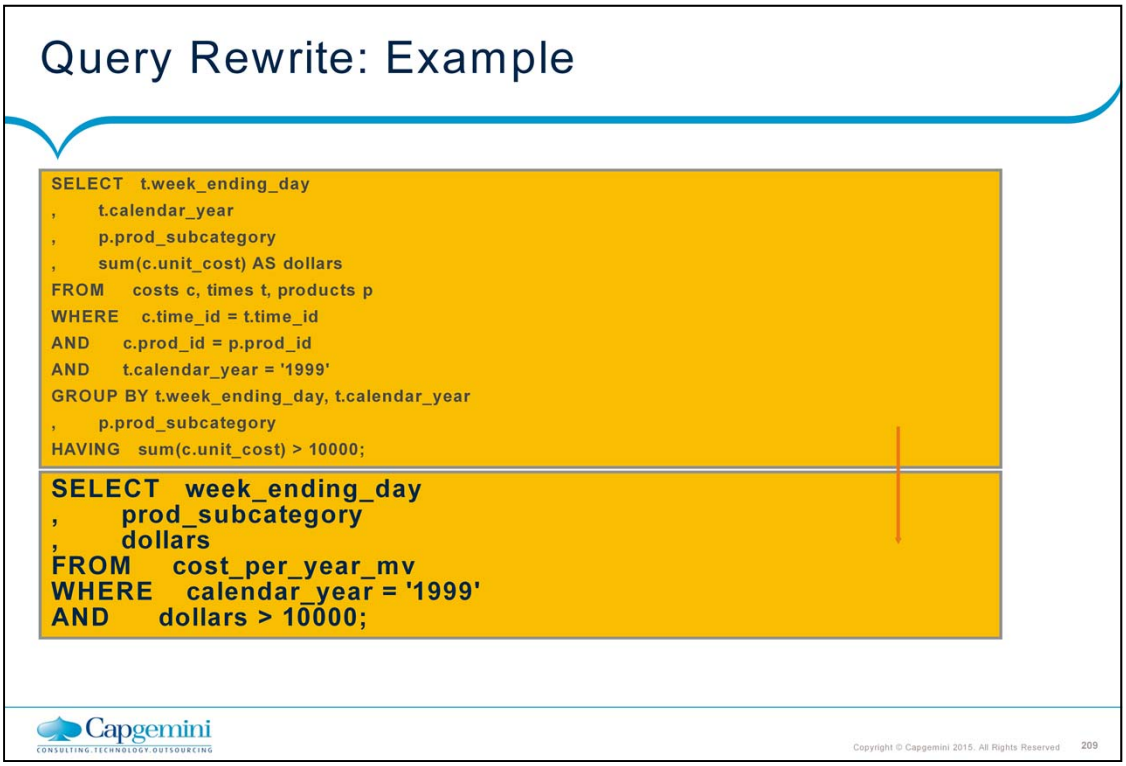
Enabling and Controlling Query Rewrites

- OPTIMIZER_MODE: Query rewrites are available with cost-based optimization only.
- QUERY_REWRITE_ENABLED: This is a dynamic instance/session parameter that can be set to the following values:
  - TRUE: Cost-based rewrite
  - FALSE: No rewrite
  - FORCE: Forced rewrite
- QUERY_REWRITE_INTEGRITY: This is also a dynamic instance/session parameter. It accepts the following values:
  - ENFORCED (default) enables query rewrites only if the optimizer can guarantee consistency. Only fresh materialized views and enabled validated constraints are used for query rewrites.
  - TRUSTED allows query rewrites based on declared (not necessarily enforced) relationships. All fresh materialized views and constraints with the RELY flag are used for query rewrites.
  - STALE_TOLERATED allows stale materialized views that do not contain the latest data to be used.
- Note: If query rewrite does not occur, try STALE_TOLERATED mode first. If the optimizer does not rewrite in this mode, it will never rewrite.



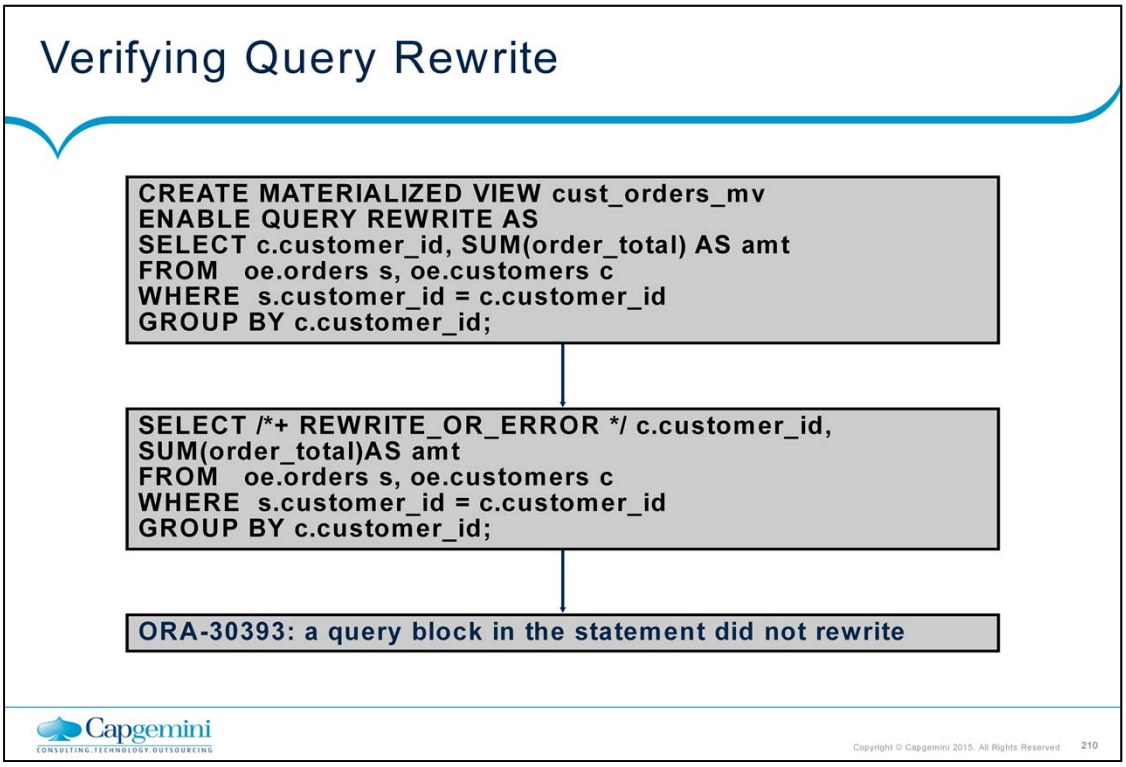
Query Rewrite: Example

- The execution plan shows that the materialized view is accessed instead of joining all four base tables to produce the result.
- A REWRITE or NOREWRITE hint overrides a materialized view's definition that is set in the CREATE or ALTER MATERIALIZED VIEW command with the QUERY REWRITE clause.
- This example shows a transparent query rewrite in which the query exactly matches the materialized view definition. The next slide shows an example of a query that does not match the materialized view definition.



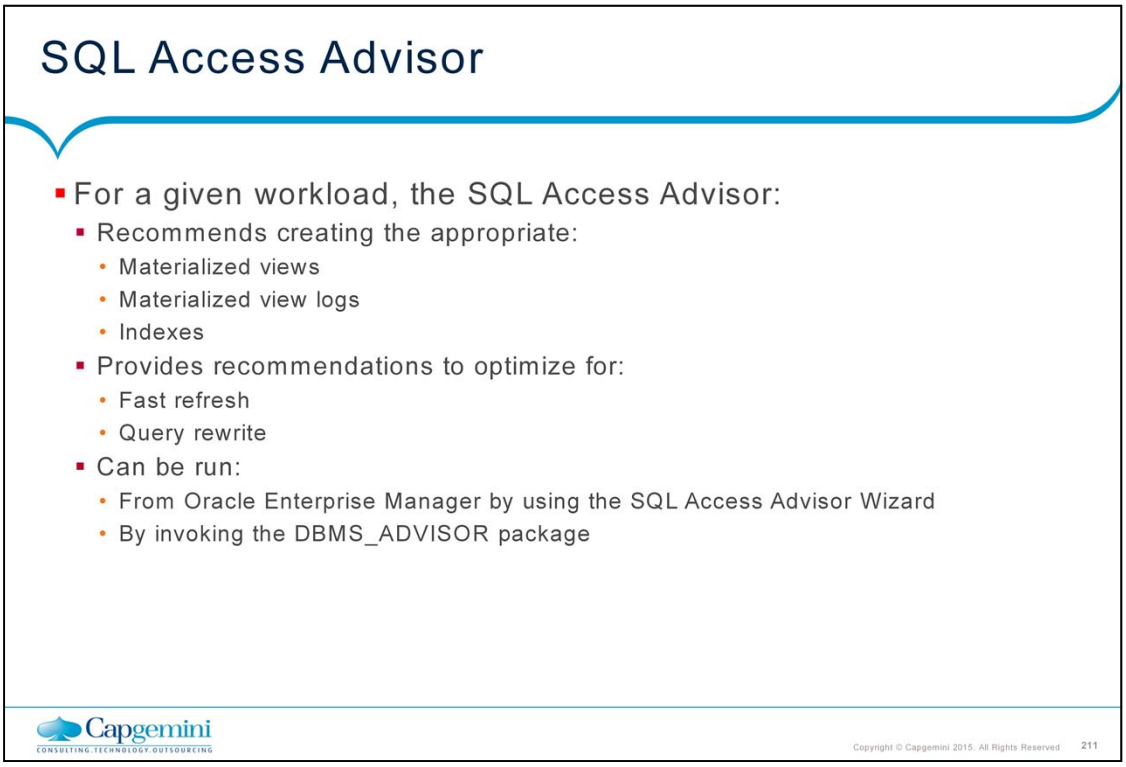
Query Rewrite: Example (continued)

- The optimizer can use the materialized view created earlier to satisfy the query.
- Note: The query in the slide does not exactly match the materialized view definition. You have added a nonjoin predicate on line 8 and a HAVING clause on the last line. The nonjoin predicate is merged into the rewritten query against the materialized view, and the HAVING clause is translated into a second component of the WHERE clause.



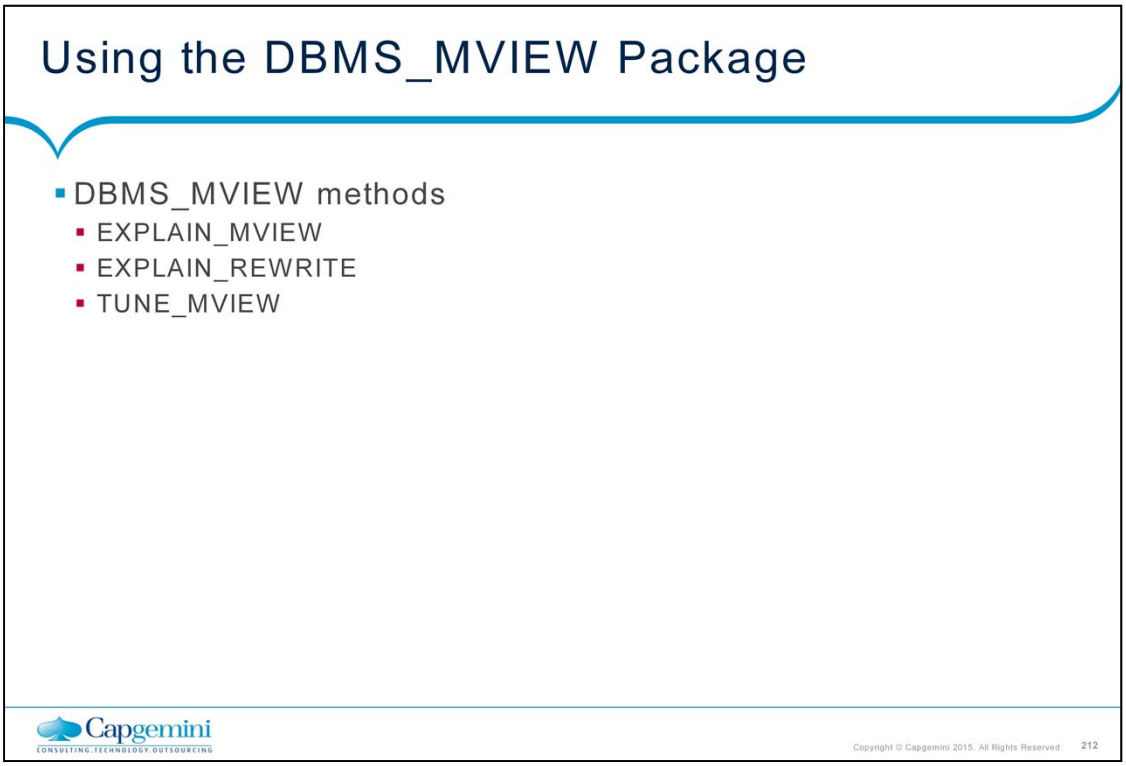
Verifying Query Rewrite

- The example in the slide shows a situation in which creating a materialized view is not useful. Using the REWRITE_OR_ERROR hint in a query causes the following error if the query fails to rewrite:
  - ORA-30393: a query block in the statement did not rewrite
- You can then decide if such a materialized view is worth retaining, based on testing with other queries.



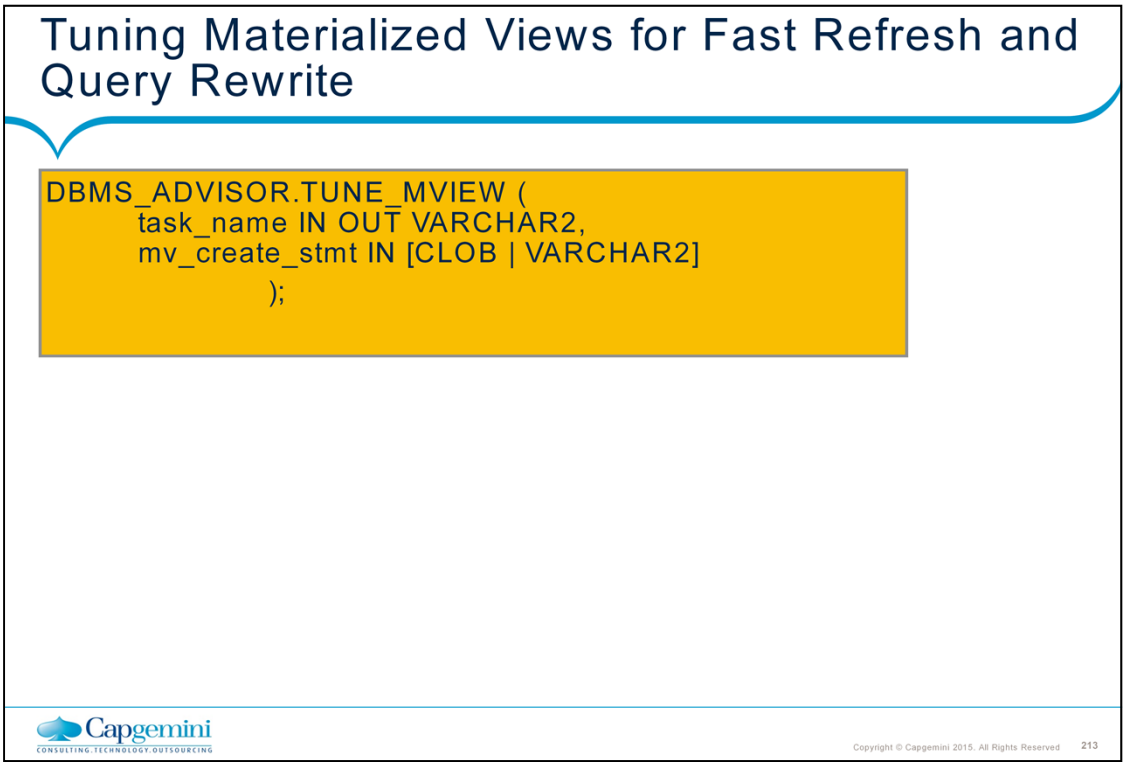
SQL Access Advisor

- The SQL Access Advisor (SAA) helps you achieve your performance goals by recommending the proper set of materialized views, materialized view logs, and indexes for a given workload. Understanding and using these structures is essential when optimizing SQL, because they can result in significant performance improvements in data retrieval. The advantages, however, come with a cost. Creation and maintenance of these objects can be time consuming, and space requirements can be significant. The SAA also recommends how to optimize materialized views so that they can be fast refreshable and take advantage of general query rewrite.
- The SAA can be run from Oracle Enterprise Manager by using the SQL Access Advisor Wizard or by invoking the DBMS_ADVISOR package. The DBMS_ADVISOR package consists of a collection of analysis and advisory functions and procedures that can be called from any PL/SQL program. If a workload is not provided, the SAA can generate and use a hypothetical workload also.



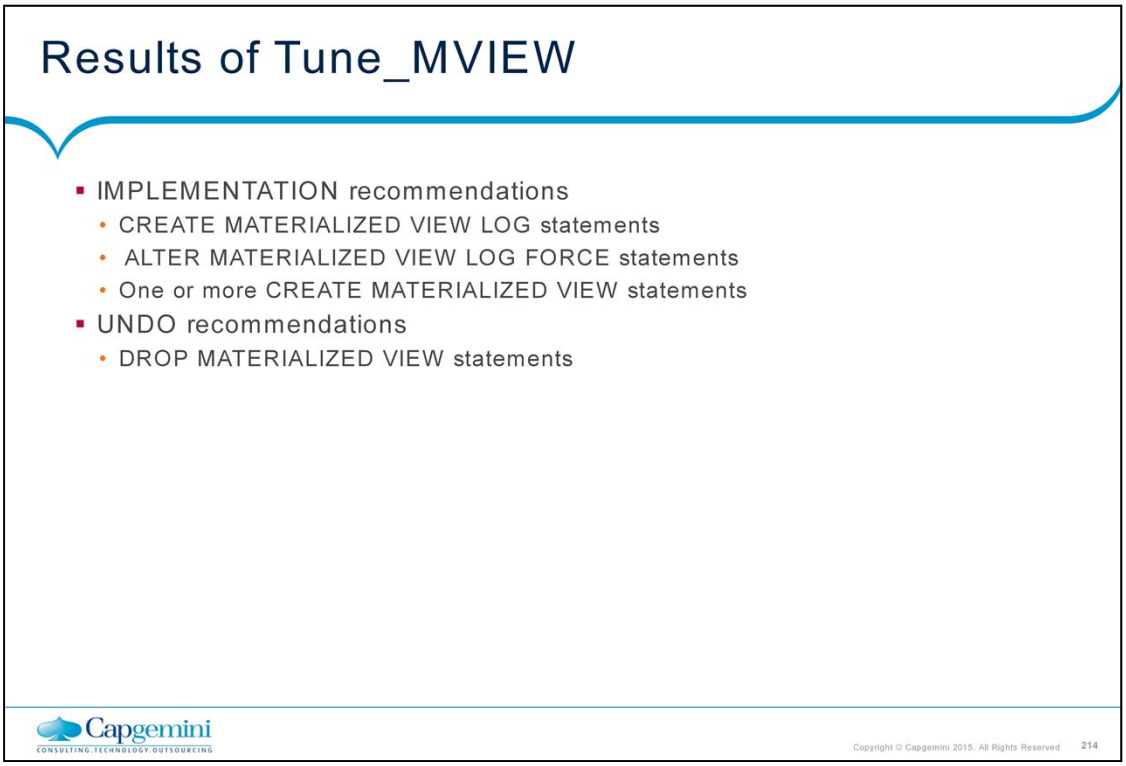
Using the DBMS_MVIEW Package

- Several DBMS_MVIEW procedures help you with materialized view fast refresh and query rewrite.
- The EXPLAIN_MVIEW procedure tells you whether a materialized view is fast refreshable or eligible for general query rewrite. EXPLAIN_REWRITE tells you whether query rewrite will occur. However, neither tells you how to achieve fast refresh or query rewrite.
- To further facilitate the use of materialized views, the TUNE_MVIEW procedure shows how to optimize your CREATE MATERIALIZED VIEW statement and how to meet other requirements (such as materialized view log for fast refresh and general query rewrite).



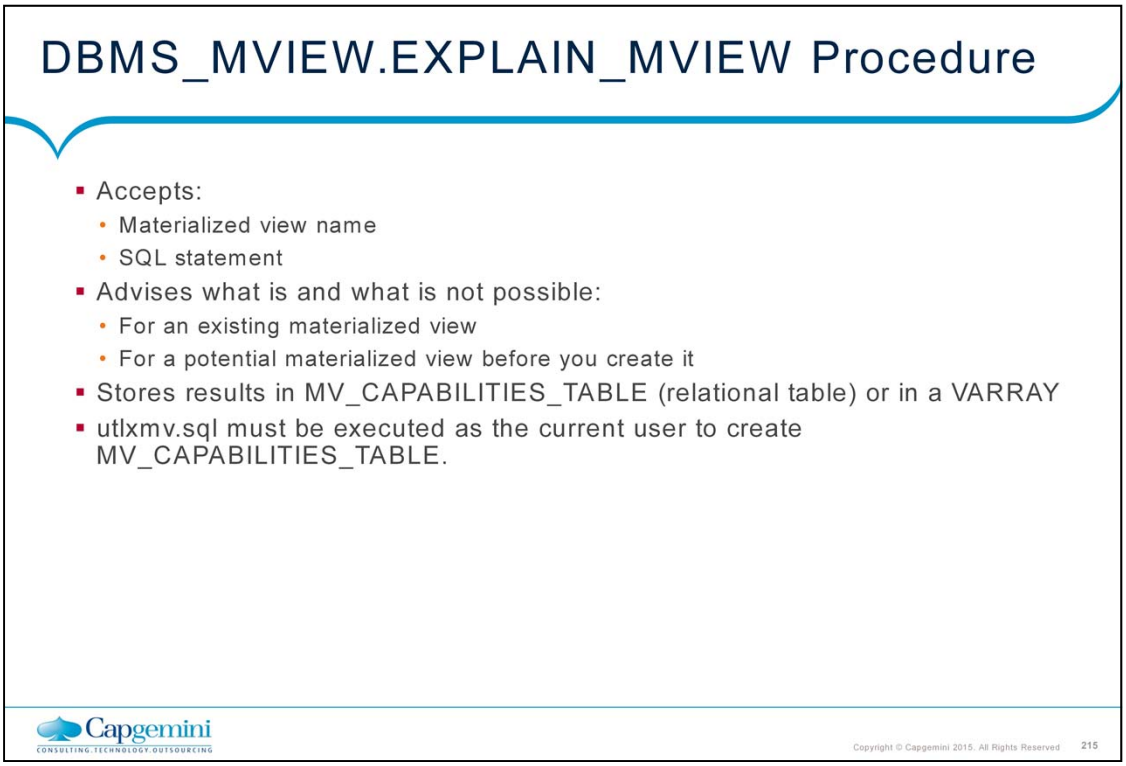
Tuning Materialized Views for Fast Refresh and Query Rewrite

- The DBMS_MVIEW.TUNE_MVIEW method analyzes and processes the input statement and generates two sets of output results: one for the materialized view implementation and the other for undoing the create materialized view operations. The two sets of output results can be accessed through Oracle views or can be stored in external script files created by the SAA. These external script files are ready to execute to implement the materialized view.
- The TUNE_MVIEW procedure takes two input parameters: task_name and mv_create_stmt. Task_name is a user-provided task identifier used to access the output results. Mv_create_stmt is a complete CREATE MATERIALIZED VIEW statement that is to be tuned. If the input CREATE MATERIALIZED VIEW statement does not have the clauses REFRESH FAST or ENABLE QUERY REWRITE, or both, TUNE_MVIEW will use the default clauses REFRESH FORCE and DISABLE QUERY REWRITE to tune the statement to be fast refreshable (if possible) or only complete refreshable.



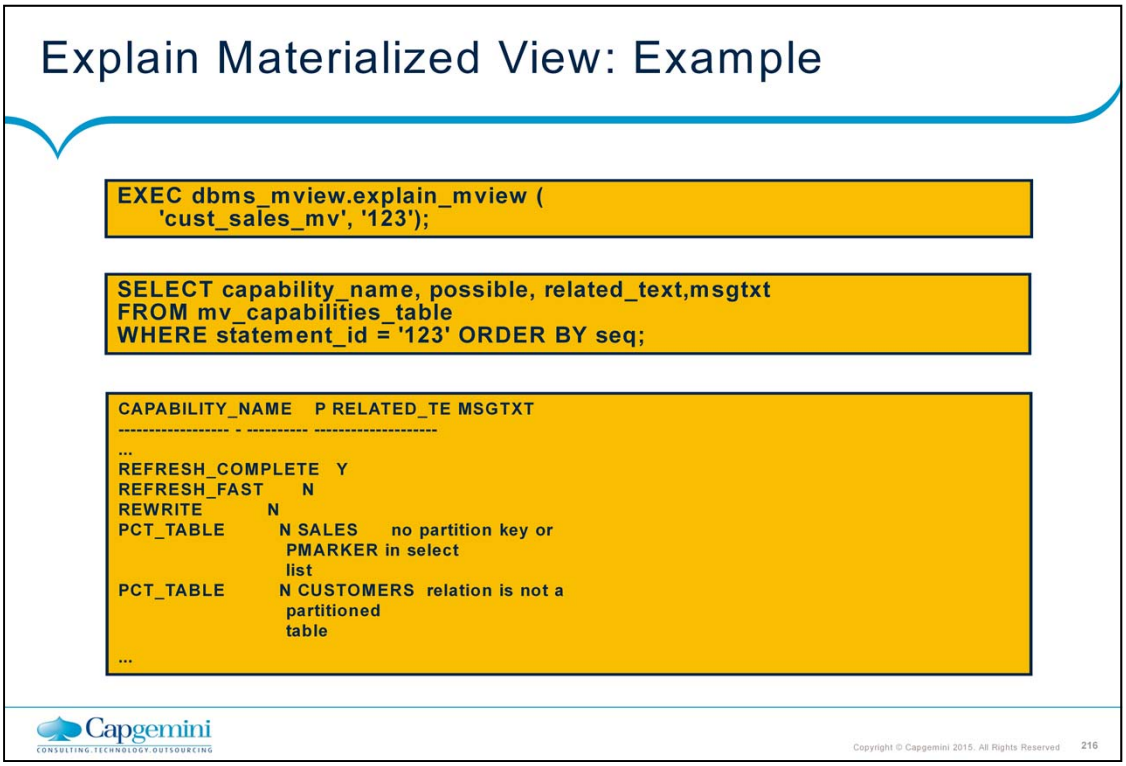
Results of Tune_MVIEW

- When the ENABLE QUERY REWRITE clause is specified, TUNE_MVIEW also fixes the statement by using a process similar to REFRESH FAST, which redefines the materialized view so that many of the advanced forms of query rewrite are possible.
- The TUNE_MVIEW procedure generates two sets of output results as executable statements. One set of output results, IMPLEMENTATION, is for implementing materialized views and required components such as materialized view logs to achieve fast refreshability and query rewritability. The other set of output results, UNDO, is for dropping the materialized views (if you decide that they are not required).
- The output statements for the IMPLEMENTATION process include:
  - CREATE MATERIALIZED VIEW LOG statements, which create any missing materialized view logs required for fast refresh
  - ALTER MATERIALIZED VIEW LOG FORCE statements, which fix any materialized view log–related requirements (such as missing filter columns, sequence, and so on) that are required for fast refresh



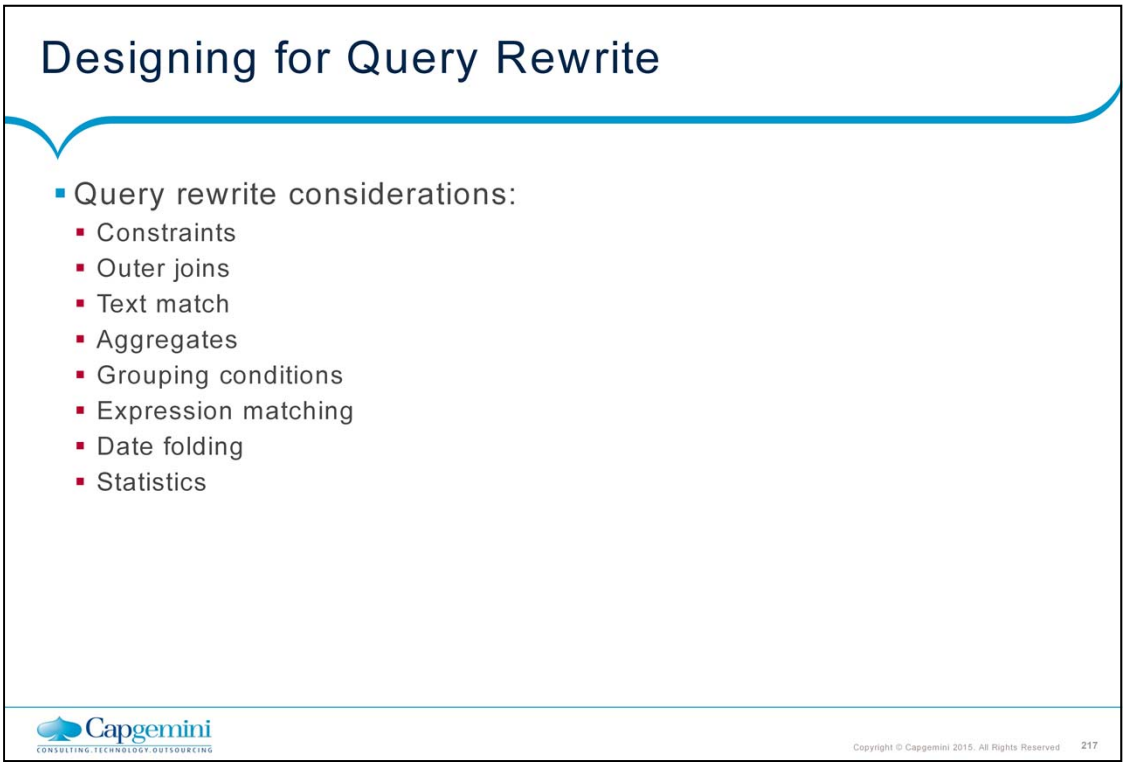
DBMS_MVIEW.EXPLAIN_MVIEW Procedure

- The purpose of the Explain Materialized View procedure is to advise what is and is not possible with a given materialized view or potential materialized view. This package advises the user by providing answers to the following questions:
  - Is this materialized view fast refreshable?
  - What are the types of query rewrite that can be done with this materialized view?
- The process for using this package is very simple. The procedure DBMS_MVIEW.EXPLAIN_MVIEW is called, passing in as parameters the schema and materialized view name for an existing materialized view. Alternatively, you can specify the select string for a potential materialized view. The materialized view or potential materialized view is then analyzed, and the results are written either to a table called MV_CAPABILITIES_TABLE (the default) or to a VARRAY of type ExplainMVArrayType called MSG_ARRAY.
- Note: Except when using VARRAYS, you must run the utlxmv.sql script prior to calling EXPLAIN_MVIEW. The script creates the MV_CAPABILITIES_TABLE in the current schema. The script is found in the admin directory.



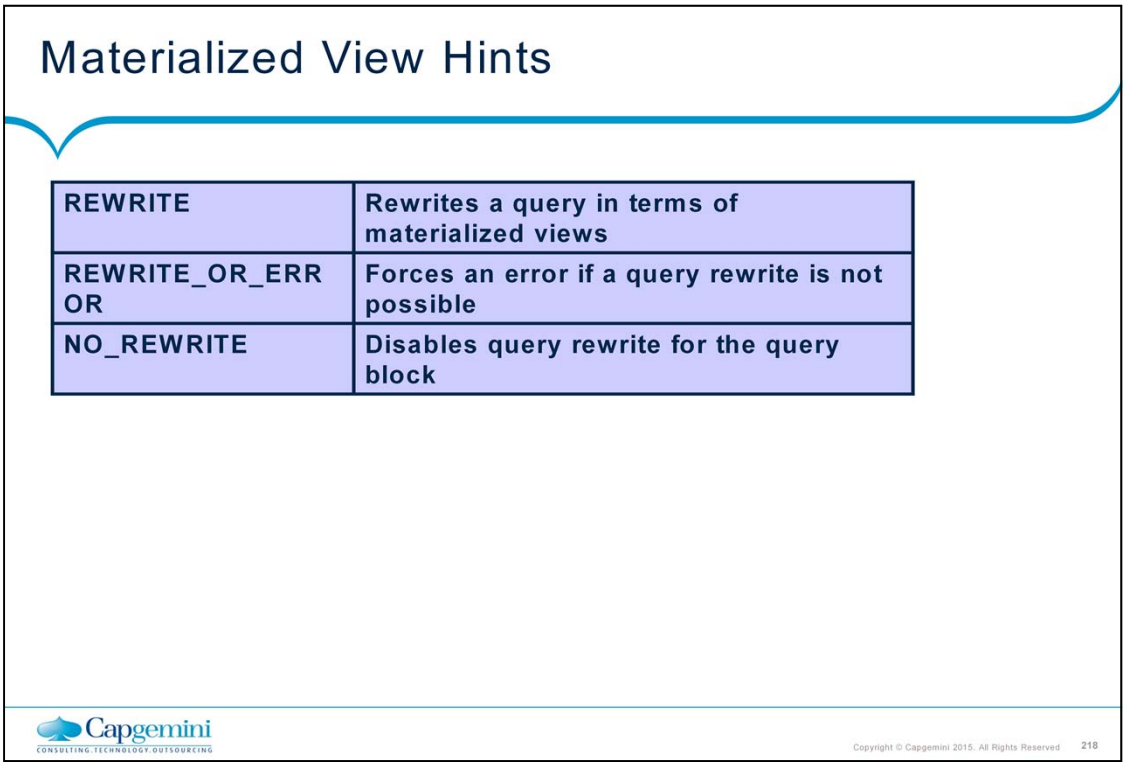
Explain Materialized View: Example

- In the example in the slide, suppose that the MV_CAPABILITIES_TABLE has already been created. You want to analyze an already-existing materialized view called CUST_SALES_MV that was created in the SH schema. You need to assign an ID for this analysis so that we can retrieve it subsequently in the MV_CAPABILITIES_TABLE. You also need to use the SEQ column in an ORDER BY clause so that the rows appear in a logical order.
- If a capability is not possible, N appears in the P column and an explanation appears in the MSGTXT column. If a capability is not possible for more than one reason, a row is displayed for each reason.



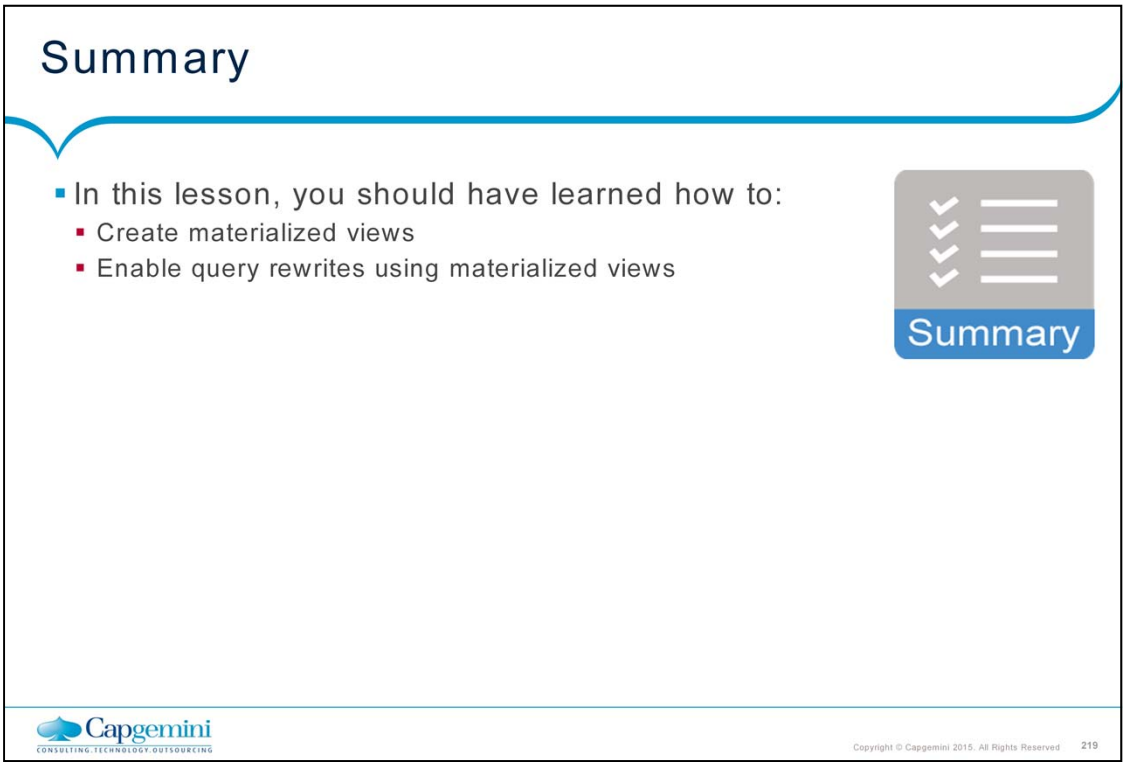
Designing for Query Rewrite

- Constraints: Make sure that all inner joins that are referred to in a materialized view have referential integrity (foreign key and primary key constraints) with additional NOT NULL constraints on the foreign key columns. Because constraints tend to impose a large overhead, you could make them NO VALIDATE and RELY and set the parameter QUERY_REWRITE_INTEGRITY to STALE_TOLERATED or TRUSTED. However, if you set QUERY_REWRITE_INTEGRITY to ENFORCED, all constraints must be enabled, enforced, and validated to obtain maximum rewritability. You should avoid using the ON DELETE clause because it can lead to unexpected results.
- Outer joins: Another way of avoiding constraints is to use outer joins in the materialized view. Query rewrite will be able to derive an inner join in the query, such as (A.a = B.b), from an outer join in the materialized view (A.a = B.b(+)) as long as the row ID of B or column B.b is available in the materialized view. Most of the support for rewrites with outer joins is provided for materialized views with joins only. To exploit it, a materialized view with outer joins should store the row ID or primary key of the inner table of an outer join.



Materialized View Hints

- REWRITE, NOREWRITE, and REWRITE_OR_ERROR are hints that are used with materialized views to control query rewrites.
- REWRITE
- The REWRITE hint forces the cost-based optimizer to rewrite a query in terms of materialized views (when possible) without cost consideration. Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, then the Oracle Server uses that view regardless of its cost.
- The server does not consider views outside of the list. If you do not specify a view list, then the server searches for an eligible materialized view and always uses it regardless of its cost.
- REWRITE_OR_ERROR
- This hint forces an error if a query rewrite is not possible.
- NOREWRITE
- The NOREWRITE hint disables query rewrite for the query block, overriding the setting of the QUERY_REWRITE_ENABLED parameter. Use the NOREWRITE hint on any query block of a request.



Summary

- This lesson introduced you to materialized views and query rewrites. A materialized view stores both the definition of a view and the rows resulting from the execution of the view. Like a view, it uses a query as its basis; however, the query is executed at the time the view is created and the results are stored in a table.
- Because accessing a materialized view can be significantly faster than accessing the underlying base tables, the optimizer rewrites a query to access the view when the query allows it. The query rewrite activity is transparent to applications.

Summary

- Introduction to SQL tuning
- Describe why the SQL statements are performing poorly
- Introduction to Oracle Optimizer
- Discuss the need for Optimizer
- Explain the various phases of Optimization
- Gather Execution Plans
- Interpret Execution Plans
- Interpret the output of TKPROF
- Gather Optimizer statistics
- Use Hints appropriately

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

