Oracle Database 12c: SQL Tuning for Developers

Student Guide - Volume I

D79995GC10
Edition 1.0
October 2013
D84109

ORACLE

Author

Dimpi Rani Sarmah

Technical Contributors and Reviewers

Diganta Choudhury
Laszlo Czinkoczki
Joel Goodman
Nancy Greenberg
Naoki Kato
Sean Kim
Bryan Roberts
Swarnapriya Shridhar
Raza Siddiqui
Branislav Valny

Editors

Smita Kommini
Vijayalakshmi Narasimhan

Graphic Designers

Seema Bopaiah
Divya Thallap

Publishers

Pavithran Adka
Nita Brozowski
Srividya Rameshkumar

Contents

1 Course Introduction

Lesson Agenda

- Course Objectives, Course Outline, Prerequisites, and Activities Used in the Course
- Sample Schemas Used in the Course
- Class Account Information
- SQL Environments Available in the Course
- Overview of the Workshops, Demos, Code Examples, Solution Scripts, and Appendixes Used in the Course

Course Objectives

After completing this course, you should be able to:

- Choose an appropriate SQL tuning approach
- Gather suspected session statistics by using the SQL trace facility and interpret the traced information
- Identify poorly performing SQL statements
- Use basic tuning techniques to tune inefficient SQLs
- Interpret execution plans
- Describe the Oracle optimizer fundamentals
  - Explain the various phases of optimization
  - Control the behavior of the optimizer
  - Perform optimizer access paths, join, and other operations
  - List optimizer statistics best practices
- Manage SQL performance through changes

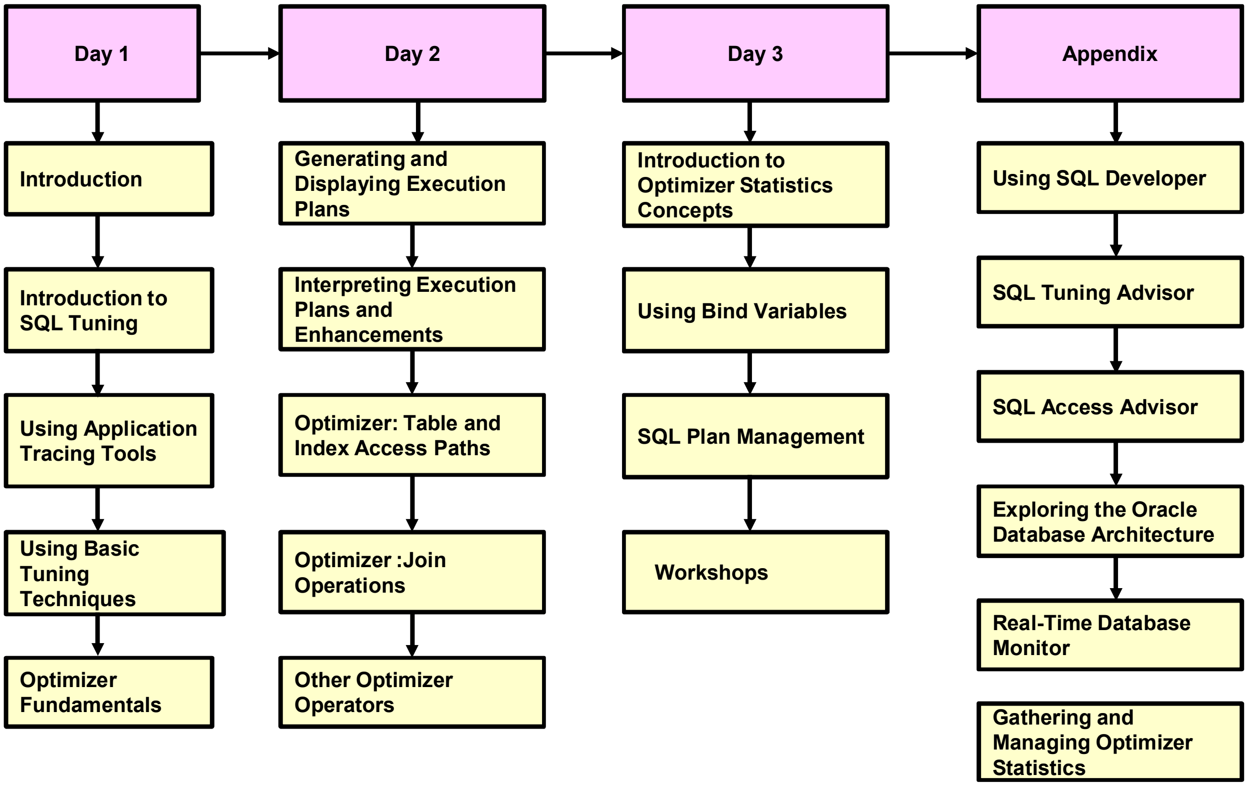
Audience and Prerequisites

- This course is intended for experienced Oracle SQL developers or Oracle DBAs who require a thorough understanding of Serial SQL execution.
- A good working knowledge and understanding of SQL statements is assumed.
- The following training for Oracle Database 12*c* is recommended: *Introduction to SQL* course or equivalent experience

This course does not teach the fundamentals of SQL. Rather, it focuses on topics that are specific to tuning SQL statements.

A good working knowledge and understanding of SQL statements is assumed. Prior training for Oracle Database 12*c* (*Introduction to SQL* course or equivalent experience) is recommended.

Course Outline Map



Activities

This course includes three types of activities to help you understand the concepts:

- Quizzes test your knowledge of the important concepts in each chapter.
- Practices reinforce your learning through guided, hands-on exercises.
- Workshops give you an opportunity to gain practical experience through problem solving.

ORACLE

About You

To ensure that the class is customized to meet your specific needs and to encourage interaction, answer the following questions:

- Which organization do you work for?
- What is your role in your organization?
- What is your level of SQL tuning expertise?
- What Oracle versions do you use?
- What do you hope to learn from this class?

Lesson Agenda

- Course Objectives, Course Outline, Prerequisites, and Activities Used in the Course
- Sample Schemas Used in the Course
- Class Account Information
- SQL Environments Available in the Course
- Overview of the Workshops, Demos, Code Examples, Solution Scripts, and Appendixes Used in the Course

Sample Schemas Used in the Course

- `Sales History` (`SH`) schema
- `Human Resources` (`HR`) schema
- `Order Entry(OE)` schema
- `Scott` schema

ORACLE

The sample company portrayed by Oracle Database sample schemas operates worldwide to fulfill orders for several different products. The company has the following divisions:

- The Sales History division tracks business statistics to facilitate business decisions.
- The Human Resources division tracks information about employees and facilities.

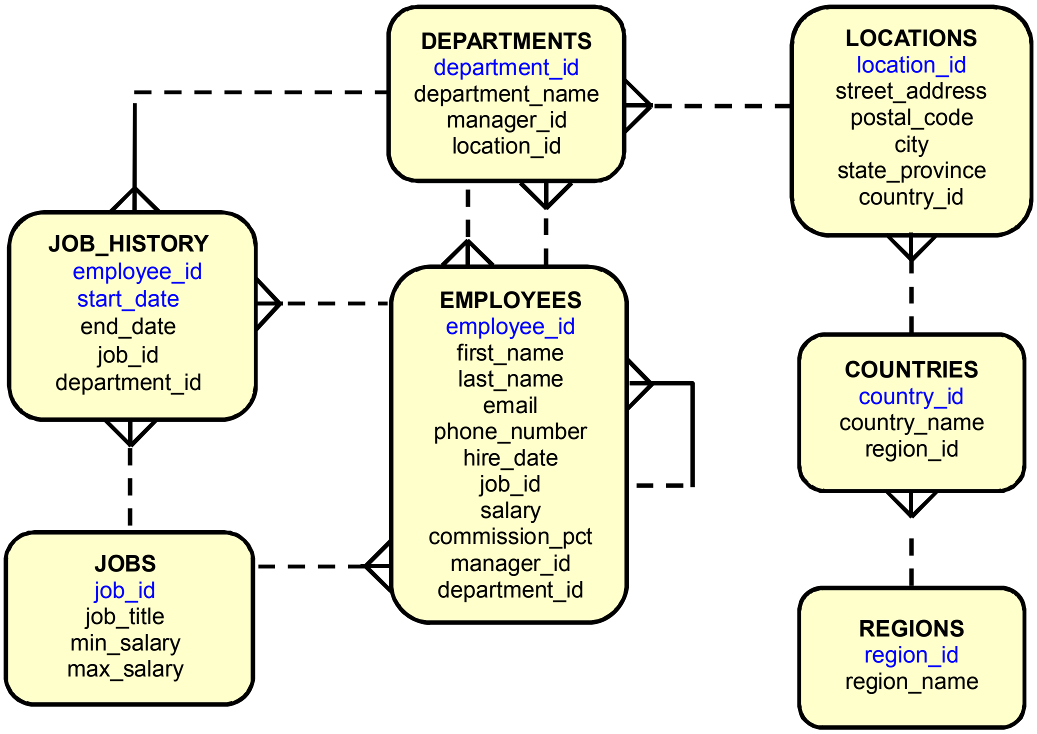
Each division is represented by a schema.

All scripts necessary to create the `SH` schema reside in the `$ORACLE_HOME/demo/schema/sales_history` folder.

All scripts necessary to create the `HR` schema reside in the `$ORACLE_HOME/demo/schema/human_resources` folder.

Note: The code examples and the workshops in this course specify the schema that must be used.

Human Resources (HR) Schema



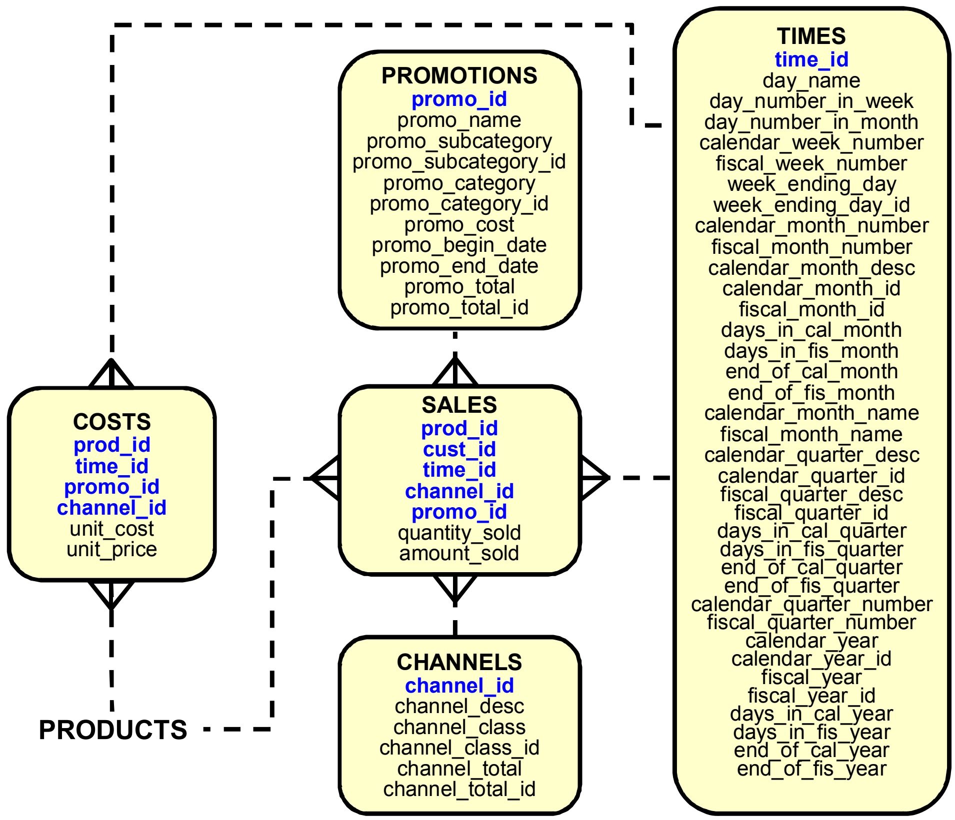
In the `Human Resource (HR)` records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn commissions in addition to their salaries.

The company also tracks information about jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee resigns, the duration for which the employee worked, the job identification number, and the department are recorded.

Because the sample company is regionally diverse, it tracks the locations of its warehouses and departments. Each employee is assigned to a department, and each department is identified by a unique department number or by a short name. Each department is associated with one location, and each location has a full address that includes the street name, postal code, city, state or province, and the country code.

In places where the departments and warehouses are located, the company records such details as the country name, currency symbol, currency name, and the region where the country is located geographically.

Sales History (SH) Schema

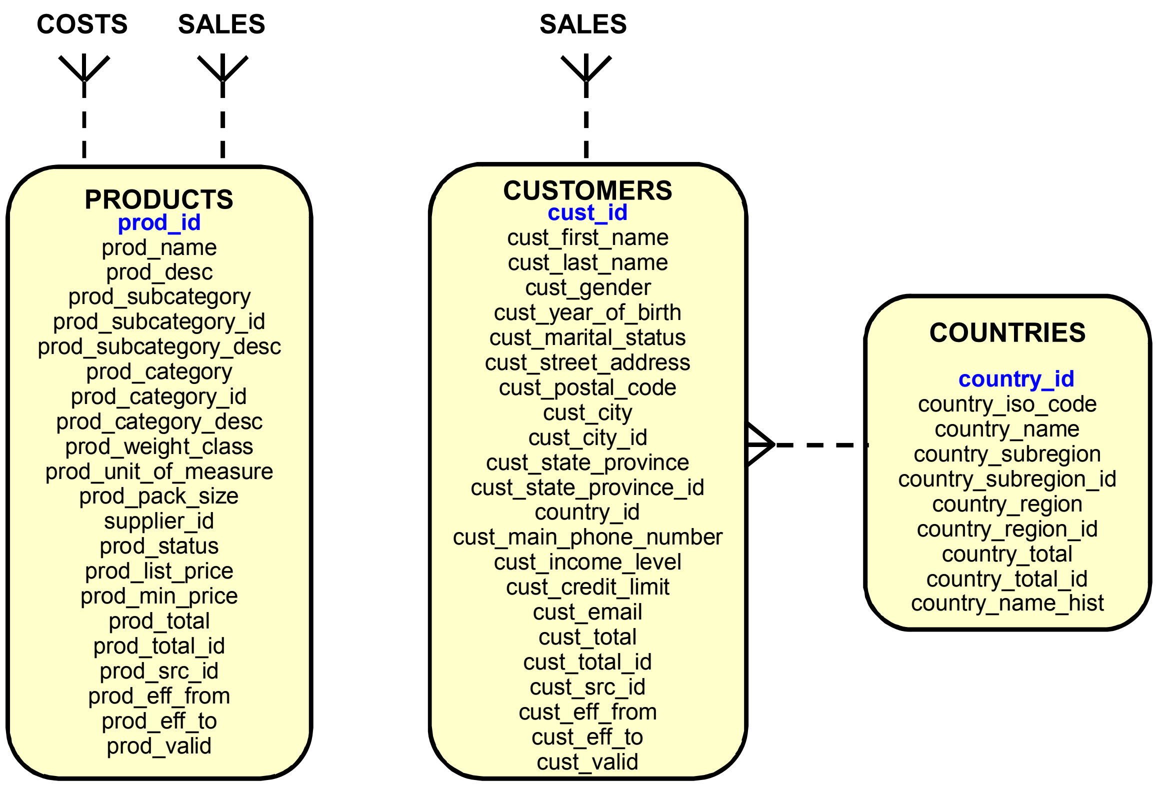


ORACLE

The company does a high volume of business and, therefore, runs business statistics reports to support decision making. Many reports are time-based and nonvolatile; that is, they analyze past data trends. The company regularly loads data into its data warehouse to gather statistics for the reports. The reports include annual, quarterly, monthly, and weekly sales figures by product.

The company also runs reports on the distribution channels through which its sales are delivered. When the company runs special promotions on its products, it analyzes the impact of the promotions on sales. It also analyzes sales by geographical area.

Sales History (SH) Schema

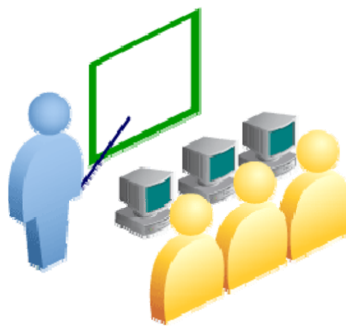


Lesson Agenda

- Course Objectives, Course Outline, Prerequisites, and Activities Used in the Course
- Sample Schemas Used in the Course
- Class Account Information
- SQL Environments Available in the Course
- Overview of the Workshops, Demos, Code Examples, Solution Scripts, and Appendixes Used in the Course

Class Account Information

- Your account IDs are `sh`, `hr`, `scott`, and `oe`.
- The password matches your account ID.
- Each machine has a stand-alone installation of Oracle Database 12*c* Enterprise Edition for Linux, with access to the `SH`, `OE`, `SCOTT`, and `HR` sample schemas.



Lesson Agenda

- Course Objectives, Course Outline, Prerequisites, and Activities Used in the Course
- Sample Schemas Used in the Course
- Class Account Information
- SQL Environments Available in the Course
- Overview of the Workshops, Demos, Code Examples, Solution Scripts, and Appendixes Used in the Course

SQL Environments Available in the Course

This course setup provides the following tools for you to execute SQL statements:

- Oracle SQL*Plus
- Oracle SQL Developer
- Oracle Enterprise Manager Cloud Control (EMCC)

Oracle provides several tools that can be used to execute SQL statements. Here are some of the tools that are available for use in this course:

- **Oracle SQL*Plus:** A window or command-line application
- **Oracle SQL Developer:** A free graphical tool
- **Oracle Enterprise Manager Cloud Control (EMCC):** A web-based tool for managing your Oracle database (installed with the database)

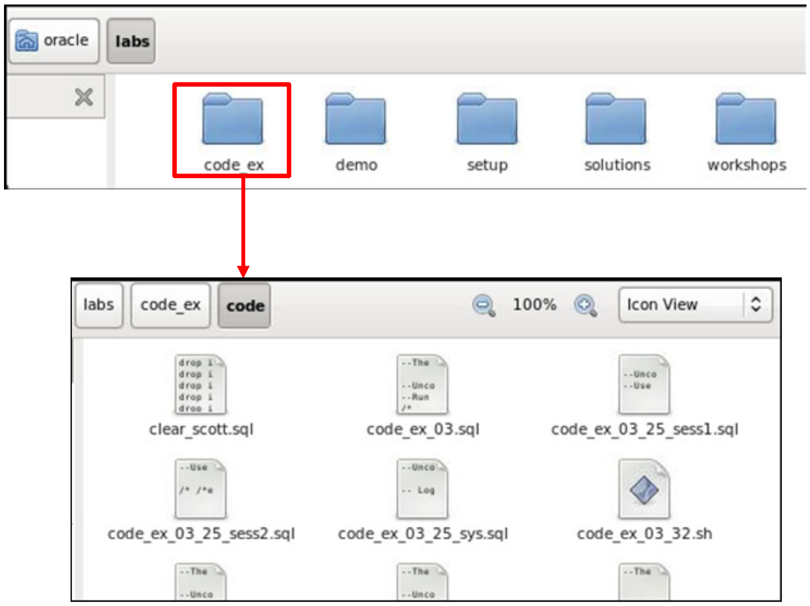
Note: The code and screen examples presented in the course are from output in the Oracle SQL Developer environment.

Lesson Agenda

- Course Objectives, Course Outline, Prerequisites, and Activities Used in the Course
- Sample Schemas Used in the Course
- Class Account Information
- SQL Environments Available in the Course
- Overview of the Workshops, Demos, Code Examples, Solution Scripts, and Appendixes Used in the Course

Workshops, Demo Scripts, Code Examples, and Solution Scripts

Workshops, demos, code examples, and solution scripts are available to you.



All complete lesson-wise code examples are provided in the `code_ex` folder, as displayed in the slide. The files in this folder are named after the page on which the code is found. For example:

- `/home/oracle/labs/code_ex/code/code_ex_03.sql` holds the code example shown in the slide portion in the lesson titled "Using Application Tracing Tools."

Lesson-wise solution scripts are provided in the `solutions` folder. For example, the `Application_Tracing` folder has the solution scripts for Practice 3.

- `/home/oracle/labs/solutions/Application_Tracing` holds the solution scripts for the lesson titled "Using Application Tracing Tools."

The solution files, workshop files, demos, and code_ex files are located in `/home/oracle/labs`.

Note: You should save all script files in the `labs` folder.

Appendixes in the Course

- Appendix A: Using SQL Developer
- Appendix B: SQL Tuning Advisor
- Appendix C: SQL Access Advisor
- Appendix D: Oracle Database Architecture
- Appendix E: Real-Time Database Monitor
- Appendix F: Gathering and Managing Optimizer Statistics

Additional Resources

For additional information about Oracle Database 12*c* and Oracle Database 12*c*: SQL Tuning for Developers, refer to the following:

- *Oracle Learning Library:*
  - http://apex.oracle.com/pls/apex/f?p=44785:1:0::NO
- *Oracle Cloud:*
  - https://cloud.oracle.com/mycloud/f?p=service:home:0
- *Optimizer Blog:*
  - *https://blogs.oracle.com/optimizer/entry/how_does_sql_plan_management*
- *OTN: http://www.oracle.com/technetwork/index.html*

2

Introduction to SQL Tuning

Objectives

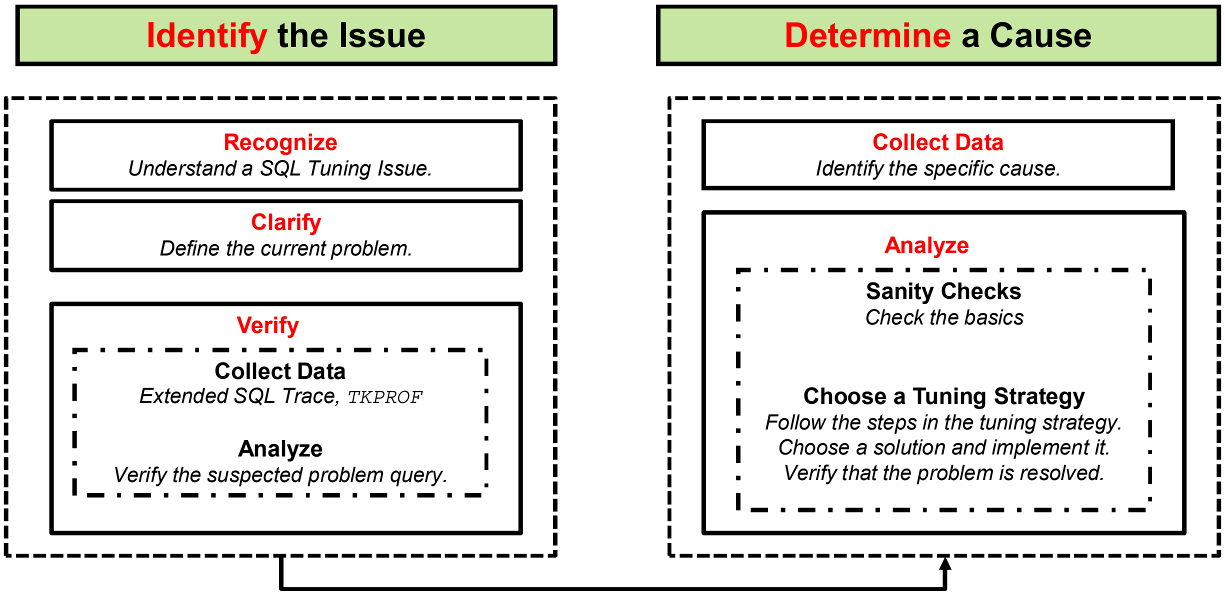
After completing this lesson, you should be able to:

- Describe SQL tuning
- Explain SQL tuning strategies
- Describe Oracle SQL Developer

Lesson Agenda

- SQL Tuning Session
  - Recognize: What Is Bad SQL?
  - Clarify: Understand the Current Issue
  - Verify: Collect Data
  - Verify: Is the Bad SQL a Real Problem? (Top-Down Analysis)
- SQL Tuning Strategies
  - Sanity Checks
  - Advanced SQL Tuning Analysis
  - Parse Time Reduction
  - Plan Comparison
  - Quick Solution
  - Query Analysis
- Development Environments: Overview

SQL Tuning Session



Oracle Corporation has developed a tuning methodology based on years of experience. This lesson focuses on two subject areas involved in tuning SQL: the general SQL tuning session and the possible SQL tuning strategies.

To identify the issue properly and to determine a cause of the problem, you must be able to:

- Recognize a SQL tuning issue
- Clarify the details of the issue
- Verify that the issue is the problem
- Check the basics (after the SQL problem is verified)
- Choose an appropriate tuning strategy

Note: The methodology presented in this lesson is also presented in the *Oracle Database Performance Tuning Guide* and the *Oracle Performance Diagnostic Guide* (MOS note 390374.1).

Recognize: What Is Bad SQL?

- Bad SQL uses more resources than necessary.
- Bad SQL has the following characteristics:
  - Excessive parse time
  - Excessive I/O (physical reads and writes)
  - Excessive CPU time
  - Excessive waits

One of the benefits of SQL is that you can write different SQL statements that produce the same result. Any SQL statement that produces the correct result is a correct SQL statement. However, different SQL can require different amounts of resources. Bad SQL can be correct, but bad SQL is inefficient, because it uses more resources than necessary.

The symptoms of bad SQL can be any of the characteristics listed in the slide.

Bad SQL can result from a bad design, poor coding, or an inefficient execution plan. You can control the design or code and influence the optimizer to produce a better execution plan.

Conceptually, there is an optimum execution plan for any given result set from a given set of relational data. Based on time and resources, the optimizer attempts to find this optimum execution plan, which may take a long time. For example, you would not be willing to wait five minutes for the optimizer to produce a plan that reduces the run time by five seconds. The order in which the trial execution plans are evaluated by the optimizer is influenced by many factors, including how the SQL is written.

Clarify: Understand the Current Issue

Changes that might trigger the issue:

- Database upgraded
- Statistics gathered
- Schema changed
- Database parameter changed
- Application changed
- Operating system (OS) and hardware changed
- Data volume changed by more active users

In this step, you need to understand the problem, because a clear problem statement is critical to finding the cause of and solution to the problem. Skipping this step is risky because you might address the wrong problem and waste time and effort. Later on, the real problem may become clearer and you may have to revisit and reclarify the issue.

Note: For more details about SQL tuning issues, see MOS master note 199083.1, *SQL Query Performance Overview*.

Verify: Collect Data

These tools are available to identify bad SQL statements:

- Client Reports
- OS Statistics
- SQL Trace Facility
- Trace Analyzer (`TRCANLZR`)
- `SQLTXPLAIN` (SQLT)
- SQL Performance Analyzer (SPA)
- Automatic Workload Repository (AWR) Report
- Active Session History (ASH) Report
- Top SQL Report
- Automatic Database Diagnostic Monitor (ADDM)

In this step, you gather data to verify that the performance problem originates in the database.

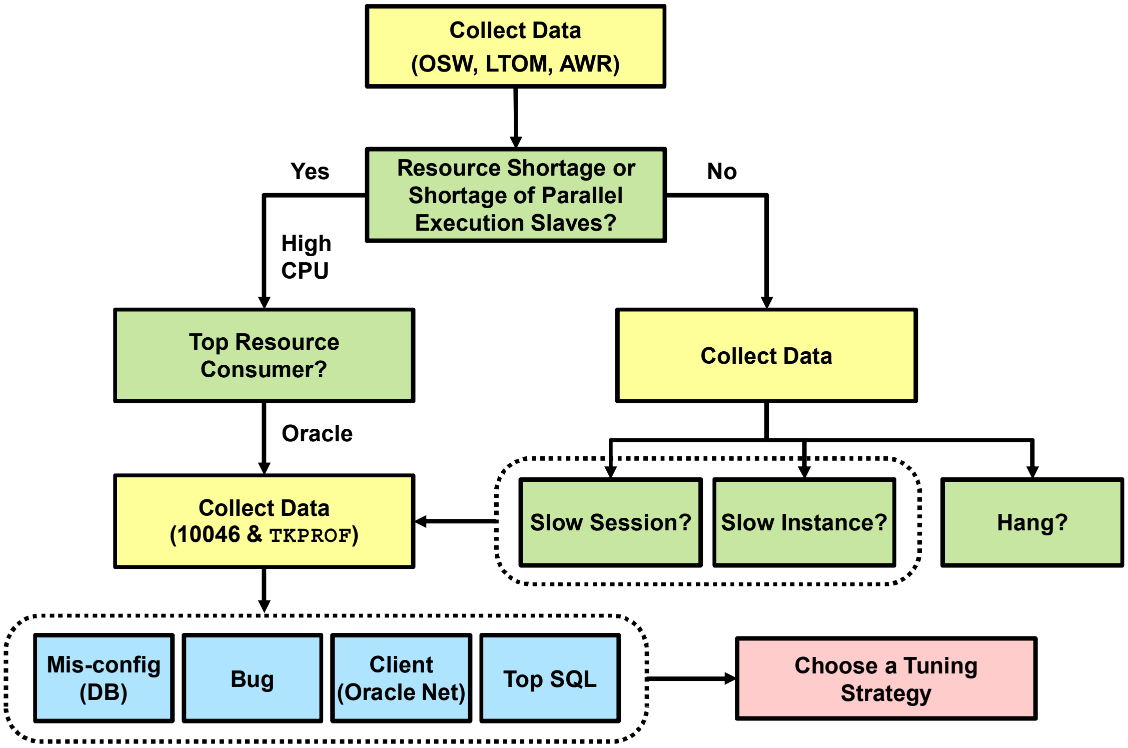
The following tools are available to identify bad SQL statements:

- **OS Statistics:** This tool checks the OS statistics and general machine health before tuning the instance to verify that the problem is in the database instance. For more information, see MOS note 369412.1, *Case Study: Resolving High CPU Usage on Oracle Servers*.
- **SQL Trace Facility:** This basic performance diagnostic tool helps monitor and tune applications running against the Oracle database. It enables you to assess the efficiency of the SQL statements that an application runs, and generates statistics for each statement. The trace files serve as input for `TKPROF`. For more information, see MOS note 376442.1, *Recommended Method for Obtaining 10046 Trace for Investigating SQL Query Performance* and MOS note 39817.1, *Interpreting Raw SQL Trace*.
- **Trace Analyzer:** This tool inputs an event 10046 SQL trace file, connects to the database, and generates a comprehensive report for process performance analysis and tuning. For more information, see MOS note 224270.1, *Trace Analyzer `TRCANLZR`*.

- **`SQLTXPLAIN` (SQLT):** This tool helps diagnose SQL statements that are performing poorly. It is often requested by Oracle Support. For more information, see MOS note 215187.1, *SQLT (`SQLTXPLAIN`)*.
- **SQL Performance Analyzer (SPA):** This tool automates the process of assessing the overall effect of a change—such as upgrading a database or adding new indexes—on the full SQL workload by identifying performance divergence for each statement.
- **Automatic Workload Repository (AWR) Report:** AWR reports include a set of Top SQL listings. Each report lists the top SQL statements sorted by resource usage in the following categories: Elapsed Time, CPU Time, Gets, Reads, Executions, Parse Calls, Sharable Memory, and Version Count. The individual reports do not include the full SQL text, but only a report of all SQL text by `SQL_ID`.
- **Active Session History (ASH) Report:** ASH reports include a set of top SQL statements that are associated with the top events, SQL statements that are associated with the top row sources, top SQL using literals, and the SQL text for these SQL statements.
- **Top SQL Report:** The Top SQL reports are very useful for identifying the statements that consume the most resources on your system. Studies have shown that, typically, 20% of the SQL statements consume 80% of the resources, and 10% of the statements consume 50% of the resources. This means that you can improve the performance of the entire system by identifying and tuning the top SQL statements.
- **Automatic Database Diagnostic Monitor (ADDM):** This tool continually analyzes the performance data that is collected from the database instance. Review *ADDM Findings - SQL as a Main Issue*. For more information, see MOS note 250655.1, *How to Use the ADDM*.
- **Automatic SQL Tuning:** Oracle Database 11*g* automates the SQL tuning process by identifying problematic SQL statements, running SQL Tuning Advisor (STA), and implementing the resulting SQL profile recommendations to tune the statement without requiring user intervention. Automatic SQL Tuning uses the `AUTOTASK` framework through a task called “Automatic SQL Tuning” that runs every night by default.

Note: In many cases, some tools are not available to the SQL developer due to lack of privileges. The DBA usually makes the first attempt to identify bad SQL statements and collect data when performance is good as well as when it is bad.

Verify: Is the Bad SQL a Real Problem? (Top-Down Analysis)



Does the performance issue originate in the OS, the instance, or the application SQL?

This question is not always easy to answer. Poorly performing SQL can show high CPU usage and appear to be a CPU issue. An improperly configured instance can lead to high CPU utilization in the OS.

To eliminate possibilities, examine the collected data.

1. Are CPU resources scarce? Check CPU usage by looking for the following:
   - Total CPU utilization (USER + SYS) should be less than 90%.
   - Run queue size per CPU should be less than 4.
2. What processes use most of the CPU? Examine the collected data to find out what kind of process uses most of the CPU. If most of the CPU is used by this instance, you have verified that Oracle processes are responsible for the CPU consumption.

3. Collect a 10046 trace and review the collected data to identify the problem SQLs.
   - Studies have shown that, typically, 20% of the SQL statements consume 80% of the resources, and 10% of the statements consume 50% of the resources.
   - By identifying and tuning the top SQL statements, you can improve the performance for the entire system.

Determine the scope of the problem to focus your efforts on the solutions that provide the most benefit.

For more details, see MOS note 369412.1, *Case Study: Resolving High Usage on Oracle Servers*.

Lesson Agenda

- SQL Tuning Session
  - Recognize: What Is Bad SQL?
  - Clarify: Understand the Current Issue
  - Verify: Collect Data
  - Verify: Is the Bad SQL a Real Problem? (Top-Down Analysis)
- SQL Tuning Strategies
  - Sanity Checks
  - Advanced SQL Tuning Analysis
  - Parse Time Reduction
  - Plan Comparison
  - Quick Solution
  - Query Analysis
- Development Environments: Overview

SQL Tuning Strategies: Overview

- Sanity checks
- Advanced SQL tuning analysis
- Parse time reduction
- Plan comparison
- Quick solution
- Query analysis

After identifying the SQL problem, choose an efficient way to tune the SQL statement. Your choice of tuning strategy depends on the type of performance problem.

Answer the following questions to determine your tuning strategy:

- Do you know what a good plan is?
- Do you want to solve this problem quickly or do you want to determine the cause?
- Does the query spend most of its time in the parse phase?
- Do you have the Tuning Pack option?

Checking the Basics

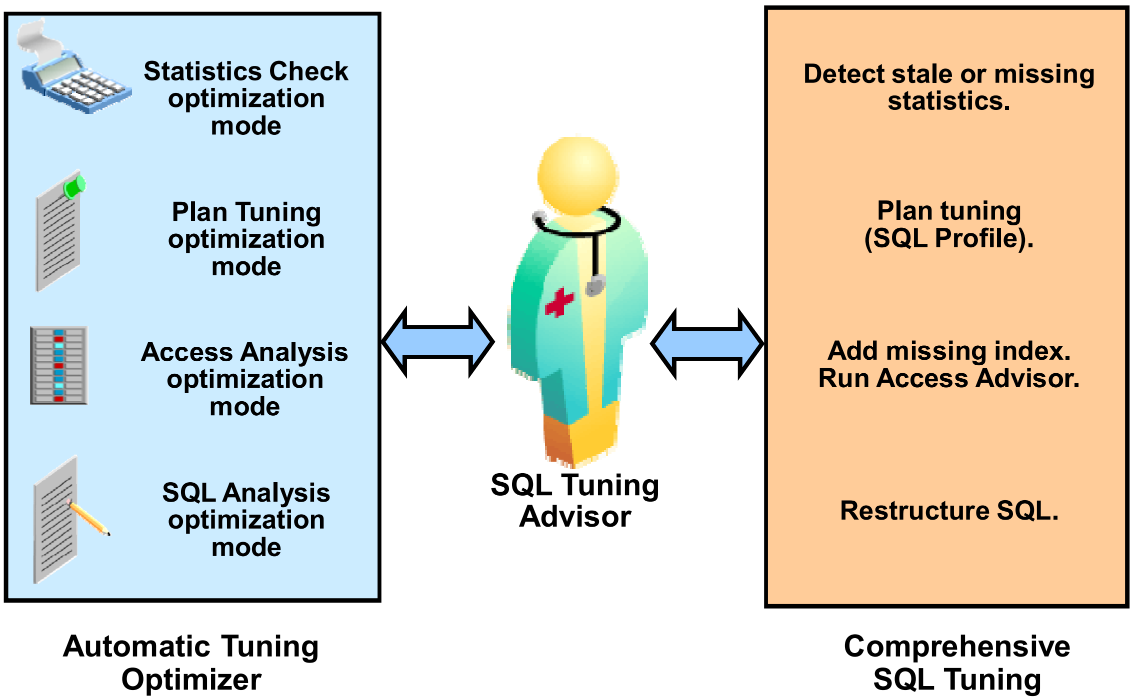
Always check the basics first. Ensure that:

- Up-to-date statistics are collected properly
- Reasonable initialization parameters are set
- The proper optimizer mode is set
- Appropriate and valid hints are used

Always start this process by checking the statistics, optimizer mode, and important initialization parameters, and then choose a tuning strategy that matches your problem and objectives.

- **Ensure that fresh and accurate table / index statistics exist:** Accurate statistics in all query tables and indexes are essential so that the optimizer produces good execution plans.
- **Ensure that reasonable initialization parameters are set:** The optimizer uses the values of the initialization parameters to estimate the cost of various operations in the execution plan. When certain parameters are improperly set, the cost estimates will be inaccurate and will create suboptimal plans.
- **Ensure that the proper optimizer mode is set:** The use of the optimizer is essential for this tuning effort because the rule-based optimizer (RBO) is no longer supported.
- **Ensure that appropriate and valid hints are used:** When hints are used, the execution plans tend to be much less flexible and big changes to the data volume or distribution may lead to suboptimal plans.

Advanced SQL Tuning Analysis



Oracle Database is able to perform advanced SQL tuning analysis by using the STA (and related Access Advisor). This approach is the preferred way to begin a tuning effort if you are using Oracle Database.

STA is primarily the driver of the tuning process. It calls the Automatic Tuning Optimizer (ATO) to perform the following types of analyses:

- Statistics Analysis
- SQL Profiling
- Access Path Analysis
- SQL Structure Analysis

Note: You must have a Tuning Pack option in order to use these features.

Parse Time Reduction Strategy

- Example 1: Inefficient SQL statements

```
SELECT * FROM ….

call     count       cpu    elapsed    disk     query   current     rows
------- ------  --------  --------  ------  --------  --------  -------
Parse      555    100.09    300.83       0         0         0        0
Execute    555      0.42      0.78       0         0         0        0
Fetch      555     14.04     85.03     513   1448514         0    11724
------- ------  --------  --------  ------  --------  --------  -------
total     1665    114.55    386.65     513   1448514         0    11724
```

- Example 2: Cursor sharing
- Example 3: Connection management

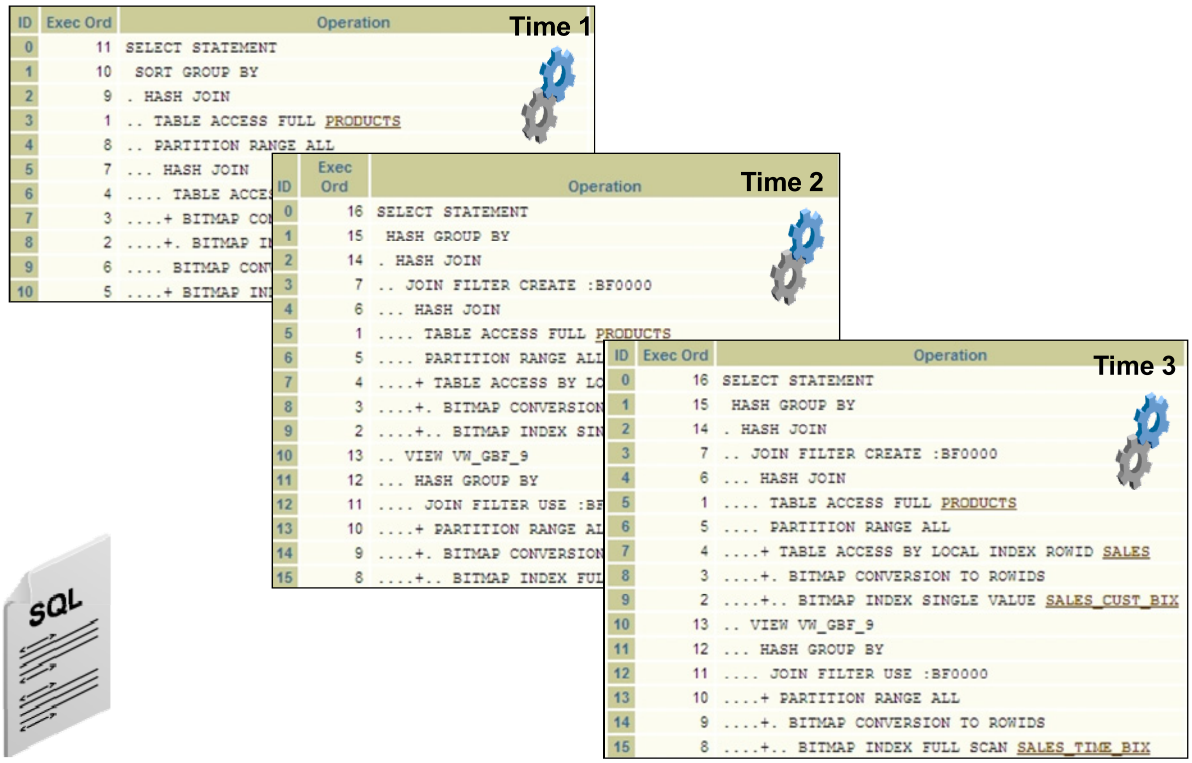
If the query spends most of its time parsing, typical query-tuning techniques that alter the execution plan to reduce logical I/O during execute or fetch calls probably will not help. The focus of the tuning should be to reduce parse times.

If the parse time spent on CPU is more than 50% of the parse elapsed time, the parse time is dominated by the CPU; otherwise, it is dominated by waits.

The following is the strategy for parse time reduction:

- Use dynamic statistics for the queries.
- Use many `IN-LIST/OR` parameters for queries.
- Have partitioned table with many partitions (more than 1000).
- Ensure that waits for large query text are sent from the client.

Plan Comparison Strategy



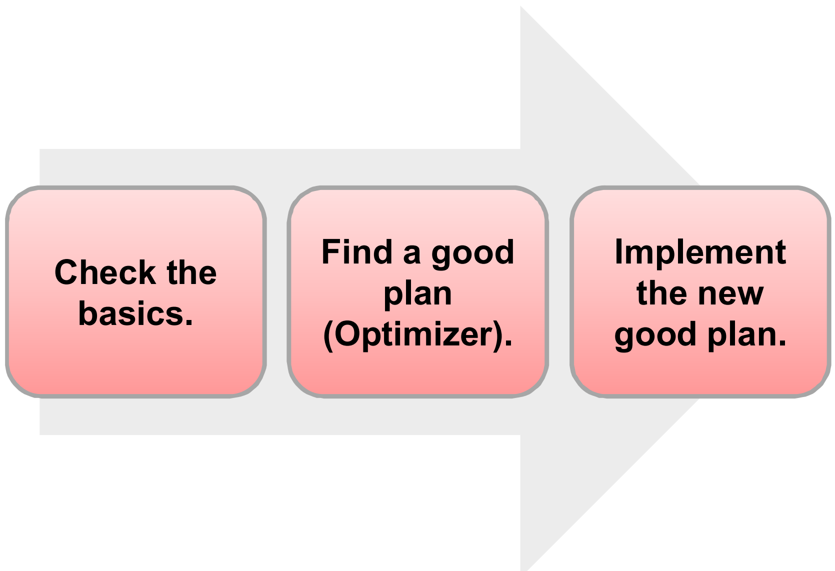
ID	Exec Ord	Operation
0	16	SELECT STATEMENT
1	15	HASH GROUP BY
2	14	. HASH JOIN
3	7	.. JOIN FILTER CREATE :BF0000
4	6	... HASH JOIN
5	1	.... TABLE ACCESS FULL PRODUCTS
6	5	.... PARTITION RANGE ALL
7	4	....+ TABLE ACCESS BY LOCAL INDEX ROWID SALES
8	3	....+. BITMAP CONVERSION TO ROWIDS
9	2	....+.. BITMAP INDEX SINGLE VALUE SALES_CUST_BIX
10	13	.. VIEW VW_GBF_9
11	12	... HASH GROUP BY
12	11	.... JOIN FILTER USE :BF0000
13	10	....+ PARTITION RANGE ALL
14	9	....+. BITMAP CONVERSION TO ROWIDS
15	8	....+.. BITMAP INDEX FULL SCAN SALES_TIME_BIX

If you have a “good” execution plan, you can use the “Execution Plan Comparison” strategy to find where the good and bad plans differ. That way, you can modify the query to produce a good plan or you can focus on the particular place where they differ and determine the cause for the difference.

The following is the strategy for plan comparison:

- Compare the good and bad plans.
- Find the differences.
- Fix the query by finding ways to make a bad plan look like the good one.
- See “Quick Solution Strategy” in this lesson for more suggestions.

Quick Solution Strategy



If you need to make the query perform well immediately and you are not interested in the underlying cause of the problem, you can use the quick solution strategy to give the optimizer more information and possibly obtain a better execution plan.

The goal of this approach is to change some high-level optimizer settings to create a better plan (for example, change optimizer mode or use dynamic statistics). Then, use hints or other means to make the optimizer generate the better plan. You can use the better plan to find an underlying cause later.

The following is the strategy for a quick solution:

- Check the basics.
- Find a good plan.
- Implement the new good plan.

Finding a Good Plan

- Use SQL Tuning Advisor (if you can).
- Get a good test case.
- Leverage the optimizer by changing settings to obtain the better plan (SESSION only):
  - `OPTIMIZER_MODE (FIRST_ROWS_N, ALL_ROWS)`
  - `OPTIMIZER_FEATURES_ENABLE`
  - `OPTIMIZER_INDEX_COST_ADJ`
  - `OPTIMIZER_INDEX_CACHING`
- Try dynamic statistics at high levels.
- Try appropriate hints.

Try the following changes first because they are likely to give the best results in the shortest time:

- **`OPTIMIZER` mode:** If the optimizer mode is currently `ALL_ROWS`, use `FIRST_ROWS_N` (choose a value for N that reflects the number of rows that the user wants to see right away) or vice versa.
- **`OPTIMIZER_FEATURES_ENABLE` parameter:** If a particular query performed better in an older version (for example, before a migration), use this parameter to "roll back" the optimizer to the older version. In Oracle Database 10*g* and later, you can set this parameter at the session level, which is preferred over the system-wide level.
- **Dynamic statistics:** This approach samples the number of rows returned by the query and determines very accurate selectivity estimates that often lead to good execution plans.

Implementing the New Good Plan

- Accept a profile from STA.
- Use a SQLT profile that was generated from a good plan.
- Use SQL Plan Management.

Use the techniques listed in the slide to implement the new good plan. The technique depends on the modifiability of the query or the application.

Query Analysis Strategy: Overview

- Use this strategy when:
  - A good plan is not available
  - An urgent solution is not required
  - You want to determine an underlying cause
  - The query may be modified
- This strategy focuses on:
  - Statistics and parameters
  - SQL statement structure
  - Data access paths
  - Join orders and join methods
  - Other operations, such as parallelism or partition pruning

The default strategy for query-tuning issues is to discover common problems in the query text, access paths, join orders, and join methods, and then implement solutions for the discovered problems.

The following is the strategy for query analysis:

- **Statistics and parameters:** Ensure that statistics are up-to-date and parameters are reasonable.
- **SQL statement structure:** Look for constructs known to confuse the optimizer.
- **Data access paths:** Look for inefficient ways of accessing data.
- **Join orders and join methods:** Look for join orders where large row sources are at the beginning; look for inappropriate use of join types.
- **Other operations:** Look for unexpected operations, such as parallelism or lack of partition elimination.

Query Analysis Strategy: Collecting Data

- Collect data for the query:
  - Execution plans
  - Information about each table or view
  - Object statistics and system statistics
  - Histograms
  - Parameter settings
- Tools available to collect data:
  - SQLT
  - DBMS_STATS
  - Extended SQL trace and TKPROF
  - SPREPSQL.SQL
  - AWRSQRPT.SQL

Query Analysis Strategy: Examining SQL Statements

Look at the query for common mistakes:

- Understand the volume of resulting data.
- Ensure that no join predicates are missing.
- Look for unusual predicates.
- Be careful for constructs known to cause problems such as large `IN` lists / `OR` statements, outer joins, hierarchy queries, views, inline views, and subqueries.

Query Analysis Strategy: Analyzing Execution Plans

Analyze the execution plan to find areas that need to be tuned:

- Examine data access paths, such as full table scan, index full scan, and index fast full scan.
- Examine join order and join types, such as nested loops join, hash join, and sort-merge join.
- Review the actual number of rows and the estimated number of rows that are returned by the query.
- Look for plan steps where there is a large discrepancy between the actual and estimated rows.
- Look for plan steps where cost and logical read differ significantly.

The choice of access path greatly affects the performance of queries. If the query has a predicate that reduces the number of rows in a table, an index is usually beneficial (indexes should exist for the columns in the predicate).

However, if there is no predicate to filter the rows from a table or if the predicate is not very selective and many rows are expected, a full table scan may be a better choice than an index scan. It is important to understand the actual number of rows expected for each plan step and compare it to the optimizer estimate so that you can determine if a full table scan or index access makes more sense.

The join order can have a huge impact on query performance. The optimal join order is one where the fewest rows are returned earliest in the plan. The optimizer tries to start join orders with tables that it believes will return only one row. If this estimate is wrong, the wrong table may be chosen and the performance of the query may be affected.

The choice of join type is also important. Nested loops joins are desirable when only a few rows are needed quickly and join columns are indexed. Hash joins are typically very good at joining large tables, returning many rows, or joining columns that lack indexes.

Query Analysis Strategy: Finding Execution Plans

Use the following tools to find execution plans:

- `V$SQL_PLAN` (library cache)
- `V$SQL_PLAN_MONITOR`
- `DBA_HIST_SQL_PLAN` (AWR)
- SQL management base (SQL plan baselines)
- SQL tuning set
- Trace files generated by `DBMS_MONITOR`
- Event 10053 trace file
- `SQLTXPLAIN` report
- SQL trace
- Extended SQL trace and `TKPROF`
- `SPREPSQL.SQL`
- `AWRSQRPT.SQL`

You can use the tools listed in the slide to find the execution plans. You can generate 10053 trace file with `DBMS_SQLDIAG.DUMP_TRACE`.

Query Analysis Strategy: Reviewing Common Observations and Causes

Review common observations and causes:

- Poorly written SQL
- Index used / not used
- Lack of an index
- Wrong join order
- Wrong type
- Predicates not pushed, views not merged
- Transformation improperly costed
- Other problems

Query Analysis Strategy: Determining Solutions

Consider possible solutions:

- Gather statistics properly.
- Create a new index or re-create an existing index.
- Use the SQL Advisors with the Tuning Pack option.
- Use hints to get the preferred plan.
- Use SQL Plan Management.
- Use dynamic statistics to obtain accurate selectivity estimates.
- Eliminate implicit data type conversions.
- Create a function-based index.
- Rewrite the query to permit the use of an existing index.

Consider some more possible solutions:

- Use an Index-Organized Table (IOT).
- Remove hints that influence the choice of index.
- Correct common problems with hints.
- Use the correct optimizer mode.
- Add the appropriate join predicate for the query.
- Review the intent of the query and ensure that a predicate is not missing.
- Use parallel execution / parallel data manipulation language (DML).
- Ensure that array processing is used.
- Use materialized views and query rewrite.
- Load the data in the key order.

Lesson Agenda

- SQL Tuning Session
  - Recognize: What Is Bad SQL?
  - Clarify: Understand the Current Issue
  - Verify: Collect Data
  - Verify: Is the Bad SQL a Real Problem? (Top-Down Analysis)
- SQL Tuning Strategies
  - Sanity Checks
  - Advanced SQL Tuning Analysis
  - Parse Time Reduction
  - Plan Comparison
  - Quick Solution
  - Query Analysis
- Development Environments: Overview

Development Environments: Overview

- Oracle SQL Developer
- Oracle SQL*Plus

SQL Developer

SQL*Plus

Oracle SQL Developer is shipped with Oracle Database 12*c* Release 1, and it is the default tool for the examples that are used in this class.

You can also use the Oracle SQL*Plus environment to run all SQL commands covered in this course.

What Is Oracle SQL Developer?

- Oracle SQL Developer is a free graphical tool that improves productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.
- You use Oracle SQL Developer in this course.
- Appendix A contains details for using Oracle SQL Developer.

SQL Developer

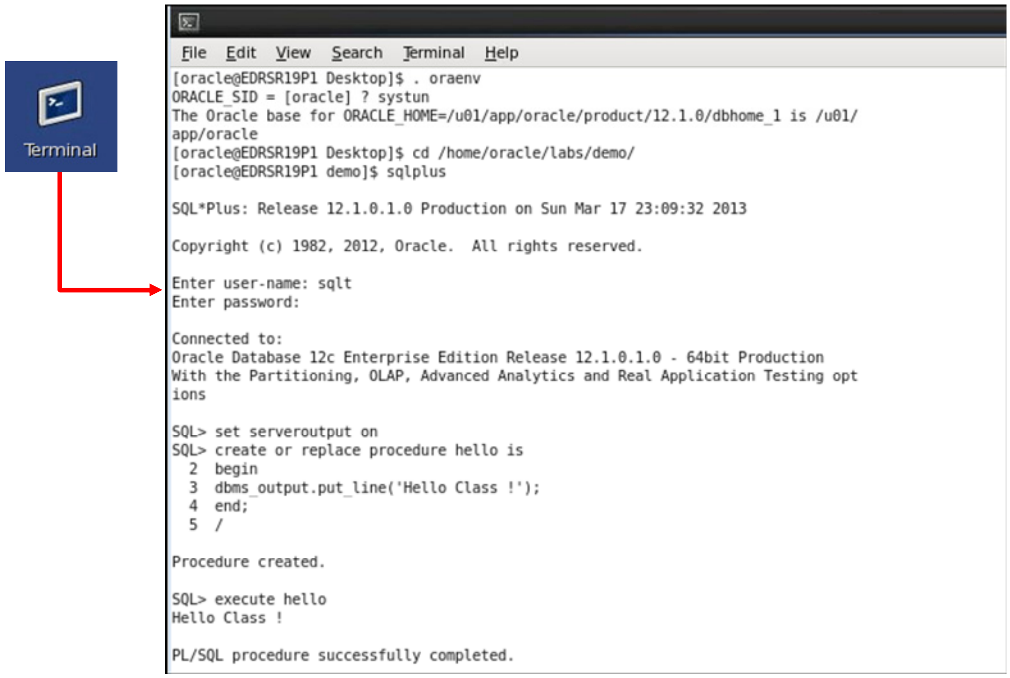
Oracle SQL Developer is a free graphical tool that is designed to improve your productivity and simplify the development of everyday database tasks. You can easily create and maintain stored procedures, test SQL statements, and view optimizer plans.

Oracle SQL Developer simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema by using standard Oracle database authentication. When connected, you can perform operations on objects in the database.

Coding PL/SQL in Oracle SQL*Plus



Oracle SQL*Plus is a command-line interface that enables you to submit SQL statements and PL/SQL blocks for execution and receive the results in an application or a command window. It is shipped with the database, and it is accessed by clicking an icon or from the command line.

When coding PL/SQL subprograms by using Oracle SQL*Plus, remember the following:

- You create subprograms by using the `CREATE SQL` statement.
- You execute subprograms by using either an anonymous PL/SQL block or the `EXECUTE` command.
- If you use the `DBMS_OUTPUT` package procedures to print text to the screen, you must first execute the `SET SERVEROUTPUT ON` command in your session.

Note

- To launch Oracle SQL*Plus in the Linux environment, open a terminal window and enter the `sqlplus` command.
- For more information about how to use Oracle SQL*Plus, see the *Oracle SQL*Plus User's Guide and Reference*.

Quiz

Although views provide clean programming interfaces, they should be used carefully because they can cause suboptimal, resource-intensive queries when nested too deeply.

a. True

b. False

Answer: a

Quiz

Identify the characteristics that must be supported by an application that is designed for SQL execution efficiency.

a. Use concurrent connections to the database.
b. Use cursors so that SQL statements are parsed once and executed multiple times.
c. For data-warehousing queries, cursor sharing is always important to get the best plan.

Answer: b

Summary

In this lesson, you should have learned how to:

- Describe SQL tuning
- Explain SQL tuning strategies
- Describe Oracle SQL Developer

Practice 2: Overview

This practice covers using SQL Developer.

3

Using Application Tracing Tools

Objectives

After completing this lesson, you should be able to:

- Discuss the steps needed before tracing
- Describe application tracing tools
- Use the SQL Tracing facility
- Perform end-to-end application tracing
- Consolidate SQL trace files by using the `trcsess` utility
- Format trace files by using the `TKPROF` utility
- Interpret the output of the `TKPROF` command
- Verify a SQL problem by using a `TKPROF` report

Lesson Agenda

- Using Application Tracing
- Application Tracing Tools
- Using the SQL Trace Facility
- End-to-End Application Tracing
- `trcsess` Utility
  - Invoking the `trcsess` Utility
  - `trcsess` Utility: Example
- `TKPROF` Utility
  - Invoking the `TKPROF` Utility
  - `TKPROF` Sorting Operations
  - Interpreting a `TKPROF` Report: Examples

Using Application Tracing: Overview

Before Tracing	• Determine the trace file location. • Choose the most impacted sessions and others.
Enable Tracing	• SQL Tracing • End-to-End Application Tracing
Workload	• Generate User Workload.
Disable Tracing	• Disable Tracing.
`trcsess`	• Consolidates trace output from selected trace files. • trcsess output=<output file> <criteria> <trace files>
`TKPROF`	• Format the contents of the trace file and place the output into a readable output file.
Review	• Interpret the `TKPROF` output.

In a production environment, you might perform the tasks listed in the slide to trace a suspected session working with DBAs. In addition, if developers can leverage application tracing during the development and testing phase, many of the issue could be solved before deployment.

You should enable application tracing only when you tune SQL statements, and disable it when you are finished because running application tracing increases system overhead. Oracle recommends that you use the `DBMS_SESSION` or `DBMS_MONITOR` packages to enable SQL tracing for a session or an instance. You may need to modify an application to contain the `ALTER SESSION` statement.

For example, to issue the `ALTER SESSION` statement in Oracle Forms, invoke Oracle Forms by using the `-s` option, or invoke Oracle Forms (Design) by using the statistics option.

You have to set the trace option in an `AFTER LOGON` trigger, such as:

```
CREATE OR REPLACE TRIGGER logon_trig
 AFTER LOGON  ON  hr.SCHEMA
 BEGIN
 EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE=TRUE';
 END;
 /
```

For more information about Oracle Forms, see the *Oracle Forms Reference*.

Steps Needed Before Tracing

- Determine the location for diagnostic traces.
- Choose the most important affected sessions:
  - Find sessions with the highest CPU consumption.
  - Find sessions with the highest waits of a certain type.
  - Find sessions with the highest DB time.
- Choose the most important affected clients, services, modules, actions, users, or sessions through Enterprise Manager (if possible) or through user feedback.

Follow these steps before enabling application tracing:

- Determine the location for diagnostic traces.

  In Oracle Database, Automatic Diagnostic Repository (ADR) is a file-based repository for database diagnostic data, such as traces, incident dumps, packages, the alert log, Health Monitor reports, and core dumps. The trace location is set by the `DIAGNOSTIC_DEST` initialization parameter.
- Choose the most important affected clients, services, modules, actions, users, or sessions; for example, users that are experiencing the problem most severely (the transaction is typically completed in one second, but now it takes 30 seconds) or users that are aggressively accumulating time in the database.

Examples:

- Find sessions with the highest CPU consumption.

```
      SELECT s.sid, s.serial#, p.spid as "OS PID",s.username, s.module,
st.value/100 as "CPU sec"
      FROM v$sesstat st, v$statname sn, v$session s, v$process p
      WHERE sn.name = 'CPU used by this session'
      AND st.statistic# = sn.statistic#
      AND st.sid = s.sid
      AND s.paddr = p.addr
      AND s.last_call_et < 1800
      AND s.logon_time > (SYSDATE - 240/1440)
      ORDER BY st.value;
```

- Find sessions with the highest waits of a certain type.

```
SELECT s.sid, s.serial#, p.spid as "OS PID", s.username, s.module,
se.time_waited
FROM v$session_event se, v$session s, v$process p
WHERE se.event = '&event_name'
AND s.last_call_et < 1800
AND s.logon_time > (SYSDATE - 240/1440)
AND se.sid = s.sid
AND s.paddr = p.addr
ORDER BY se.time_waited;
```

- Find sessions with the highest DB time

```
SELECT s.sid, s.serial#, p.spid as "OS PID", s.username, s.module,
st.value/100 as "DB Time (sec)", stcpu.value/100 as "CPU Time (sec)",
round(stcpu.value / st.value * 100,2) as "%CPU"
FROM v$sesstat st, v$statname sn, v$session s, v$sesstat stcpu,
v$statname sncpu, v$process p
WHERE sn.name = 'DB time'
AND st.statistic# = sn.statistic#
AND st.sid = s.sid
AND sncpu.name = 'CPU used by this session'
AND stcpu.statistic# = sncpu.statistic#
AND stcpu.sid = st.sid
AND s.paddr = p.addr
AND s.last_call_et < 1800
AND s.logon_time > (SYSDATE - 240/1440)
AND st.value > 0;
```

Lesson Agenda

- Using Application Tracing
- Application Tracing Tools
- Using the SQL Trace Facility
- End-to-End Application Tracing
- `trcsess` Utility
  - Invoking the `trcsess` Utility
  - `trcsess` Utility: Example
- `TKPROF` Utility
  - Invoking the `TKPROF` Utility
  - `TKPROF` Sorting Operations
  - Interpreting a `TKPROF` Report: Examples

Application Tracing Tools: Overview

- Using SQL Trace:
  - DBMS_MONITOR.DATABASE_TRACE_ENABLE
  - DBMS_SESSION.SET_SQL_TRACE
  - ALTER SESSION SET SQL_TRACE
- Using End-to-End application tracing:
  - DBMS_MONITOR.CLIENT_ID_STAT_ENABLE
  - DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE
  - DBMS_MONITOR.SESSION_TRACE_ENABLE
  - DBMS_MONITOR.DATABASE_TRACE_ENABLE

Available Tracing Tools

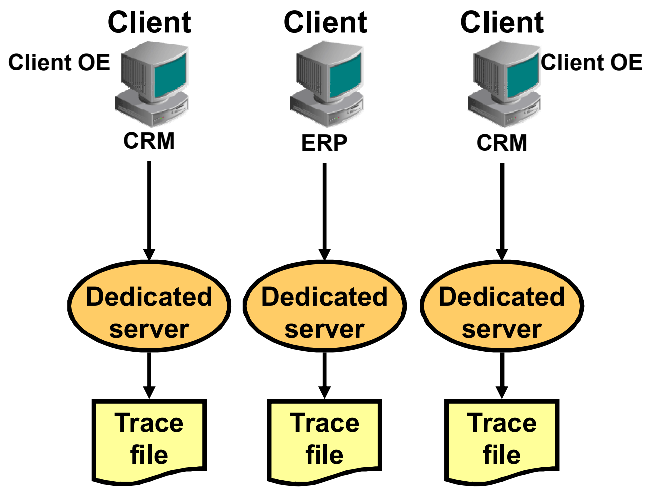
You can enable application tracing for the identified session by using one of the following, among others:

- DBMS_MONITOR (recommended)
- DBMS_APPLICATION_INFO
- DBMS_SERVICE
- DBMS_SESSION
- LOGON Trigger at a specific user level
- SQLTXPLAIN (when working with Oracle Support)

Lesson Agenda

- Using Application Tracing
- Application Tracing Tools
- Using the SQL Trace Facility
- End-to-End Application Tracing
- `trcsess` Utility
  - Invoking the `trcsess` Utility
  - `trcsess` Utility: Example
- `TKPROF` Utility
  - Invoking the `TKPROF` Utility
  - `TKPROF` Sorting Operations
  - Interpreting a `TKPROF` Report: Examples

Using the SQL Tracing Facility



- I want to retrieve traces from impacted sessions.
- I want to retrieve traces from specific users.
- I want to retrieve traces from an entire database

Oracle Database implements tracing by generating a trace file for each server process when you enable the tracing mechanism.

Tracing a specific client 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 serving it.

Tracing Your Own Session: Example

- **When to Use:** It can be used where the session is accessible to the user before the start of the statements to be traced.
- Enable the SQL Trace for your own session.

```
SQL> ALTER SESSION SET SQL_TRACE = TRUE;
```

- Execute the statement of interest.

```
SQL> select *
  2> from hr.employees natural join hr.departments
  3> where department_id = 10;
```

- Find and view the trace file.

This tracing can be used where the session is accessible to the user before the start of the statements to be traced. In addition to setting the `SQL_TRACE` parameter, you can gather 10046 trace at the session level:

```
alter session set tracefile_identifier='10046';
alter session set timed_statistics = true;
alter session set statistics_level=all;
alter session set max_dump_file_size = unlimited;
alter session set events '10046 trace name context forever,level 12';
 -- Execute the queries or operations to be traced here --
exit;
```

If the session is not exited, the trace can be disabled by using:

```
alter session set events '10046 trace name context off';
```

Note: If the section is not closed cleanly and tracing is disabled, important trace information may be missing from the trace file.

Tracing a Specific User: Example

- **When to Use:** There may be some situations where it is necessary to trace the activity of a specific user.

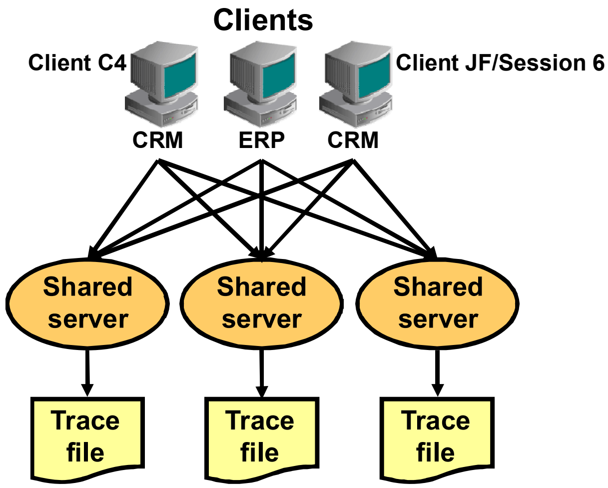
```
CREATE OR REPLACE TRIGGER SYS.set_trace
  AFTER LOGON ON DATABASE
  WHEN (USER like  '&USERNAME')
  DECLARE
      lcommand varchar(200);
 BEGIN
 EXECUTE IMMEDIATE 'alter session set statistics_level=ALL';
 EXECUTE IMMEDIATE 'alter session set max_dump_file_size=UNLIMITED';
 DBMS_MONITOR.SESSION_TRACE_ENABLE(waits=> true, binds=> true);
END set_trace;
 /
```

- Find and view the trace file.

The example in the slide shows tracing the session of a specific user using a trigger. To trace a session, the user executing the trigger needs to have been explicitly granted `alter session` and `execute on dbms_monitor` privileges, for example,

```
grant alter session to <USERNAME> ;
grant execute on dbms_monitor to <USERNAME>;
```

Consideration: Tracing Challenge



- How to retrieve traces from CRM service
- How to retrieve traces from client C4
- How to retrieve traces from session 6

Tracing a specific client is usually a problem in the shared-server model because a client is occasionally serviced by different processes. This makes it difficult to get a complete picture of the life cycle of a session. Moreover, what if you want to consolidate trace information for a particular service for performance or debugging purposes? This is also difficult because you have multiple clients using the same service and each generating trace files belonging to the server process serving it.

Note: The `trcsess` utility is useful to trace a particular session or service in such complex configurations. It is covered later in this lesson.

Lesson Agenda

- Using Application Tracing
- Application Tracing Tools
- Using the SQL Trace Facility
- End-to-End Application Tracing
- `trcsess` Utility
  – Invoking the `trcsess` Utility
  – `trcsess` Utility: Example
- `TKPROF` Utility
  – Invoking the `TKPROF` Utility
  – `TKPROF` Sorting Operations
  – Interpreting a `TKPROF` Report: Examples

End-to-End Application Tracing

- Simplifies the process of diagnosing performance problems in multitier environments by allowing application workloads to be seen by the service, module, action, client, and session
- `DBMS_MONITOR` used for finer granularity of service aggregations:
  - `SERV_MOD_ACT_STAT_ENABLE`
  - `SERV_MOD_ACT_STAT_DISABLE`
- Possible additional aggregation levels:
  - `SERVICE_NAME/MODULE`
  - `SERVICE_NAME/MODULE/ACTION`
- Database settings persist across instance restarts.

End-to-end application tracing simplifies the diagnosis of performance problems in multitier environments. In multitier environments, a request from an end client is routed to different database sessions by the middle tier, making it difficult to track a specific client. A client identifier is used to uniquely trace a specific end client through all tiers to the database server.

You can use end-to-end application tracing to identify the source of an excessive workload, such as a high-load SQL statement. Also, you can identify what a user's session does at the database level to resolve that user's performance problems.

End-to-end application tracing also simplifies management of application workloads by tracking specific modules and actions in a service. Workload problems can be identified by:

- **Client identifier:** Specifies an end user based on the login ID, such as `HR`
- **Service:** Specifies a group of applications with common attributes, service-level thresholds, and priorities; or a single application
- **Module:** Specifies a functional block within an application
- **Action:** Specifies an action, such as an `INSERT` or an `UPDATE` operation, in a module
- **Session:** Specifies a session based on a given database session identifier (SID)

The primary interface for end-to-end application tracing is Enterprise Manager. Other tools are discussed later in this lesson.

Service Tracing: Example

- Using services with client applications:

```
AP=(DESCRIPTION=
      (ADDRESS=(PROTOCOL=TCP)(HOST=mynode)(PORT=1521))
    (CONNECT_DATA=(SERVICE_NAME=AP)))

url="jdbc:oracle:oci:@ERP"

url="jdbc:oracle:thin:@(DESCRIPTION=
      (ADDRESS=(PROTOCOL=TCP)(HOST=mynode)(PORT=1521))
    (CONNECT_DATA=(SERVICE_NAME=AP)))"
```

- Trace on service:

```
exec DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE('AP');
```


The concept of a service was first introduced in Oracle 8*i* as a means for the listener to perform connection load balancing between nodes and instances of a cluster. However, the concept, definition, and implementation of services have been dramatically expanded. A service organizes work execution within the database to make it more manageable, measurable, tunable, and recoverable. A service is a grouping of related tasks within the database with common functionality, quality expectations, and priority relative to other services. A service provides a single-system image for managing competing applications that run within a single instance and across multiple instances and databases. Services can be configured, administered, enabled, disabled, and measured as a single entity by using standard interfaces, Enterprise Manager, and `SRVCTL`.

Services provide an additional dimension to performance tuning. With services, workloads are visible and measurable. Tuning by “service and SQL” replaces tuning by “session and SQL” in the majority of systems where sessions are anonymous and shared.

From a tracing point of view, a service provides a handle that permits capturing trace information by service name regardless of the session. In the second code box, all sessions that log in under the `AP` service are traced. A trace file is created for each session that uses the service. You can also enable tracing for specific tasks within a service.

Module Tracing: Example

- Call or add the following procedure in your code to set the name of the current application or module.

```
CREATE or replace PROCEDURE add_employees(...) AS
BEGIN
      DBMS_APPLICATION_INFO.SET_MODULE(
           module_name => 'add_employees',
           action_name => 'insert into emp');
      INSERT INTO emp
        (ename, empno, sal, mgr, job, ...)
      VALUES (name, emp_seq.nextval, salary, manager, ...);
      DBMS_APPLICATION_INFO.SET_MODULE(null,null);
END;
```

- Trace on SERVICE_NAME/MODULE.

```
exec DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE( -
'<SERIVCE_NAME>', 'add_employees' );
```


You can call or add the procedure in your code to set the name of the current application or module. The second example shows how the "insert into emp" action within the "add_employees" module are traced. Tracing by service, module, and action enables you to focus your tuning efforts on specific SQL, rather than sifting through trace files with SQL from different programs. Only the SQL statements that are identified with this MODULE and ACTION are recorded in the trace file. With this feature, relevant wait events for a specific module can be identified.

Action Tracing: Example

- Call or add the following procedure in your code to set the name of the current action within the current module.

```
CREATE OR REPLACE PROCEDURE bal_tran (amt IN NUMBER(7,2)) AS
BEGIN
-- balance transfer transaction
      DBMS_APPLICATION_INFO.SET_ACTION(
           action_name => 'transfer from chk to sav');
      UPDATE chk SET bal = bal + :amt WHERE acct# = :acct;
      UPDATE sav SET bal = bal - :amt WHERE acct# = :acct;
      COMMIT;
      DBMS_APPLICATION_INFO.SET_ACTION(null);
END;
```

- Trace on `SERVICE_NAME/MODULE/ACTION`.

```
exec DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE( -
'<SERVICE_NAME>', '<MODULE>', 'transfer from chk to sav' );
```


In the slide is an example of a transaction that uses the registration procedure. The action name should be descriptive text about the current action being performed. You should probably set the action name before the start of every transaction.

Set the transaction name to `NULL` after the transaction completes so that subsequent transactions are logged correctly. If you do not set the transaction name to `NULL`, subsequent transactions may be logged with the previous transaction's name.

Client Tracing: Example

- Call or add the following procedures in your code to set/clear the client ID in the session.

```
PROCEDURE set_ora_session_id (p_session_id IN VARCHAR2)
IS
BEGIN
  dbms_session.set_identifier (p_session_id);
END set_ora_session_id;

PROCEDURE clear_ora_session_id
IS
BEGIN
  dbms_session.clear_identifier;
END clear_ora_session_id;
```

- Trace a particular client identifier:

```
exec DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE
   (client_id=>'C4', waits => TRUE, binds => FALSE);
```

ORACLE

In the first example, the `p_session parameter` value is set as the client identifier for SQL tracing. You then start tracing for a particular client identifier, as shown by the second example. In this example, `C4` is the client identifier for which SQL tracing is to be enabled. The `TRUE` argument specifies that wait information is present in the trace file. The `FALSE` argument specifies that bind information is not present in the trace file.

Although not shown in the slide, you can use the `CLIENT_ID_TRACE_DISABLE` procedure to disable tracing globally for the database for a given client identifier. To disable tracing, for the previous example, execute the following command:

```
exec DBMS_MONITOR.CLIENT_ID_TRACE_DISABLE(client_id => 'C4');
```

Note

- `SET_IDENTIFIER` initializes the current session with a client identifier to identify the associated global application context.
- `client_id` is case sensitive; it must match the `client_id` parameter in the `set_context`.
- This procedure is executable by public.
- If you set the client identifier using the `DBMS_APPLICATION_INFO.SET_CLIENT_INFO` procedure, you must then run `DBMS_SESSION.SET_IDENTIFIER` so that the client identifier settings are the same.

Session Tracing: Example

- For all sessions in the database:

```
exec dbms_monitor.DATABASE_TRACE_ENABLE(TRUE,TRUE);
```

```
exec dbms_monitor.DATABASE_TRACE_DISABLE();
```

- For a particular session:

```
exec dbms_monitor.SESSION_TRACE_ENABLE(session_id=> 27,
serial_num=>60, waits=>TRUE, binds=>FALSE);
```

```
exec dbms_monitor.SESSION_TRACE_DISABLE(session_id =>27,
serial_num=>60);
```


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 for 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`.

Use the `SESSION_TRACE_DISABLE` procedure to disable the trace for a given database session identifier and serial number.

Tracing Your Own Session: Example

- Enabling trace:

```
exec DBMS_SESSION.SESSION_TRACE_ENABLE(waits => TRUE, binds =>
FALSE);
```

- Disabling trace:

```
exec DBMS_SESSION.SESSION_TRACE_DISABLE();
```

- Easily identifying your trace files:

```
alter session set tracefile_identifier='mytraceid';
```


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 viewing its contents. Each time this parameter is dynamically modified at the session level, the next trace dump written to a trace file has 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.

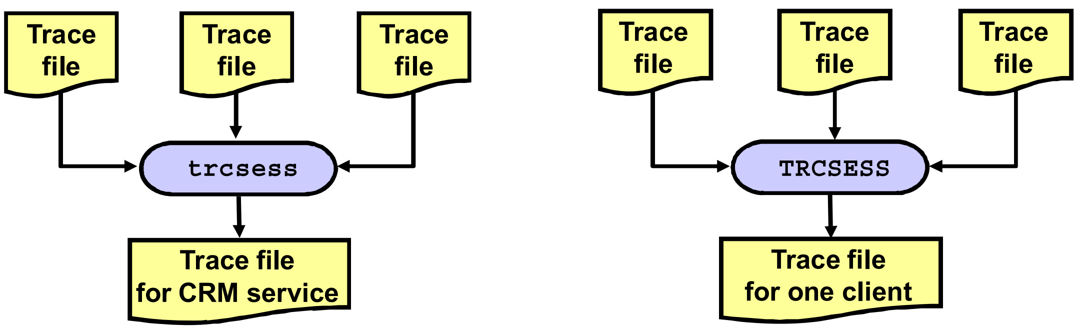
Lesson Agenda

- Using Application Tracing
- Application Tracing Tools
- Using the SQL Trace Facility
- End-to-End Application Tracing
- `trcsess` Utility
  - Invoking the `trcsess` Utility
  - `trcsess` Utility: Example
- `TKPROF` Utility
  - Invoking the `TKPROF` Utility
  - `TKPROF` Sorting Operations
  - Interpreting a `TKPROF` Report: Examples

`trcsess` Utility

Use the `trcsess` utility to consolidates trace output from selected trace files based on several criteria:

- Session ID
- Client ID
- Service name
- Action name
- Module name



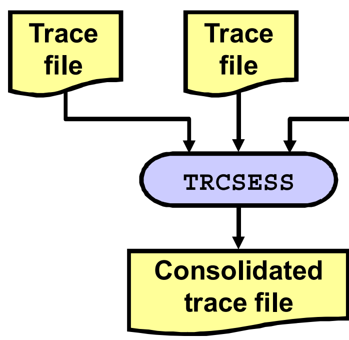
The `trcsess` utility consolidates trace output from selected trace files on the basis of several criteria: session ID, client identifier, service name, action name, and module name. After `trcsess` merges the trace information into a single output file, the output file can be processed by `TKPROF`.

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` utility 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.

Invoking the `trcsess` Utility

```
trcsess  [output=output_file_name]
         [session=session_id]
         [clientid=client_identifier]
         [service=service_name]
         [action=action_name]
         [module=module_name]
         [<trace file names>]
```



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 specified session. 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.

Using the `trcsess` Utility: Example

First session

```
exec dbms_session.set_identifier('HR session');
```

Second session

```
exec dbms_session.set_identifier('HR session');
```

Third session

```
exec
DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE(client_id=>'HR
session', waits => FALSE, binds => FALSE);
```

```
select * from employees;
```

```
select * from departments;
```

...

```
exec DBMS_MONITOR.CLIENT_ID_TRACE_DISABLE( -
client_id => 'HR session');
```

```
trcsess output=mytrace.trc clientid='HR session'
$ORACLE_BASE/diag/rdbms/systun/systun/trace/*.trc
```

ORACLE

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 sessions set their client identifiers to the `'HR session'` value. This is done using the `DBMS_SESSION` package. Then, the third session enables tracing for these two sessions by 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 by 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/systun/systun/trace` directory, which is the default in most cases.

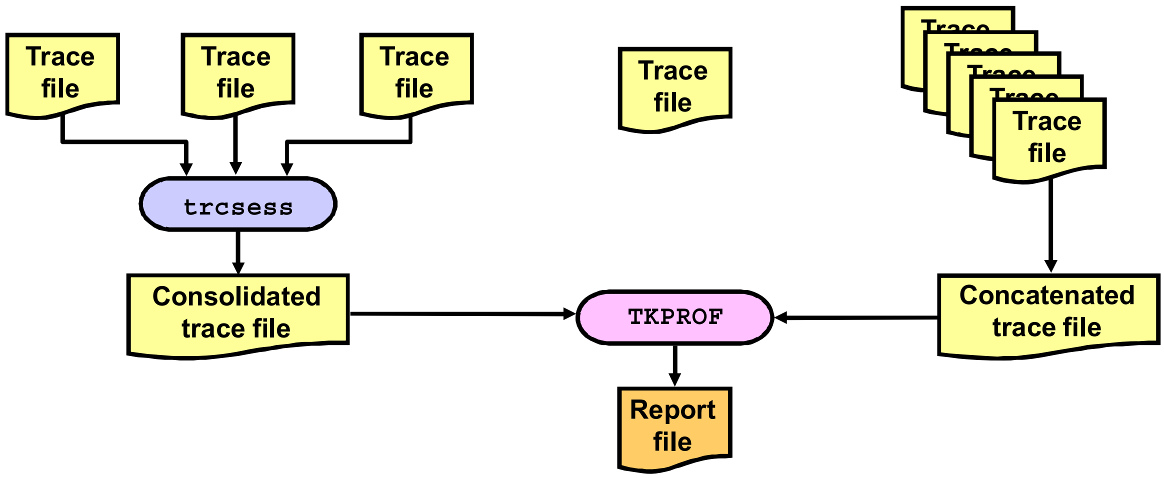
Lesson Agenda

- Using Application Tracing
- Application Tracing Tools
- Using the SQL Trace Facility
- End-to-End Application Tracing
- `trcsess` Utility
  - Invoking the `trcsess` Utility
  - `trcsess` Utility: Example
- `TKPROF` Utility
  - Invoking the `TKPROF` Utility
  - `TKPROF` Sorting Operations
  - Interpreting a `TKPROF` Report: Examples

`TKPROF` Utility: Overview

Use the `TKPROF` utility to format your SQL trace files:

- Sort raw trace files to exhibit top SQL statements.
- Filter dictionary statements.



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 several 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, and then run `TKPROF` on the result.

`TKPROF` does not report `COMMIT`s and `ROLLBACK`s 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 11*g*.

Invoking the TKPROF Utility

```
tkprof inputfile outputfile [waits=yes|no]
                            [sort=option]
                            [print=n]
                            [aggregate=yes|no]
                            [insert=sqlscriptfile]
                            [sys=yes|no]
                            [table=schema.table]
                            [explain=user/password]
                            [record=statementfile]
                            [width=n]
```


When you enter the `TKPROF` command without any arguments, it generates a usage message and 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`.
- **`sort`:** Sorts traced SQL statements in descending order of the specified sort option before listing them in the output file. If more than one option is specified, the output is sorted in descending order by the sum of the values specified in the sort options. If you omit this parameter, `TKPROF` lists statements in 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.

- **insert:** Creates a SQL script to store the trace file statistics in the database. TKPROF creates this script with the name that you specify for sqlscriptfile. This script creates a table and inserts a row of statistics for each traced SQL statement into the table.
- **sys:** Enables and disables the listing of SQL statements issued by the SYS user, or recursive SQL statements, in the output file. The default value of YES causes TKPROF to list these statements. The value of NO causes TKPROF to omit them. This parameter does not affect the optional SQL script. The SQL script always inserts statistics for all traced SQL statements, including recursive SQL statements.
- **table:** Specifies the schema and name of the table into which tkprof temporarily places execution plans before writing them to the output file. If the specified table already exists, TKPROF deletes all rows in the table, uses it for the EXPLAIN PLAN statement (which writes more rows into the table), and then deletes those rows. If this table does not exist, TKPROF creates it, uses it, and then drops it. The specified user must be able to issue INSERT, SELECT, and DELETE statements against the table. If the table does not already exist, the user must also be able to issue the CREATE TABLE and DROP TABLE statements. This option allows multiple individuals to run TKPROF concurrently with the same user in the EXPLAIN value. These individuals can specify different TABLE values and avoid destructively interfering with each other's processing on the temporary plan table. If you use the EXPLAIN parameter without the TABLE parameter, TKPROF uses the PROF$PLAN_TABLE table in the schema of the user specified by the EXPLAIN parameter. If you use the TABLE parameter without the EXPLAIN parameter, TKPROF ignores the TABLE parameter. If no plan table exists, TKPROF creates the PROF$PLAN_TABLE table and then drops it at the end.
- **explain:** Determines the execution plan for each SQL statement in the trace file and writes these execution plans to the output file. TKPROF determines execution plans by issuing the EXPLAIN PLAN statement after connecting to the system with the user and password specified in this parameter. The specified user must have CREATE SESSION system privileges. TKPROF takes longer to process a large trace file if the EXPLAIN option is used.
- **record:** Creates a SQL script with the specified file name statementfile with all the nonrecursive SQL statements in the trace file. This can be used to replay the user events from the trace file.
- **width:** An integer that controls the output line width of some TKPROF output, such as the explain plan. This parameter is useful for postprocessing of TKPROF output.

The input and output files are the only required arguments.

`TKPROF` Sorting Options

Sort Option	Description
`prscnt`	Number of times parse was called
`prscpu`	CPU time parsing
`prsela`	Elapsed time parsing
`prsdsk`	Number of disk reads during parse
`prsqry`	Number of buffers for consistent read during parse
`prscu`	Number of buffers for current read during parse
`prsmis`	Number of misses in the library cache during parse
`execnt`	Number of executes that were called
`execpu`	CPU time spent executing
`exeela`	Elapsed time executing
`exedsk`	Number of disk reads during execute
`exeqry`	Number of buffers for consistent read during execute
`execu`	Number of buffers for current read during execute

The table lists all the sort options you can use with the sort argument of `TKPROF`.

`TKPROF` Sorting Options

Sort Option	Description
`exerow`	Number of rows processed during execute
`exemis`	Number of library cache misses during execute
`fchcnt`	Number of times fetch was called
`fchcpu`	CPU time spent fetching
`fchela`	Elapsed time fetching
`fchdsk`	Number of disk reads during fetch
`fchqry`	Number of buffers for consistent read during fetch
`fchcu`	Number of buffers for current read during fetch
`fchrow`	Number of rows fetched
`userid`	User ID of user that parsed the cursor

TKPROF Report Structure

```
...
select max(cust_credit_limit) from customers where cust_city ='Paris'


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.02       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0         15          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.02       0.02          0         15          0           1

Misses in library cache during parse: 1
Optimizer mode: FIRST_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS FULL EMPLOYEES (cr=15 r=0 w=0 time=1743 us)
      1    SORT AGGREGATE (cr=7 r=0 w=0 time=777 us)
    107      TABLE ACCESS FULL EMPLOYEES (cr=7 r=0 w=0 time=655 us)

Elapsed times include waiting on following events:
Event waited on                             Times Max. Wait  Total Waited
---------------------------------------- Waited ----------  ------------
SQL*Net message to client                      2       0.00          0.00
SQL*Net message from client                    2       9.62          9.62
```


A TKPROF report for an individual cursor includes the following overall structure:

- SQL statement
- Parse/execute/fetch statistics and timings
- Library cache information
- Row source plan
- Events waited for by the statement

Parse/Execute/Fetch Statistics and Timings

This section contains the bulk of useful timing information for each statement.

This can be used in conjunction with “Row source plan” and “Events waited for by the statement” to give a full picture.

Output of the `TKPROF` Command

The `TKPROF` output file lists the statistics for a SQL statement by the SQL processing step. The step for each row that contains statistics is identified by the value of the `call` column.

- **PARSE:** This step translates the SQL statement into an execution plan and includes checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.
- **EXECUTE:** This step is the actual execution of the statement by the Oracle server. For `INSERT`, `UPDATE`, and `DELETE` statements, this step modifies the data (including sorts when needed). For `SELECT` statements, this step identifies the selected rows.
- **FETCH:** This step retrieves rows returned by a query and sorts them when needed. Fetches are performed only for `SELECT` statements.

Note: The PARSE value includes both hard and soft parses. A hard parse refers to the development of the execution plan (including optimization); it is subsequently stored in the library cache. A soft parse means that a SQL statement is sent for parsing to the database, but the database finds it in the library cache and only needs to verify things, such as access rights. Hard parses can be expensive, particularly due to the optimization. A soft parse is mostly expensive in terms of library cache activity 33.

Next to the `CALL` column, `TKPROF` displays the following statistics for each statement:

- **`Count`:** Number of times that a statement was parsed, executed, or fetched. Check this column for values greater than 1 before interpreting the statistics in the other columns. Unless the `AGGREGATE = NO` option is used, `TKPROF` aggregates identical statement executions into one summary table.
- **`CPU`:** Total CPU time in seconds for all parse, execute, or fetch calls
- **`Elapsed`:** Total elapsed time in seconds for all parse, execute, or fetch calls
- **`Disk`:** Total number of data blocks physically read from the data files on disk for all parse, execute, or fetch calls
- **`Query`:** Total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls. (Buffers are usually retrieved in consistent mode for queries.)
- **`Current`:** Total number of buffers retrieved in current mode. (Buffers typically are retrieved in current mode for data manipulation language statements. However, segment header blocks are always retrieved in current mode.)
- **`Rows`:** Total number of rows processed by the SQL statement. (This total does not include rows processed by subqueries of the SQL statement. For `SELECT` statements, the number of rows returned appears for the fetch step. For `UPDATE`, `DELETE`, and `INSERT` statements, the number of rows processed appears for the execute step.)

Note

- DISK is equivalent to physical reads from `v$sysstat` or `AUTOTRACE`.
- QUERY is equivalent to consistent gets from `v$sysstat` or `AUTOTRACE`.
- CURRENT is equivalent to db block gets from `v$sysstat` or `AUTOTRACE`.

Library Cache Information

Tracing a statement records some information regarding library cache usage, which is externalized by `TKPROF` in this section. Most important here is “Misses in library cache during parse,” which shows whether or not a statement is being reparsed. If a statement is being shared well, you should see a minimal number of misses here (1 or 0 preferably). If sharing is not occurring, high values in this field can indicate that.

Row Source Plan

This section displays the access path used at execution time for each statement, along with timing and actual row counts returned by each step in the plan. This can be very useful for several reasons.

Row source plans are generated from STAT lines in the raw trace.

If the cursor is not closed cleanly, STAT lines are not recorded and the row source plan is not displayed. Setting `SQL_TRACE` to false does not close all cursors. Cursors are closed in Oracle SQL*Plus immediately after execution. The safest way to close all cursors is to cleanly exit the session in question.

Interpreting a TKPROF Report: Example 1

- Row source plan:

```
Rows     Row Source Operation
-------  ---------------------------------------------------
 [A]   1  [B]TABLE ACCESS FULL EMPLOYEES([C]cr=15 [D]r=0 [E]w=0 [F]time=1743 us)
       1        SORT AGGREGATE (cr=7 r=0 w=0 time=777 us)
     107          TABLE ACCESS FULL EMPLOYEES (cr=7 r=0 w=0 time=655 us)
```

- Spotting relatively high resource usage:

```
update …
where …


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1         7        122          0          0          0           0
Execute      1        75        461          5     [H]297       [I]3        [J]1
Fetch        0         0          0          0          0          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2        82        583          5        297          3           2
```


- **Row count [A]:** The “row counts” output in this section is the actual number of rows returned at each step in the query execution. These actual counts can be compared with the estimated cardinalities (row counts) from an optimizer explain plan. Any differences may indicate a statistical problem that may result in a poor plan choice. For more information, see MOS note 214106.1, *Using TKProf to compare actual and predicted row counts*.
- **Row Source Operation [B]:** It shows the operation executed at this step in the plan.
- **IO Statistics [C, D, E]:** For each step in the plan, [C] is the consistent reads, [D] is the physical reads, and [E] is the writes. These statistics can be useful in identifying steps that read or write a particularly large proportion of the overall data.
- **Timing [F]:** It shows the cumulative elapsed time for the step and the steps that preceded it. This section is very useful when looking for the point in an access path that takes all the time. By looking for the point where the majority of the time is taken, it is possible to narrow down several problems.

This statement is a single execution of an update.

- **[H]** shows that this query visits 297 buffers to find the rows to update.
- **[I]** shows that only three buffers are visited to perform the update.
- **[J]** shows that only one row is updated.

Reading 297 buffers to update one row is a lot of work and tends to indicate that the access path is not efficient. Perhaps a missing index would improve the access performance.

Interpreting a TKPROF Report: Example 2

Spotting overparsing:

```
select …

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse     [M]2    [N]221        329          0         45          0           0
Execute      3    [O]  9     [P]17          0          0          0           0
Fetch        3         6          8          0      [L]4          0        [K]1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        8       236        354          0         49          0           1

Misses in library cache during parse: 1[Q]
…
```


Here you have a `SELECT` statement that you suspect may be a candidate for overparsing.

- **[K]** shows that the query returned one row.
- **[L]** shows that four buffers were read to get this row back.
- This is fine.
- **[M]** shows that the statement is parsed twice, which is not desirable because the Parse CPU usage is high [N] compared to the Execute figures [O] and [P] (that is, the elapsed time for execute is 17 seconds, but the statement spends over 300 seconds to determine the access paths, costs, and so on in the Parse phase).
- **[Q]** shows that these parses are hard parses. If [Q] were 1, the statement would have had one hard parse followed by a soft parse, (which just looks up the already parsed detail in the library cache). For more information, see MOS note 32895.1, *SQL Parsing Flow Diagram*. This is not a particularly bad example in terms of total counts because the query was executed only a few times. However, if this pattern is reproduced for each execution, this could be a significant issue. Excessive parsing should be avoided when possible by ensuring that code is shared, through one of the following methods:
  - Use bind variables.
  - Make the shared pool large enough to hold query definitions in memory long enough to be reused.

Interpreting a TKPROF Report: Example 3

Spotting queries that execute too much:

```
update …
set …
where …

call      count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse         0         0          0          0          0          0           0
Execute  488719  66476.95   66557.80          1     488729    1970566      488719
Fetch         0         0          0          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    488719  66476.95   66557.80          1     488729    1970566      488719

…
```


For the example in the slide, the update executes 488,719 times and takes a total of ~65,000 seconds to do this. The majority of the time is spent on CPU. A single row is updated per execution. For each row updated, ~1 buffer is queried. ~2 million buffers are visited to perform the update.

On an average, the elapsed time is ~0.1 seconds per execution. A subsecond execution time would normally be acceptable for most queries, but if the query is not scalable and is executed numerous times, the time can quickly add up to a large number.

It appears that, in this case, the update may be part of a loop where individual values are passed and one row is updated per value. This structure does not scale with a large number of values, meaning that it can become inefficient.

One potential solution is to try to “batch up” the updates so that multiple rows are updated within the same execution. As Oracle database releases have progressed, several optimizations and enhancements have been made to improve the handling of “batch” operations and to make them more efficient. In this way, code modifications to replace frequently executed, relatively inefficient statements by more scalable operations can have a significant impact.

What to Verify: Example

Was total elapsed time in `TKPROF` account for the application response time measured when the application was executed?

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse     1165      0.66       2.15          0         45          0           0
Execute   2926      1.23       2.92          0          0          0           0
Fetch     2945    117.03     398.23       5548    1699259         16       39654
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     7036    118.92     403.31       5548    1699304         16       39654

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0      0.00       0.00          0          0          0           0
Execute      0      0.00       0.00          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        0         0       0.00          0          0          0           0
……
```


Sometimes, to execute a SQL statement issued by a user, the Oracle Server must issue additional statements. Such statements are called recursive calls or recursive SQL statements. For example, you insert a row into a table that does not have enough space to hold that row, Oracle Database 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.

Suppose that the application ran in 410 seconds. Look in the "Overall Totals" section at the bottom of the `TKPROF` report to see the total trace elapsed time (assuming the trace file was started just before the application executed and was stopped just after the execution finished). In this case, 403 seconds out of 410 seconds seen from the user's point of view were spent in the database. Query tuning will indeed help in this situation.

For more information, see MOS note 390374.1, *Oracle Performance Diagnostic Guide*.

What to Verify: Example

Was the time spent parsing, executing, and fetching account for most of the elapsed time recorded in the trace?

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        8      0.00       0.00          0         14          0          14
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       10         0       0.00          0         14          0          14

…

Rows     Row Source Operation
-------  ---------------------------------------------------
     14  TABLE ACCESS FULL EMPLOYEES (cr=14 r=0 w=0 time=377 us)

Elapsed times include waiting on following events:
Event waited on                             Times Max. Wait  Total Waited
----------------------------------------   Waited ---------- ------------
SQL*Net message to client                       8       0.00          0.00
SQL*Net message from client                     8      16.36         78.39
```


Did the time spent parsing, executing, and fetching account for most of the elapsed time recorded in the trace?

- If so, continue to the next slide.
- If not, check client waits (“SQLNet Message from Client”) time between calls.

Are the client waits occurring in between fetch calls for the same cursor?

- If so, update the problem statement to note this fact and continue to the next slide.
- If most of the time is spent waiting in between calls for different cursors, the bottleneck is in the client tier or network. SQL tuning may not improve the performance of the application. This is no longer a query-tuning issue, but requires analysis of the client or network.

The goal of query tuning is to reduce the amount of time a query takes to parse, execute, and/or fetch data. If the trace file shows that these operations occur quickly relative to the total elapsed time, you may actually need to tune the client or network.

When the database is spending most of the time idle between executions of cursors, you should suspect that a client or network is slow.

Alternatively, when most of the query's elapsed time is idle time between fetches of the same cursor, you should suspect that the client is not utilizing bulk (array) fetches (you may see similar waits between executions of the same cursor when bulk inserts or updates are not used). The result of this is that it would be futile to tune a query that is actually spending most of its time outside the database; you must know this before you start tuning the query.

To confirm whether the waits are due to a slow client, examine the 10046 trace for the SQL statement and look for WAITs in between FETCH calls, as follows:

```
PARSING IN CURSOR #2 len=29 dep=0 uid=57 oct=3 lid=57 tim=1016349402066
hv=3058029015 ad='94239ec0'

select empno, ename from emp

END OF STMT

PARSE #2:c=0,e=5797,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=4,tim=1016349402036

EXEC #2:c=0,e=213,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1016349402675

WAIT #2: nam='SQL*Net message to client' ela= 12 p1=1650815232 p2=1 p3=0

FETCH #2:c=0,e=423,p=0,cr=7,cu=0,mis=0,r=1,dep=0,og=4,tim=1016349403494 <== Call
Finished

WAIT #2: nam='SQL*Net message from client' ela= 1103179 p1=1650815232 p2=1 p3=0 <==
Wait for client

WAIT #2: nam='SQL*Net message to client' ela= 10 p1=1650815232 p2=1 p3=0

FETCH #2:c=0,e=330,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1016350507608 <== Call
Finished (2 rows)

WAIT #2: nam='SQL*Net message from client' ela= 29367263 p1=1650815232 p2=1 p3=0 <==
Wait for client

WAIT #2: nam='SQL*Net message to client' ela= 9 p1=1650815232 p2=1 p3=0

FETCH #2:c=0,e=321,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1016379876558 <== Call
Finished (2 rows)

WAIT #2: nam='SQL*Net message from client' ela= 11256970 p1=1650815232 p2=1 p3=0 <==
Wait for client

WAIT #2: nam='SQL*Net message to client' ela= 10 p1=1650815232 p2=1 p3=0.

FETCH #2:c=0,e=486,p=0,cr=1,cu=0,mis=0,r=1,dep=0,og=4,tim=1016409054527

WAIT #2: nam='SQL*Net message from client' ela= 18747616 p1=1650815232 p2=1 p3=0

STAT #2 id=1 cnt=14 pid=0 pos=1 obj=49049 op='TABLE ACCESS FULL EMP (cr=14 pr=0 pw=0
time=377 us)'
```

Note: Between each FETCH call, there is a wait for the client. The client is slow and responds every 1 to 2 seconds.

If it appears that most waits occur in between calls for the SQL*Net message from the client event, then reduce client bottlenecks.

What to Verify: Example

- Is the query you expect to tune shown at the top of the `TKPROF` report?
- Does the query spend most of its time in the Execute and Fetch phases (not the Parse phase)? Make sure the trace file contains data only from the recent test.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse      555    100.09     300.83          0          0          0           0
Execute    555      0.42       0.78          0          0          0           0
Fetch      555     14.04      85.03        513    1448514          0       11724
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     1665    114.55     368.65        513    1448514          0       11724
```


Is the query you expect to tune shown at the top of the `TKPROF` report?

- If so, continue to the next slide.
- If not:

  Q1: Was the SQL reported in `TKPROF` as the highest elapsed time a PL/SQL procedure?

  Skip down the file to the first non-PL/SQL query. If this query is the suspected query, continue with the next question.

  Otherwise, the problem statement needs to change to identify either the PL/SQL or the first non-PL/SQL query found in the trace file. After updating the problem statement, continue with the next slide.

  Q2: Was the wrong session traced?

  Q3: Was the session traced properly?
  (started trace too late or finished too early)

Do not continue until you review your data-collection procedures to ensure that you are collecting the data properly.

Does the query spend most of its time in the Execute and Fetch phases (not the Parse phase)?

Make sure the trace file contains only data from the recent test:

- If so, you are done with verifying that this query is the one that should be tuned.
- If not, there may be a parsing problem that needs to be investigated. Normal query tuning techniques that alter the execution plan probably will not help. Update the problem statement to point out that you want to improve the parse time. Proceed to investigate possible causes for this in the lesson titled “Using Bind Variables.”

For example, the elapsed time spent parsing was 300.83 seconds compared to only 85.03 seconds for fetching. This query is having trouble parsing; tuning the query’s execution plan will not give the greatest performance gain.

Quiz

In an environment with an application server that uses a connection pool, you use _________ to identify which trace files need to be combined to get an overall trace of the application.

a. `trcsess`

b. `TKPROF`

c. Oracle SQL Developer

d. `DBMS_APPLICATION_INFO`

Answer: d

Summary

In this lesson, you should have learned how to:

- Discuss the steps needed before tracing
- Use application tracing tools
- Use the SQL Tracing facility
- Perform end-to-end application tracing
- Consolidate SQL trace files by using the `trcsess` utility
- Format trace files by using the `TKPROF` utility
- Interpret the output of the `TKPROF` command
- Verify a SQL problem by using a `TKPROF` report

Practice 3: Overview

This practice covers the following topics:

- Creating a service
- Tracing your application by using services
- Interpreting trace information by using `trcsess` and `TKPROF`

4

Understanding Basic Tuning Techniques

Objectives

After completing this lesson, you should be able to:

- Describe how to develop efficient SQL statements
- Examine some common mistakes

Lesson Agenda

- Developing Efficient SQL: Overview
- Scripts Used in This Lesson
- Examining Some Common Mistakes

Developing Efficient SQL: Overview

There are several ways you can improve SQL statement efficiency:

- Verifying optimizing statistics
- Reviewing the execution plan
- Restructuring the inefficient SQL statements
- Restructuring the indexes
- Modifying or disabling triggers and constraints
- Restructuring the data
- Maintaining stable execution plans over time
- Visiting data as few times as possible

This lesson focuses on how to develop efficient SQL statements by using basic tuning techniques.

Note: The guidelines described in this lesson are oriented to the production of frequently executed SQL. Most techniques that are discouraged here can legitimately be employed in ad hoc statements or in applications run infrequently where performance is not critical.

Scripts Used in This Lesson

The following scripts are used to show basic SQL tuning tips through examples of inefficient SQL:

- `create_sqlt.sh`: Configure the demo environment.
- Demo scripts are located in the `/home/oracle/labs/demo` directory.
  - Execute demo scripts and format the traced information by using `tkprof: $ tkprof <trace file name> <output> sys=no`
  - Review the output of the demo script: `demo<nn>_<mm>_output.txt`
- Unless specified, execute all scripts as `sqlt` (password: `oracle_4U`).

For more examples, you can use the demo scripts in the `/home/oracle/labs/demo` directory.

Before executing the demo scripts, execute `create_sqlt.sh` to set up your environment.

To capture the actual execution statistics and plan, run the `TKPROF` command instead of the explain plan.

Use the `tkprof <trace file> <output file> sys=no` command to format the traced information and review the outputs of the demo scripts.

For example: `$ tkprof systun_ora_2378_demo01_01.trc /home/oracle/Desktop/demo_04_01_01_output.txt sys=no`

Note that you already learned how to interpret the `tkprof` output.

Example 1: Table Design

Index information:

- customers: `cust_postal_code_ix: cust_postal_code`
- postal_codes: `postal_codes_pk: code1 + code2`

```
SELECT p.town_name, c.cust_last_name
FROM customers c, postal_codes p
WHERE p.code1 = substr(c.cust_postal_code,1,2)
AND p.code2 = substr(c.cust_postal_code,3,3)
AND p.code1 = '67'
AND c.country_id = 52790;

Rows     Row Source Operation
-------  ---------------------------------------------------
  911  NESTED LOOPS  (cr=3401 pr=1150 pw=0 time=330288 us)
  911   TABLE ACCESS FULL CUSTOMERS (cr=1517 pr=1150 pw=0 time=189016 us)
  911   TABLE ACCESS BY INDEX ROWID POSTAL_CODES (cr=1884 pr=0 pw=0 …)
  911    INDEX UNIQUE SCAN POSTAL_CODES_PK (cr=973 pr=0 pw=0 time=43418 us)
```


Any expression using a column, such as a function having the column as its argument, causes the optimizer to ignore the possibility of using an index on that column, even a unique index. In the example, the query looks fine, but because the `CUST_POSTAL_CODE_IX` index is not used, more data has to be scanned for the join operation (55500 rows). To reduce the amount of data to be joined in the `CUSTOMERS` table, you can create a function-based index that the database can use, or re-create the table to avoid the unnecessary column transformation.

Test with the following scripts to see if other plans are available. Assume that you can redesign the `CUSTOMERS` table. What do you observe?

- `Setup_04_01.sql`
- `demo_04_01_01.sql` (original query)
- `demo_04_01_02.sql` (original query after redesigning the table)
- `demo_04_01_03.sql` (rewritten query after redesigning the table)

Example 2: Index Usage

Index information:

customers: `customers_pk : cust_id`

```
[A] SELECT cust_first_name, cust_last_name FROM customers
    WHERE cust_id = 1030;

[B] SELECT cust_first_name, cust_last_name FROM customers
    WHERE cust_id <> 1030;

[C] SELECT cust_first_name, cust_Last_Name FROM customers
    WHERE cust_id < 10;

[D] SELECT cust_first_name, cust_last_name FROM customers
    WHERE cust_id < 10000;

[E] SELECT cust_first_name, cust_last_name FROM customers
    WHERE cust_id between 70 AND 80;
```


This example is to determine when the Oracle optimizer can use indexes. Indexes may be used for three types of conditions: equality search [A], unbounded range [B, C, D], and bounded range [E]. However, even then the optimizer considers the selectivity of the operation before using an index. (If you have time, try changing the values in the statement [E] to 7000 and 8000 and see what happens.) The index is not used if the `NOT EQUAL` (<>) operator is present. Selectivity is covered in the lesson titled " Introduction to Optimizer Statistics Concepts."

Note: Index usage may be forced by using an `INDEX` hint.

Example 3: Transformed Index

Index Information:

- customers: `cust_credit_limit : cust_cust_credit_limit_idx`

```
[A]  SELECT cust_id
     FROM customers
     WHERE cust_credit_limit*1.10 = 11000;

[B]  SELECT cust_id
     FROM customers
     WHERE cust_credit_limit = 3000/2;

Row Source Operation
---------------------------------------------------
TABLE ACCESS FULL CUSTOMERS (cr=1846 pr=1453 pw=0 time=42903 us cost=406 …)
```


The results of the query [A] and [B] show that although the `CUST_CREDIT_LIMIT` column is indexed, the index is not used by default.

[A] This can happen if the indexed column is part of an expression in the `WHERE` clause.

[B] An index is usable only if the indexed column appears clean in the `WHERE` clause and even then may be used based only on selectivity. The `CUST_CREDIT_LIMIT` column has poor selectivity because it has only eight values, which may result in the index being ignored at times.

Notice how the optimizer automatically filters on 1500.

Example 4: Data Type Mismatch

Index information:

- customers: `cust_postal_code_idx : cust_postal_code`

```
describe customers
 Name                         Null?    Type
 ---------------------------- -------- ----------------------
 CUST_POSTAL_CODE             NOT NULL VARCHAR2(10)
 …
SELECT cust_street_address
FROM customers
WHERE cust_postal_code =  68054;

Rows   Row Source Operation
----   ---------------------------------------------------
 193   TABLE ACCESS FULL CUSTOMERS (cr=1471 pr=1448 pw=0
         time=147876 us)
```


This query has a condition that forces implicit data type conversion; the `CUST_POSTAL_CODE` column is a character type and the constant is a numeric type. You should avoid mixed-mode expressions, and beware of implicit-type conversions.

Use the following scripts to examine how the optimizer handles different data types. What do you observe?

- `demo_04_04_01.sql` (numcol = numexpr)
- `demo_04_04_02.sql` (varcharcol = numberexpr)
- `demo_04_04_03.sql` (datecol = charexpr)
- `demo_04_04_04.sql` (varcharcol = dateexpr)
- `demo_04_04_05.sql` (datecol = numexpr)

Example 5: Tuning the ORDER BY Clause

```
[A] SELECT cust_first_name , cust_last_name, cust_credit_limit
    FROM customers
    ORDER BY cust_credit_limit;

[B] SELECT cust_first_name, cust_last_name, cust_credit_limit
    FROM customers
    ORDER BY cust_id;

[C] SELECT cust_first_name, cust_last_name, cust_city
    FROM customers
    WHERE cust_city = 'Paris'
    ORDER BY cust_id;

[d] SELECT cust_first_name, cust_last_name, cust_city
    FROM customers
    WHERE cust_id < 200
    ORDER BY cust_id;
```


A sort operation is a common operation in Oracle Database. If the Oracle server is able to perform all sort activity in the Program Global Area (PGA), the performance is probably acceptable. However, sometimes the Oracle server writes intermediate results to disk (temporary tablespace). The statistics on the sort operation can be obtained by using various tools. The examples [A-C] show sort operations caused by the `ORDER BY` clause. There are a few possible ways to tune the `ORDER BY` clause, such as by tuning PGA memory or by creating indexes.

Review the following outputs of the queries [A-D]. Note that all indexes are dropped except for the primary key index.

[A] Execute the query [A]. The Oracle server must perform a sort operation.

```
Rows (1st) Rows (avg) Rows (max) Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
     55500      55500      55500  SORT ORDER BY (cr=1456 pr=1453 pw=0 time=319063 us…)
     55500      55500      55500   TABLE ACCESS FULL CUSTOMERS (cr=1456 pr=1453 …)
```

Create an index on the CUST_CREDIT_LIMIT column and execute the query [A].

Even though the CUST_CREDIT_LIMIT column is indexed, the optimizer does not use the index to avoid a sort operation.

```
Rows (1st) Rows (avg) Rows (max) Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
     55500      55500      55500  SORT ORDER BY (cr=1456 pr=1453 pw=0 time=319063 us…)
     55500      55500      55500   TABLE ACCESS FULL CUSTOMERS (cr=1456 pr=1453 …)
```

[B] Replace the CUST_CREDIT_LIMIT column with the primary key column, which is CUST_ID. Even with the primary key column in the ORDER BY clause, the optimizer does not use the index on the CUST_ID column.

```
Rows (1st) Rows (avg) Rows (max) Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
     55500      55500      55500  SORT ORDER BY (cr=1456 pr=1453 pw=0 time=319063 us…)
     55500      55500      55500   TABLE ACCESS FULL CUSTOMERS (cr=1456 pr=1453 …)
```

[C] Add a condition in the WHERE clause and create a new index on the CUST_CITY column. The optimizer does not use the indexed column, CUST_ID, to avoid a sort operation.

```
Rows (1st) Rows (avg) Rows (max) Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
        77         77         77  SORT ORDER BY (cr=77 pr=77 pw=0 time=83620 us cost=65 …)
        77         77   TABLE ACCESS BY INDEX ROWID CUSTOMERS (cr=77 pr=77 pw=0 …)
        77         77         77    INDEX RANGE SCAN TEST01 (cr=2 pr=2 pw=0 time=54151 …)
```

[D] Add a condition based on the CUST_ID column in the WHERE clause. The optimizer uses the index on CUST_ID to avoid a sort operation.

```
Rows (1st) Rows (avg) Rows (max) Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
       199        199        199  TABLE ACCESS BY INDEX ROWID CUSTOMERS (cr=210 pr=55 …)
       199        199        199   INDEX RANGE SCAN CUSTOMERS_PK (cr=16 pr=2 pw=0 …)
```

As observed, the sort operation is avoided only in the query [D]. The Oracle server accesses the CUSTOMERS table based on the index key order. The result is already sorted.

Example 6: Retrieving a MAX value

Index information:

customers: `cust_cust_credit_limit_ix : cust_credit_limit`

```
[A] SELECT max(cust_credit_limit)
    FROM customers;

Row Source Operation
---------------------------------------------
SORT AGGREGATE (cr=2 pr=1 pw=0 time=1177 us)
  INDEX FULL SCAN (MIN/MAX) CUST_CUST_CREDIT_LIMIT_IX ..)

[B] SELECT max(cust_credit_limit+1000)
    FROM customers;

Row Source Operation
---------------------------------------------
SORT AGGREGATE (cr=2 pr=1 pw=0 time=1177 us)
  INDEX FULL SCAN (MIN/MAX) CUST_CUST_CREDIT_LIMIT_IX ..)

[C] SELECT max(cust_credit_limit*2)
    FROM customers;
```


The examples [A] and [B] show that an index can be useful to retrieve a maximum value (and a minimum value). If no index is available, the optimizer must scan the full table and perform a sort to find a `MIN/MAX` value. In the example [C], the operation on the indexed column value prevents the index from being used (see example 3).

Try creating an index on (`CUST_CREDIT_LIMIT*2`). Does this help?

Example 7: Retrieving a MAX value

Index information:

sales: `sales_pk : time_id + prod__id + cust_id + channel_id`

```
SELECT *
FROM sales
WHERE time_id = (SELECT max(time_id)
                 FROM sales
                 WHERE prod_id = :prod_id
                 AND cust_id = :cust_id);

Correct Result:

 PROD_ID    CUST_ID    TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
     115      11457  29-DEC-98          3        999             1       10.61
```


The query is used to retrieve the latest date when a product was sold by a customer and then return the sales details. However, the query does not return the correct result. The subquery executes before the main query and the result of the subquery is used by the main query. Eventually, the query is equivalent to `SELECT * FROM sales WHERE time_id =` *<the result from the subquery>*.

The following correlated subquery can return the correct result:

```
select *
from sales s1
where time_id = (select max(time_id)
                            from sales s2
                            where s1.prod_id = s2.prod_id
                            and s1.cust_id = s2.cust_id)
and prod_id = :prod_id
and cust_id = :cust_id;
```

Note that the rewritten query processes all the rows of the outer table. Each row in the outer table is checked against every row from the inner condition.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.02       0.02          0          0          0           0
Fetch        2      0.00       0.00          4          8          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.04       0.04          4          8          0           1
```

You can tune the query further in several ways.
If the `TIME_ID` column is a part of the index, you can take advantage of the index.

```
variable prod_id number
variable cust_id number
execute :prod_id := 13
execute :cust_id := 11453

select /*+ index_desc(sales sales_pk) */ *
from sales
where prod_id = :prod_id
and cust_id = :cust_id
and rownum = 1;

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          4          4          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          4          4          0           1
```

Example 8: Correlated Subquery

```
SELECT department_id, last_name, salary
FROM employees e1
WHERE salary > (SELECT AVG(salary)
                FROM employees e2
                WHERE e1.department_id = e2.department_id)
ORDER BY department_id;

Rows     Row Source Operation
-------  ------------------------------------------------
     38  SORT ORDER BY (cr=14 pr=7 pw=0 time=0 us cost=26..)
     38   FILTER  (cr=14 pr=7 pw=0 time=2960 us)
    107    TABLE ACCESS FULL EMPLOYEES (cr=7 pr=6 pw=0..)
     11    SORT GROUP BY NOSORT (cr=7 pr=1 pw=0 time=0..)
    106     TABLE ACCESS BY INDEX ROWID EMPLOYEES (cr=7..)
    106      INDEX RANGE SCAN EMP_DEPARTMENT_IX (cr=3 ..)
```


The query is used to return data about employees whose salaries exceed their department average. The query assigns an alias to employees and the table containing the salary information, and then uses the alias in a correlated subquery.

For each row of the `EMPLOYEES` table, the parent query uses the correlated subquery to compute the average salary for members of the same department. The correlated subquery performs the following steps for each row of the `EMPLOYEES` table:

- The `department_id` of the row is determined.
- The `department_id` is then used to evaluate the parent query.
- If the salary in that row is greater than the average salary of the departments of that row, the row is returned.

The subquery is evaluated once for each row of the `EMPLOYEES` table.

Use the following scripts to review execution statistics and execution plans. What do you observe?

- `demo_04_08_01.sql` (original query)
- `demo_04_08_02.sql` (rewritten query using inline view)

Example 9: `UNION` and `UNION ALL`

Index information:

```
customers: cust_first_name_idx : cust_first_name
           cust_last_name_idx  : cust_last_name
```

```
SELECT cust_last_name
FROM customers
WHERE cust_city = 'Paris'
UNION
SELECT cust_last_name FROM customers
WHERE cust_credit_limit < 10000;

Rows     Row Source Operation
-------  ---------------------------------------------------
    883  SORT UNIQUE (cr=2915 pr=0..cost=1016 size=535572..)
  44837   UNION-ALL  (cr=2915 pr=0 pw=0 time=79452 us)
     77    TABLE ACCESS FULL CUSTOMERS (cr=1458 pr=0..)
  44760    TABLE ACCESS FULL CUSTOMERS (cr=1457 pr=0..)
```


The `UNION` operator unconditionally results in sort operations, regardless of the presence of indexes. To investigate this, create any indexes you like. The sorts are needed because the SQL set operators are supposed to filter duplicate rows from the result.

There is one exception: The `UNION ALL` operator does not perform a sort and does not filter duplicate rows. Use the `UNION ALL` operator if you are sure that there are no duplicate rows or that duplicate rows cause no semantic problems.

Use the following script for further testing:

`demo_04_09_01.sql`

Example 10: Avoiding the Use of HAVING

Index information:

customers: `cust_cust_city_idx:  cust_city`

```
[A] SELECT cust_city, avg(cust_credit_limit)
    FROM customers
    GROUP BY cust_city
    HAVING cust_city = 'Paris';

[B] SELECT cust_city, avg(cust_credit_limit)
    FROM customers
    WHERE cust_city = 'Paris'
    GROUP BY cust_city
```


This example examines the HAVING operator. You first examine if the index on the CUST_CITY column is used.

The following trace information for the query [A] shows that the index is not used because the HAVING clause is always evaluated after the GROUP BY clause. Note that the query [A] is a badly formulated SQL statement. A HAVING clause usually contains a group function: COUNT, SUM, AVG, MIN, or MAX.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.04       0.04       1454       1459          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.04       0.04       1454       1459          0           1
```

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      1  FILTER  (cr=1459 pr=1454 pw=0 time=0 us)
    620   HASH GROUP BY (cr=1459 pr=1454 pw=0 time=866 us ….)
  55500    TABLE ACCESS FULL CUSTOMERS (cr=1459 pr=1454 …..)
```

Query [B] is logically equivalent to query [A]. The following trace information for query [B] shows that the index on the `CUST_CITY` column is used to reduce the set of rows that must be sorted.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.03        529         77          0           1
------- ------  -------- ----------   ---------- ---------- ----------  ----------
total        4      0.00       0.03        529         77          0           1

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT GROUP BY NOSORT (cr=77 pr=529 pw=0 time=0 us cost=85 size=14 card=1)
     77   TABLE ACCESS BY INDEX ROWID CUSTOMERS (cr=77 pr=529 pw=0 us …)
     77    INDEX RANGE SCAN CUST_CUST_CITY_IDX (cr=2 pr=9 pw=0 time=0 us ..)
```

Example 11: Tuning the `BETWEEN` Operator

Index information:

customers: `cust_country_state_city_ix : country_id + cust_state_province + cust_city`

```
SELECT cust_id, cust_first_name, cust_last_name,
cust_state_province
FROM customers
WHERE country_id between 52788 and 52790
AND cust_state_province like 'W%';

Rows     Row Source Operation
-------  ---------------------------------------------------
   1125  TABLE ACCESS FULL CUSTOMERS (cr=1532 pr=1453 pw=0
          time=73972 us)
```


The `BETWEEN` operator evaluates whether a value lies in a specified range. For example, `x BETWEEN a AND b` returns the same value as `(x>=a) AND (x<=b)`. If column `x` is indexed and `"x BETWEEN a AND b"` is restrictive (rejects a high percentage of the rows seen), the optimizer might choose the index. However, in the example, the optimizer creates the full table scan instead. It could be because the condition with the `BETWEEN` operator returns all matching rows in the `CUSTOMERS` table (26168 rows). It prevents the optimizer from using the Index Scan.

Use the following script to test the rewritten queries, which are logically equivalent to the one in the slide. See which query uses the optimal plan.

`demo_04_11_01.sql`

- Query1 (full table scan)
- Query2 (index skip scan)
- Query3 (index range scan)

Example 12: Tuning the Join Order

General rules:

- Avoid a full table scan if it is more efficient to get the required rows through an index.
- Avoid using an index that fetches 10,000 rows from the driving table if you could instead use another index that fetches 100 rows.
- Choose the join order so as to join fewer rows to tables later in the join order.

```
SELECT info
FROM taba a, tabb b, tabc c
WHERE a.acol BETWEEN 100 AND 200
AND b.bcol BETWEEN 10000 AND 20000
AND c.ccol BETWEEN 10000 AND 20000
AND a.key1 = b.key1
AND a.key2 = c.key2;
```


Join order can have a significant effect on performance. The main objective of SQL tuning is to avoid performing unnecessary work to access rows that do not affect the result. This leads to three general rules listed in the slide.

The example shows how to tune the join order effectively:

1. Choose the driving table and the driving index (if any).

   The first three conditions in the example are filter conditions applying to only a single table each. The last two conditions are join conditions.

   Filter conditions dominate the choice of driving table and index. In general, the driving table is the one containing the filter condition that eliminates the highest percentage of the table. Thus, because the range of 100 to 200 is narrow compared with the range of `acol`, but the ranges of 10000 and 20000 are relatively large, `taba` is the driving table, all else being equal.

With nested loops joins, the joins all happen through the join indexes, the indexes on the primary or foreign keys used to connect that table to an earlier table in the join tree. Rarely do you use the indexes on the non-join conditions, except for the driving table. Thus, after `taba` is chosen as the driving table, use the indexes on `b.key1` and `c.key2` to drive into `tabb` and `tabc`, respectively.

2. Choose the best join order, driving to the best unused filters earliest.

   You can reduce the work of the following join by first joining to the table with the best still-unused filter. Thus, if `"bcol BETWEEN ..."` is more restrictive (rejects a higher percentage of the rows seen) than `"ccol BETWEEN ..."`, the last join becomes easier (with fewer rows) if `tabb` is joined before `tabc`.
3. You can use the `ORDERED` or `STAR` hint to force the join order.

Joins are covered in later lessons in detail.

Example 13: Testing for Existence of Rows

Query:

Check only if there are customers who purchased a specific product from those who have a credit limit that is greater than 10000.

```
......
SELECT count(*) into :v_count
FROM sales s, customers c
WHERE s.cust_id = c.cust_id
AND prod_id = :prod_id
AND c.cust_credit_limit > 10000;
IF v_count > 0 THEN
......
Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=1561 pr=1441 pw=0 time=155233 us)
   1422   HASH JOIN  (cr=1561 pr=1441 pw=0 time=146010 us)
   4805    TABLE ACCESS FULL CUSTOMERS (cr=1458 pr=1441 …)
  19403    INDEX RANGE SCAN SALES_PK (cr=103 pr=0 pw=0 …)
```


The example shows a block of PL/SQL code. It is used to check only if there are customers who purchased a specific product from those who have a credit limit that is greater than 10000. The query in the example scans the entire `CUSTOMERS` table even if it checks only the existence of rows that meet the conditions.

To reduce the amount of data that needs to be scanned to satisfy the requirement, the `EXISTS` operator can be used. The `EXISTS` operator ensures that the search in the inner query does not continue when at least one match is found.

Use the following script to test with other rewritten SQL statements:

- `demo_04_13_01.sql`
  - Query 1 (original SQL)
  - Query 2 (rewritten query with the `EXISTS` operator)
  - Query 3 (rewritten query with the `EXISTS` operator)

Example 14: LIKE '%STRING'

Index information:

customers: `cust_last_name_ix : cust_last_name`

```
SELECT cust_first_name, cust_last_name
FROM customers
WHERE cust_last_name like '%ing';

Rows     Row Source Operation
-------  ---------------------------------------------------
    635  TABLE ACCESS FULL CUSTOMERS (cr=1501 pr=1426 pw=0
           time=19839 us)
```


In general, an index can be used if the search pattern looks like `STRING%` and the indexed column is very selective. However, the query in the example includes the search pattern that starts with a wildcard to find customers who have their last names ending with "ing." It causes the optimizer to ignore the possibility of using the existing index. You can test with a reverse key index to take advantage of index scan.

```
create index cust_last_name_rix on
customers(REVERSE(cust_last_name));
select cust_first_name, cust_last_name
from customers
where reverse(cust_last_name) like 'gni%';
```

Use the following script to test the query with the reverse key index.

`demo_04_14_01.sql`

- Query1 (original query)
- Query2 (rewritten query)

Example 15: Using Caution When Managing Views

Query:

Find employees in a specified state.

```
CREATE OR REPLACE VIEW emp_dept
AS
SELECT d.department_id, d.department_name, d.location_id,
       e.employee_id, e.last_name, e.first_name, e.salary,
       e.job_id FROM  departments d ,employees e
WHERE e.department_id (+) = d.department_id;

SELECT v.last_name, v.first_name, l.state_province
FROM locations l, emp_dept v
WHERE l.state_province = 'California'
AND   v.location_id = l.location_id (+);
```


Be careful when joining views, when performing outer joins to views, and when reusing an existing view for a new purpose.

1. Use caution when joining complex views.

   Joins to complex views are not recommended, particularly joins from one complex view to another. Often this results in the entire view being instantiated, and then the query is run against the view data.

2. Do not recycle views.

   Beware of writing a view for one purpose and then using it for other purposes to which it might be ill-suited. Querying from a view requires all tables from the view to be accessed for the data to be returned. Before reusing a view, determine whether all tables in the view need to be accessed to return the data. If not, then do not use the view. Instead, use the base tables, or if necessary, define a new view. The goal is to refer to the minimum number of tables and views necessary to return the required data.

Consider the following example:

```
SELECT department_name FROM emp_dept WHERE department_id = 10;
```

The entire view is first instantiated by performing a join of the employees and departments tables and then aggregating the data. However, you can obtain `department_name` and `department_id` directly from the `DEPARTMENTS` table. It is inefficient to obtain this information by querying the `emp_dept` view.

3. Use caution when unnesting subqueries.

   Subquery unnesting merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.
4. Use caution when performing outer joins to views.
   In the case of an outer join to a multitable view, the query optimizer (in Release 8.1.6 and later) can drive from an outer join column, if an equality predicate is defined on it.

An outer join within a view is problematic because the performance implications of the outer join are not visible.

Example 16: Writing a Combined SQL Statement

```
SELECT count(*)
FROM customers
WHERE cust_gender='F'
AND country_id=52771;

SELECT count(*)
FROM customers
WHERE cust_gender='F'
AND country_id=52771
AND cust_marital_status is not null;

SELECT count(*)
FROM customers
WHERE cust_gender='F'
AND country_id=52771
AND (cust_marital_status is null OR
cust_marital_status='single');
```


The example shows three queries that perform a similar task with similar conditions in the `WHERE` clause. All three queries refer to the same table, thus the `CUSTOMERS` table has to be scanned three times.

If the SQL statements are rewritten into a combined SQL statement by using the `DECODE` function, you can minimize the number of times that the `CUSTOMERS` table has to be scanned.

Use the following scripts to compare the rewritten query with the original query:

- `demo_04_16_01.sql` (Case 1)
- `demo_04_16_02.sql` (Case 2)

Example 17: Writing a Multitable INSERT Statement

```
INSERT INTO sales (product_id, customer_id, today, 3, promotion_id,
quantity_per_day, amount_per_day)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
quantity_per_day, p.prod_min_price*0.8 AS product_cost, p.prod_list_price AS
product_price
FROM sales_activity_direct s, products p
WHERE s.product_id = p.prod_id AND TRUNC(sales_date) = TRUNC(SYSDATE)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id,
s.promotion_id, p.prod_min_price*0.8,
p.prod_list_price;

INSERT INTO costs (product_id, today, promotion_id, 3,product_cost,
product_price)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
quantity_per_day, p.prod_min_price*0.8 AS product_cost, p.prod_list_price AS
product_price
FROM sales_activity_direct s, products p
WHERE s.product_id = p.prod_id AND TRUNC(sales_date) = TRUNC(SYSDATE)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id,
s.promotion_id, p.prod_min_price*0.8,
p.prod_list_price;
```


In a data warehouse scenario, you need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called extraction, transformation, and loading (ETL).

The statement given in the slide aggregates the transactional sales information, which is stored in the `SALES_ACTIVITY_DIRECT` table, on a daily basis, and it is inserted into both the `SALES` and the `COSTS` fact table for the current day.

To minimize the table scan, you can combine two `INSERT` statements into a multitable `INSERT` statement. In a multitable `INSERT` statement, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

Use the following scripts to test the different cases:

- `demo_04_17.txt`
- `demo_04_17_01.sql` (multiple inserts)
- `demo_04_17_02.sql` (unconditional ALL `INSERT`)
- `demo_04_17_03.sql` (`UPDATE`, `DELETE`, `INSERT`)
- `demo_04_17_04.sql` (`MERGE` for Conditional `INSERT/DELETE/UPDATE`)

Example 18: Using a Temporary Table

```
[A] SELECT sum(amount_sold)
    FROM sales s, times t, customers c
    WHERE s.time_id  = t.time_id
    AND s.cust_id  = c.cust_id
    AND t.day_name = 'Friday'
    AND country_id = 52772;
[B] SELECT sum(amount_sold)
    FROM sales s, times t, products p
    WHERE s.time_id  = t.time_id
    AND s.prod_id  = p.prod_id
    AND t.day_name = 'Friday'
    AND prod_category = 'Electronics';
[C] SELECT sum(amount_sold)
    FROM sales s, times t, promotions p
    WHERE s.time_id  = t.time_id
    AND s.promo_id = p.promo_id
    AND t.day_name = 'Friday'
    AND promo_category= 'TV';
```


The three queries in this example create sales reports based on different criteria. A common result is being shared here, which is the sales information on Friday. To improve the SQL efficiency, you can store intermediate results.

Rewrite the SQL statements to minimize access to the `SALES` and `TIMES` tables.

Note: Intermediate, or staging, tables are quite common in relational database systems because they temporarily store some intermediate results. In many applications, they are useful, but Oracle Database requires additional resources to create them. Always consider whether their benefit is more than the cost to create them. Avoid staging tables when the information is not reused multiple times.

Use the following scripts to test a few cases:

- `demo_04_18_01.sql`
- `demo_04_18_02.sql`

Example 19: Using the `WITH` Clause

```
[A] SELECT s.prod_id, s.amount_sold, t.week_ending_day
    FROM sales s , times t , products p
    WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id
    AND p.prod_category = 'Photo'
    AND p.prod_name LIKE '%Memory%'
    AND t.week_ending_day BETWEEN TO_DATE('01-JUL-2001','dd-
    MON-yyyy')
    AND TO_DATE('16-JUL-2001','dd-MON-yyyy');
[B] SELECT p.prod_name product, s.week_ending_day,
    SUM(s.amount_sold) revenue FROM products p LEFT OUTER
    JOIN (SELECT prod_id, amount_sold, week_ending_day
    FROM sales_numbers) s ON (s.prod_id = p.prod_id)
    WHERE p.prod_category = 'Photo' AND p.prod_name LIKE
   '%Memory%'
    GROUP BY p.prod_name, s.week_ending_day
[C] SELECT distinct week_ending_day week FROM times
    WHERE week_ending_day BETWEEN TO_DATE('01-JUL-2001','dd-
    MON-yyyy') AND TO_DATE('16-JUL-2001','dd-MON-yyyy')
[D] SELECT w.week, pr.product, nvl(pr.revenue,0) revenue
    FROM product_revenue pr PARTITION BY (product) RIGHT
    OUTER JOIN weeks w ON (w.week = pr.week_ending_day)
```


The `WITH` clause (formally known as subquery_factoring_clause) enables you to reuse the same query block in a `SELECT` statement when it occurs more than once within a complex query. Use of the SQL `WITH` clause is very similar to the use of global temporary tables (GTT), a technique that is often employed to improve query speed for complex subqueries. This is particularly useful when a query has multiple references to the same query block and there are joins and aggregations. Using the `WITH` clause, Oracle retrieves the results of a query block and stores them in the user's temporary tablespace.

The following example shows how the Oracle SQL `WITH` clause works and how the `WITH` clause and global temporary tables can be used to speed up Oracle queries.

The example in the slide compares the sales of memory card products in the Photo category for the first three week endings in July 2001. You can write a query that has multiple references to the same query block and joins and aggregations.

The following output is obtained from the queries [A, B, C, D]:

```
WEEK          PRODUCT             REVENUE                  W_W_DIFF       PERCENT
--------- -------------------   --------------     ------------------- ------------
01-JUL-01   128MB Memory Card          $0.00                               0.0
……
15-JUL-01   128MB Memory Card          $0.00          -$6,567.48           0.0
```

The query takes into account that some products may not have sold at all in that period, and it returns the increase or decrease in revenue relative to the week before. Finally, the query retrieves the percentage contribution of the memory card sales for that particular week.

Consider using the `WITH` clause in the queries [A, B, C, D] to eliminate much of this complexity by incrementally building up the queries.

`demo_04_19_01.sql`

Example 20: Partition Pruning

Index Information:

sales_np: `sales_pk : prod_id + cust_id + time_id + channel_id + promo_id`

```
SELECT sum(quantity_sold)
FROM sales_np
WHERE time_id between to_date('19980101', 'yyyymmdd')
AND to_date('19981231', 'yyyymmdd');

Rows      Row Source Operation
-------   ---------------------------------------------------
      1   SORT AGGREGATE (cr=4441 pr=4182 pw=0 …)
 178834    TABLE ACCESS FULL SALES_NP (cr=4441…)
```


A full table scan reads all rows from a table and filters out those that do not meet the selection criteria. During a full table scan, all blocks in the table that are under the high-water mark are scanned. The high-water mark indicates the amount of used space, or space that had been formatted to receive data. Each row is examined to determine whether it satisfies the statement's `WHERE` clause.

When Oracle Database performs a full table scan, the blocks are read sequentially. Because the blocks are adjacent, the database can make I/O calls larger than a single block to speed up the process. The size of the read calls range from one block to the number of blocks indicated by the `DB_FILE_MULTIBLOCK_READ_COUNT` initialization parameter. Using multiblock reads, the database can perform a full table scan very efficiently. The database reads each block only once.

If the optimizer thinks that the query requires most of the blocks in the table, it uses a full table scan, even though indexes are available.

Use the following scripts to see if the partitioned table helps.

- `demo_04_20.sql`
- `demo_04_20_01.sql`

Example 21: Using a Bind Variable

Index information:

tab1: `tab1_b_ix : b`

```
SELECT count(*) FROM tab1 WHERE a=1;   -> 1 row
SELECT count(*) FROM tab1 WHERE a=5;   -> 1 row
SELECT count(*) FROM tab1 WHERE a=100; -> 1 row

SELECT sql_text,executions
FROM v$sql
WHERE sql_text like '%tab1%';

SQL_TEXT                                         EXECUTIONS
-----------------------------------------  ----------------
select count(*) from tab1 where a=1                       1
select count(*) from tab1 where a=5                       1
select count(*) from tab1 where a=100                     1
```


There are three almost identical SQL statements stored in the library cache. For each different SQL statement, the optimizer must perform all the steps for processing a new SQL statement. This may also cause the library cache to fill up quickly because of all the different statements stored in it.

You can rewrite the codes to take advantage of cursor sharing. If the cursor is shared using a bind variable rather than a literal, there will be one shared cursor, with one execution plan.

Note: The bind variable peeking feature was introduced in Oracle Database 9*i*, Release 2. Oracle Database 11*g* changes this behavior (Adaptive Cursor Sharing). The behavior is covered in a later lesson.

Example 22: NULL Usage

Index information:

customers: `customers_marital_bix : cust_marital_status`

```
CREATE INDEX customers_maritial_bix
ON customers(cust_marital_status);

SELECT  count(*)
FROM customers;

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=1457 pr=1434 pw=0 time=106195 us)
  17428 INDEX FAST FULL SCAN CUSTOMERS_PK (cr=103 pr=0 pw=0 …)
```


To explain why the created index is not used in this example, you should be aware that the Oracle server does not store any references to `NULL` values in a regular B*-tree index. That is why the only way to find rows containing `NULL` values is to perform a full table scan. In the case of a concatenated index, if all columns in the index are `NULL`, then the same explanation applies. There are ways to eliminate the `NULL` values from the query. For example, you can define a `NOT NULL` constraint or add the `IS NOT NULL` condition in the `WHERE` clause. If the index column has low cardinality and is infrequently modified, you can consider a bitmap index as well. The bitmap index stores a `NULL` value.

Use the following scripts to examine more cases:

- `demo_04_22_01.sql` (`NULL` usage with B*-tree Index)
- `demo_04_22_02.sql` (`NULL` usage with Bitmap Index)

Example 23: Tuning a Star Query by Using the Join Operation

Index information:

- sales: `sales_pk : prod_id + cust_id + time_id + channel_id`
- products: `products_pk : prod_id`

```
SELECT /*+ index(s sales_pk) */ sum(amount_sold)
FROM sales s
WHERE prod_id BETWEEN 130 AND 150
AND cust_id BETWEEN 10000 AND 10100;

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=1385 pr=0 pw=0 time=93104 us)
    637   TABLE ACCESS BY GLOBAL INDEX ROWID SALES PARTITION: …)
    637    INDEX RANGE SCAN SALES_PK (cr=929 pr=0 pw=0 time=6561 us)(…)
```


In the example, the optimizer chooses the indexed column to return the business data. It seems to be good. However, because the `PROD_ID` column is used with the `BETWEEN` operator in the `WHERE` clause, it takes time to scan a range of values. You can rewrite the query to see if it can be tuned. Assume that you have a business query based on two tables in the star schema, which are the `SALES` table (Fact) and the `PRODUCTS` table (Dimension). There are a few possible ways to tune the query in the slide. One solution would be a join operation. For example, the `PROD_ID` column in the `SALES_PK` index is unique in the `PRODUCTS` table. Thus, if the `SALES` table is accessed through the `PRODUCTS` table, the `SALES_PK` index could be used more efficiently. Note that the join operation would not always guarantee better performance.

Use the following scripts to test with other rewritten SQL queries:

`demo_04_23_01.sql` (test with a small range) and `demo_04_23_02.sql` (test with a large range)

- Query1 (original query)
- Query2 (query with index hint)
- Query3 (rewritten query with join operation)
- Query4 (rewritten query with join hint)

Example 24: Creating a New Index

Index information:

Costs: `costs_pk : prod_id + time_id + promo_id + channel_id`

```
SELECT prod_id, time_id, promo_id, channel_id, unit_cost
FROM costs
WHERE prod_id = 120;

Rows     Row Source Operation
-------  ---------------------------------------------------
   1974  PARTITION RANGE ALL PARTITION: 1 28 (cr=743 pr=0
          pw=0 time=91505 us)
   1974   TABLE ACCESS FULL COSTS PARTITION: 1 28 (cr=743
           pr=0 pw=0 time=47925 us)
```


When the optimizer chooses access paths, it always considers the selectivity of the operation. In this example, the optimizer chooses the full table scan even though the `COSTS_PK` index exists. It may be because the `UNIT_COST` column is not a part of the `COSTS_PK` index; excessive table access is needed.

To take advantage of the index scan, you can re-create the index, including the `UNIT_COST` column. Even if more space is needed, the index scan can be used to speed up your query. Indexes are covered in later lessons.

Note: Do not use indexes as a panacea. Application developers sometimes think that performance improves when they create more indexes. If a single programmer creates an appropriate index, this index may improve the application's performance. However, if 50 developers each create an index, application performance will probably be hampered.

Use the following script to test with another rewritten SQL statement:

`demo_04_24_01.sql`

- Query1 (original query)
- Query2 (rewritten query)

Example 25: Join Column and Index

Index information:

customers: `customers_pk : cust_id`

```
SELECT cust_state_province, sum(s.amount_sold)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id
AND c.cust_year_of_birth= 1988
GROUP BY cust_state_province;
Rows     Row Source Operation
-------  ---------------------------------------------------
     14  HASH GROUP BY (cr=10558 pr=8104 pw=0 time=2462605 us)
   1903   HASH JOIN  (cr=10558 pr=8104 pw=0 time=2342695 us)
     82    TABLE ACCESS FULL CUSTOMERS (cr=1457 …)
 918843    PARTITION RANGE ALL PARTITION: 1 28 (cr=9101 …)
 918843     TABLE ACCESS FULL SALES PARTITION: 1 28 (cr=9101…)
```


The example shows how the index on the join column influences the optimizer's behavior and performance of the query. In the example, the database uses the `CUSTOMERS` table to build the hash table. The database scans the larger `SALES` table later. Note that hash joins do not use indexes and perform full table scans. Therefore, `CUSTOMERS_PK` on the `CUST_ID` column is ignored.

There are several factors to consider when the optimizer chooses a join method. In general, the optimizer uses a hash join to join two tables if they are joined by using an equijoin and if either of the following conditions is true:

- A large amount of data must be joined.
- A large fraction of a small table must be joined.

Use the following scripts to test the rewritten queries. What do you observe?

- `demo_04_25_01.sql` (hash join)
- `demo_04_25_02.sql` (nested loops join)
- `demo_04_25_03.sql` (hash join)

Example 26: Ordering Keys for Composite Index

Index information:

customers: `cust_country_state_city_ix : country_id + cust_state_province + cust_city`

```
SELECT count(*)
FROM customers
WHERE country_id > 52772
AND cust_state_province = 'CA'
AND cust_city = 'Belmont';

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=30 pr=0 pw=0 time=1634 us)
     30   INDEX SKIP SCAN CUST_COUNTRY_STATE_CITY_IX (cr=30
          pr=0 pw=0 time=1702 us)
```


A composite index contains multiple key columns. Composite indexes can provide additional advantages over single-column indexes, such as improved selectivity and reduced input/output (I/O). A SQL statement can use an access path involving a composite index when the statement contains constructs that use a leading portion of the index. In the example, the optimizer chooses `INDEX SKIP SCAN` because the condition with the `COUNTRY_ID` column is not selective (44401 rows). You might consider the different column orders in the `CUST_COUNTRY_STATE_CITY_IX` index to see if the optimizer chooses the better access path.

Use the following scripts to test with the different column orders. What do you observe?

- `demo_04_26_01.sql`
- `demo_04_26_02.sql` (column order: `country_id + cust_state_province + cust_city`)
- `demo_04_26_03.sql` (column order: `cust_city + cust_state_province + country_id`)

Example 27: Bitmap Join Index

Index Information:

- sales: `sales_pk : prod_id + cust_id + time_id + channel_id + promo_id`
- products: `products_pk : prod_id`

```
SELECT sum(s.quantity_sold)
FROM sales s, products p
WHERE s.prod_id = p.prod_id
AND p.prod_subcategory = 'CD-ROM';

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=1613 pr=2 pw=0 time=1450183 us)
  82817   HASH JOIN  (cr=1613 pr=2 pw=0 time=1840273 us)
      6    TABLE ACCESS BY INDEX ROWID PRODUCTS (cr=2 pr=0 pw=0 …)
      6     INDEX RANGE SCAN PRODUCTS_PROD_SUBCAT_IX (cr=1 …)
 918843    PARTITION RANGE ALL PARTITION: 1 28 (cr=1611 pr=2 pw=0…)
 918843     TABLE ACCESS FULL SALES PARTITION: 1 28 (cr=1611 …)
```


In the slide, the fact table and dimension table are joined to make a sales report. Because the optimizer considers a full table scan for the fact table, which is the `SALES` table, a large volume of data is joined. The volume of data that must be joined could be reduced if join indexes used as joins were already calculated.

In addition, join indexes that contain multiple dimension tables can eliminate bitwise operations that are necessary in the star transformation with existing bitmap indexes. Finally, bitmap join indexes are much more efficient in storage than materialized join views, which do not compress row IDs of the fact tables.

Use the following scripts to test the various cases and generate SQL execution statistics:

- `demo_04_27_01.sql` (original query)
- `demo_04_27_02.sql` (query with bitmap join index)
- `demo_04_27_03.sql` (query without bitmap join index)
- `demo_04_27_04.sql` (query with bitmap join index)

Example 28: Tuning a Complex Logic

Index Information:

categories: `cat_ix : prod_category_id + prod_subcat_seq`

```
......
SELECT max(prod_subcat_seq) + 1 into v_next_seq
FROM categories
WHERE prod_category_id = v_prod_category_id;
  IF sqlcode = 100 THEN
    insert into categories
    values (v_prod_category_id, 1, v_prod_subcategory);
  ELSE
    insert into categories
    values (v_prod_category_id, v_next_seq,
            v_prod_subcategory);
  END IF;
......
```


The example is a logic that requires two SQL operations. First, `SELECT` a maximum value of the `prod_subcat_seq` column, and then `INSERT` a new row with the new value of the new `prod_subcat_seq` column. You can combine two SQL operations into one statement to reduce the number of SQL executions by using an outer join.

Use the following scripts to test the rewritten SQL statements:

```
@setup_04_28
select * from categories where prod_category_id = 202 order by 1,2;
@demo_04_28_01
call register_subcat(202, 'PMP');
select * from categories where prod_category_id = 202 order by 1,2;
@demo_04_28_02
call register_subcat_new(202, 'Laptop');
select * from categories where prod_category_id = 202 order by 1,2;
```

Example 29: Using a Materialized View

```
-- Business user query 1
SELECT cust_last_name, SUM(amount_sold)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id
GROUP BY cust_last_name;

-- Business user query 2
SELECT channel_id,
SUM(amount_sold)
FROM sales
GROUP BY channel_id;
```


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. The `FROM` clause of the query can name tables, views, and other materialized views. However, the query is executed at the time the view is created, and the results are stored in a table. You can also index the materialized view in the same way that you index other tables to improve the performance of queries executed against them.

The example shows business queries that require expensive joins with computation. There are many well-known techniques you can use to increase query performance. In particular, you can create materialized views. When using materialized views in Oracle Database, you are not required to use joins or aggregations. The basic process for a materialized view is to prejoin or precompute the result of a long-running query and store this result in a database table called a materialized view. Whenever the user or application executes the SQL query, Oracle Database transparently rewrites it to use the materialized view.

Consider a materialized view that satisfies both statements.

Note that the materialized view is not covered in this course, but you can test it with the following scripts to see how it works:

- `demo_04_29_01.sql`
- `demo_04_29_02.sql`

Note: Refer to “Oracle Database Data Warehousing Guide 12*c* Release 1 (12.1)” for more details on MV.

Example 30: Star Transformation

Index information:

- sales: `sales_pk : prod_id + cust_id + time_id + channel_id + promo_id`
- products: `products_pk : prod_id`
- channels: `channels_pk : channel_id`
- customers: `customers_pk : cust_id`

```
SELECT s.amount_sold,p.prod_name,ch.channel_desc
FROM sales s, products p, channels ch, customers c
WHERE s.prod_id=p.prod_id
AND s.channel_id=ch.channel_id
AND s.cust_id=c.cust_id
AND ch.channel_id in (3, 4)
AND c.cust_city='Asten'
AND p.prod_id>100;
```


This example shows a simple star query. To get the best possible performance for star queries, it is important to follow some basic guidelines:

- A bitmap index must be built on each of the foreign key columns of the fact tables.
- The `STAR_TRANSFORMATION_ENABLED` initialization parameter should be set to `TRUE`.

This enables an important optimizer feature for star queries. It is set to `FALSE` by default for backward compatibility unless the Data Warehouse template is used for database creation.

When a data warehouse satisfies these conditions, the majority of the star queries that run in the data warehouse use a query execution strategy known as star transformation. Star transformation is a technique aimed at executing star queries efficiently.

Note that star transformation is not covered in this course, but you can use the following scripts to test the star query with and without star transformation:

- `demo_04_30_01.sql`
- `demo_04_30_02.sql`

Summary

In this lesson, you should have learned to:

- Describe how to develop efficient SQL statements
- Examine some common mistakes

Practice 4: Overview

This practice covers the following topics:

- Rewriting queries for better performance
- Rewriting applications for better performance

5

Optimizer Fundamentals

Objectives

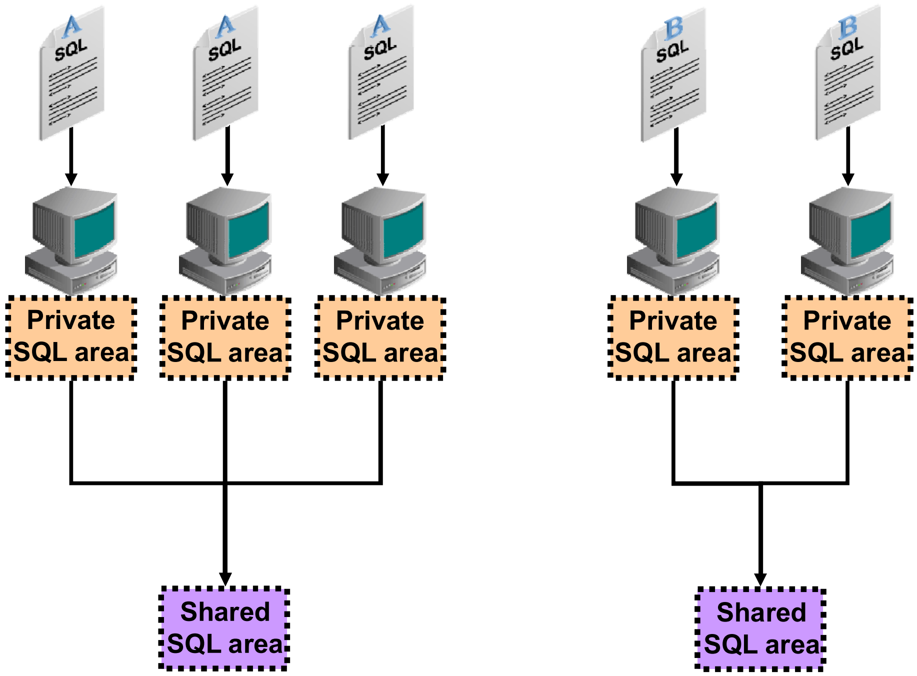
After completing this lesson, you should be able to:

- Describe each phase of SQL statement processing
- Discuss the need for an optimizer
- Explain the various phases of optimization
- Control the behavior of the optimizer

Lesson Agenda

- SQL Statement Representation
- SQL Statement Processing
- Why Do You Need an Optimizer?
- Query Transformer
- Estimator: Selectivity and Cardinality
- Plan Generator
- Adaptive Query Optimization
- Controlling the Behavior of the Optimizer

SQL Statement Representation



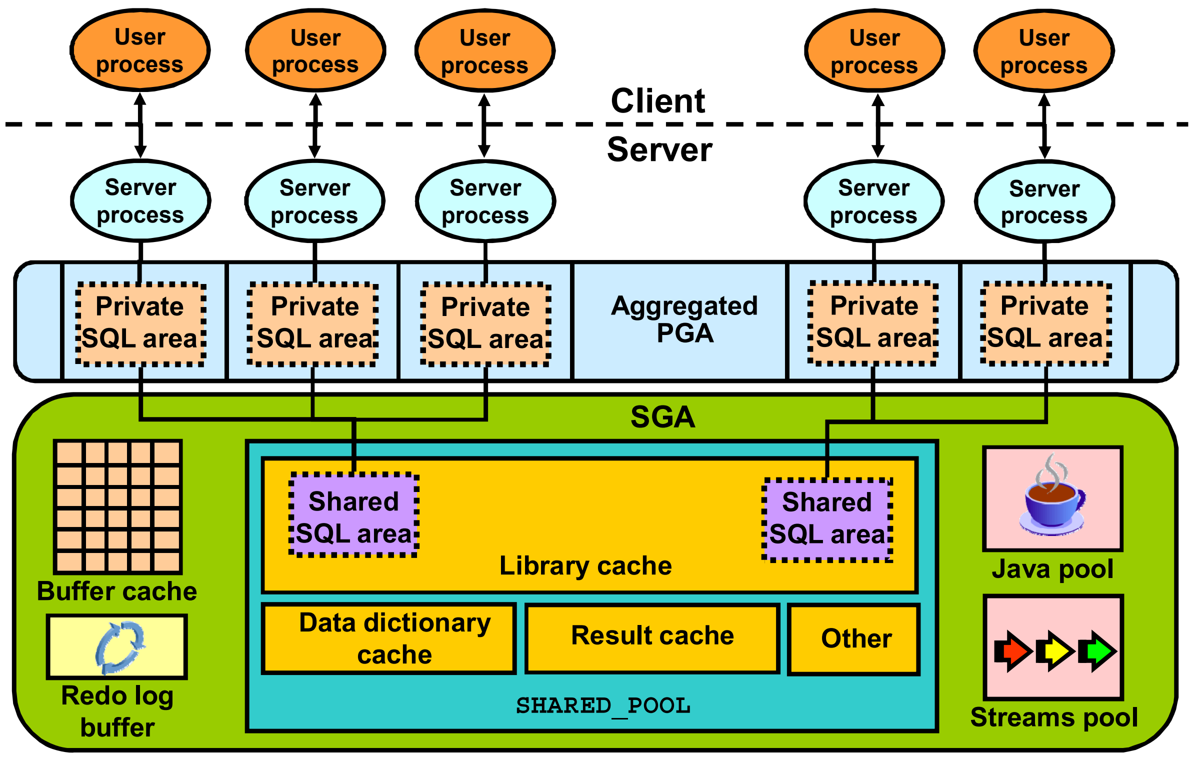
Oracle Database represents each SQL statement it runs with a shared SQL area and a private SQL area. Oracle Database recognizes when two users execute the same SQL statement and it reuses the shared SQL area for those users. However, each user must have a separate copy of the statement's private SQL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

A shared SQL area contains all optimization information that is necessary to execute the statement, whereas a private SQL area contains all runtime information that is related to a particular execution of the statement.

Oracle Database saves memory by using one shared SQL area for SQL statements that are run multiple times. Multiple runs often happen when many users run the same application.

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

SQL Statement Representation



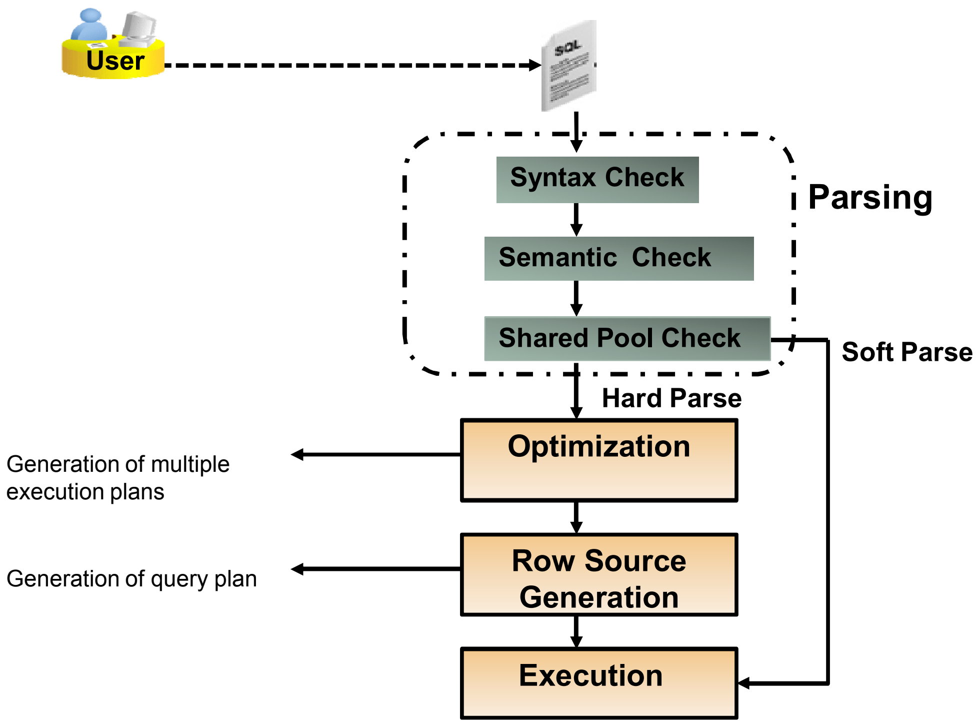
Oracle Database creates and uses memory structures for various purposes. For example, memory stores program codes that are run, data that is shared among users, and private data areas for each connected user.

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

Lesson Agenda

- SQL Statement Representation
- SQL Statement Processing
- Why Do You Need an Optimizer?
- Query Transformer
- Estimator: Selectivity and Cardinality
- Plan Generator
- Adaptive Query Optimization
- Controlling the Behavior of the Optimizer

SQL Statement Processing: Overview



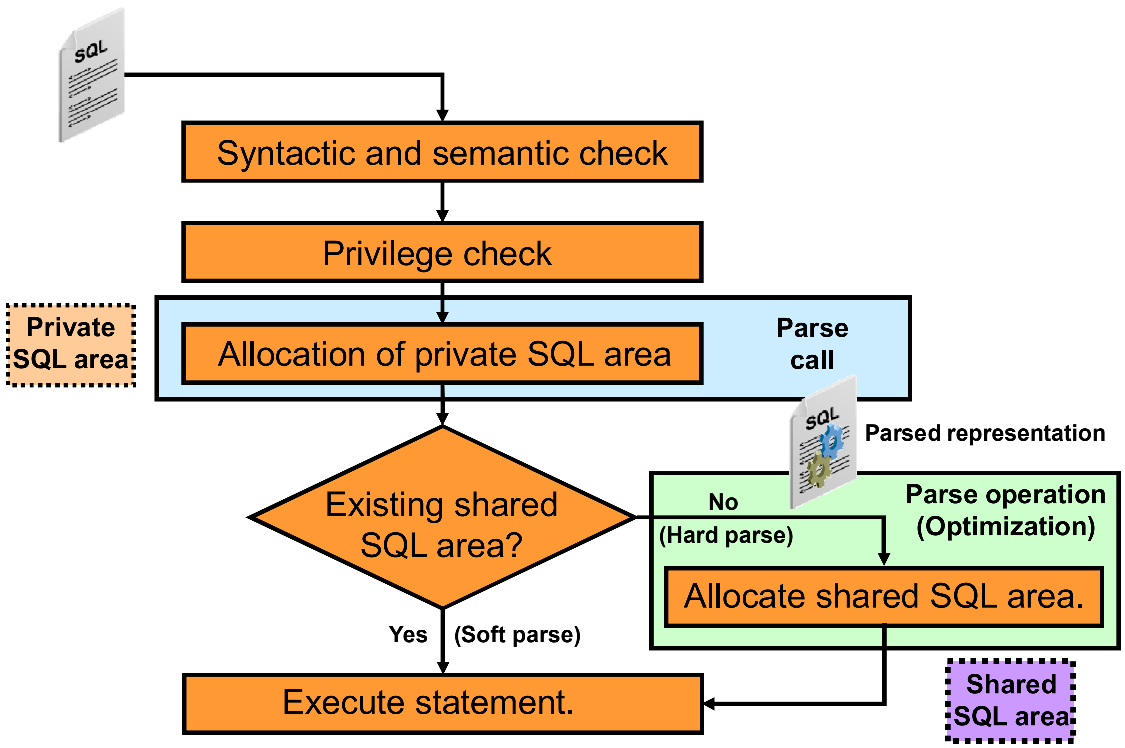
SQL processing consists of the stages of parsing, optimization, row source generation, and execution of a SQL statement. Depending on the statement, the database may omit some of these stages.

The graphic in the slide shows all the steps involved in query execution.

The steps involved in SQL processing are:

- SQL parsing
- SQL optimization
- SQL row source generation
- SQL execution

SQL Statement Parsing



SQL parsing is the first stage of SQL processing that involves separating the pieces of a SQL statement into a data structure that can be processed by other routines.

When an application issues a SQL statement, the application makes a parse call to Oracle Database. During the parse call, Oracle Database performs the following actions:

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

During the parse, the database performs a shared pool check to determine whether it can skip the resource-intensive steps of statement processing.

To this end, the database uses a hashing algorithm to generate a hash value for every SQL statement. The statement hash value is the SQL ID shown in `V$SQL.SQL_ID`.

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

- A parse call by the application associates a SQL statement with a private SQL area. After a statement has been associated with a private SQL area, it can be run repeatedly without your application making a parse call.
- A parse operation by Oracle Database allocates a shared SQL area for a SQL statement. After a shared SQL area has been allocated for a statement, it can be run repeatedly without being reparsed.

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

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

Parse operations fall into the following categories, depending on the type of statement submitted and the result of the hash check:

Hard parse

If Oracle Database cannot reuse existing code, then it must build a new executable version of the application code. This operation is known as a hard parse, or a library cache miss.

Note: The database always performs a hard parse of DDL.

During a hard parse, the database accesses the library cache and data dictionary cache several times to check the data dictionary. When the database accesses these areas, it uses a serialization device called a latch on required objects so that their definition does not change. Latch contention increases statement execution time and decreases concurrency.

Soft parse

A soft parse is any parse that is not a hard parse. If the submitted statement is the same as a reusable SQL statement in the shared pool, then Oracle Database reuses the existing code. This reuse of code is also called a library cache hit. Soft parses can vary in how much work they perform. For example, configuring the session shared SQL area can sometimes reduce the amount of latching in the soft parses, making them "softer."

In general, a soft parse is preferable to a hard parse because the database skips the optimization and row source generation steps, proceeding straight to execution.

SQL Optimization

- The purpose of the optimizer is to optimize the performance of SQL statements.
- The optimizer attempts to automatically pick the fastest method of execution for a SQL code statement.
- During the optimization stage, Oracle Database must perform a hard parse at least once for every unique DML statement and it performs the optimization during this parse.

The optimizer attempts to generate the best execution plan for a SQL statement. The best execution plan is defined as the plan with the lowest cost among all considered candidate plans.

The optimizer determines the best plan for a SQL statement by examining multiple access methods, such as full table scan or index scans, and different join methods such as nested loops and hash joins.

During the optimization stage, Oracle Database must perform a hard parse at least once for every unique DML statement and it performs the optimization during this parse. The database never optimizes DDL unless it includes a DML component such as a subquery that requires optimization.

The optimization process in more detail is covered in the later slides.

SQL Row Source Generation

- The row source generator receives an optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database.
- The execution plan takes the form of a combination of steps.
- Each step returns a row set.
- A row source is a row set returned by a step in the execution plan along with a control structure that can iteratively process the rows.
- The row source can be a table, view, or result of a join or grouping operation.
- The row source generator produces a row source tree, which is a collection of row sources.

SQL row source generator is a software that receives the optimal plan from the optimizer and outputs the execution plan for the SQL statement.

The iterative plan is a binary program that, when executed by the SQL engine, produces the result set. The execution plan takes the form of a combination of steps.

Each step returns a row set. The next step either uses the rows in this set, or the last step returns the rows to the application issuing the SQL statement.

A row source is a row set returned by a step in the execution plan along with a control structure that can iteratively process the rows.

The row source can be a table, view, or result of a join or grouping operation.

The row source generator produces a row source tree, which is a collection of row sources. The row source tree 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

SQL Row Source Generation: Example

```
SELECT e.last_name, j.job_title, d.department_name
FROM   employees e, departments d, jobs j
WHERE  e.department_id = d.department_id
AND    e.job_id = j.job_id
AND    e.last_name LIKE 'A%' ;
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3453108572

-----------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |             |     3 |   189 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                            |             |       |       |            |          |
|   2 |   NESTED LOOPS                           |             |     3 |   189 |     2   (0)| 00:00:01 |
|   3 |    NESTED LOOPS                          |             |     3 |   141 |     2   (0)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES   |     3 |    60 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN                    | EMP_NAME_IX |     3 |       |     1   (0)| 00:00:01 |
|   6 |     TABLE ACCESS BY INDEX ROWID          | JOBS        |     1 |    27 |     0   (0)| 00:00:01 |
|*  7 |      INDEX UNIQUE SCAN                   | JOB_ID_PK   |     1 |       |     0   (0)| 00:00:01 |
|*  8 |    INDEX UNIQUE SCAN                     | DEPT_ID_PK  |     1 |       |     0   (0)| 00:00:01 |
|   9 |   TABLE ACCESS BY INDEX ROWID            | DEPARTMENTS |     1 |    16 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
   7 - access("E"."JOB_ID"="J"."JOB_ID")
   8 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

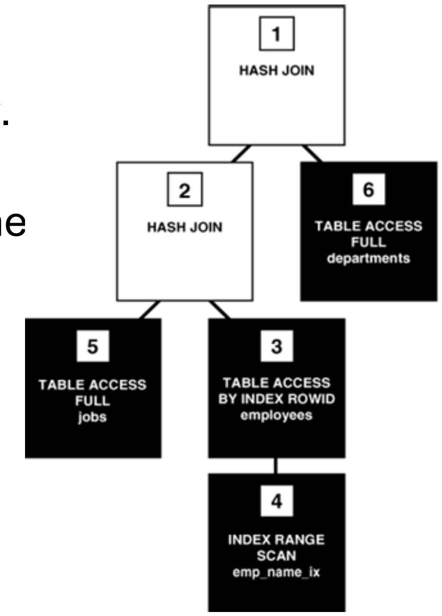
ORACLE

The example in the slide shows the execution plan of a `SELECT` statement when `AUTOTRACE` is enabled. The statement selects the last name, job title, and department name for all employees whose last names begin with the letter A.

The execution plan for this statement is the output of the row source generator.

SQL Execution

- During execution, the SQL engine executes each row source in the tree produced by the row source generator.
- Execution tree is a tree diagram that shows the flow of row sources from one step to another in an execution plan.
- When you read a plan tree you should start from the bottom up.



During execution, the SQL engine executes each row source in the tree produced by the row source generator. This step is the only mandatory step in DML processing.

The figure in the slide is an execution tree, also called a parse tree. In general, the order of the steps in execution is the reverse of the order in the plan, so you read the plan from the bottom up.

Each step in an execution plan has an ID number. Initial spaces in the Operation column of the plan indicate hierarchical relationships. For example, if the name of an operation is preceded by two spaces, then this operation is a child of an operation preceded by one space. Operations preceded by one space are children of the `SELECT` statement itself.

During execution, the database reads the data from disk into memory if the data is not in memory. The database also takes out any locks and latches necessary to ensure data integrity, and logs any changes made during the SQL execution. The final stage of processing a SQL statement is closing the cursor.

Quiz

The __________ step involves separating the pieces of a SQL statement into a data structure that can be processed by other routines.

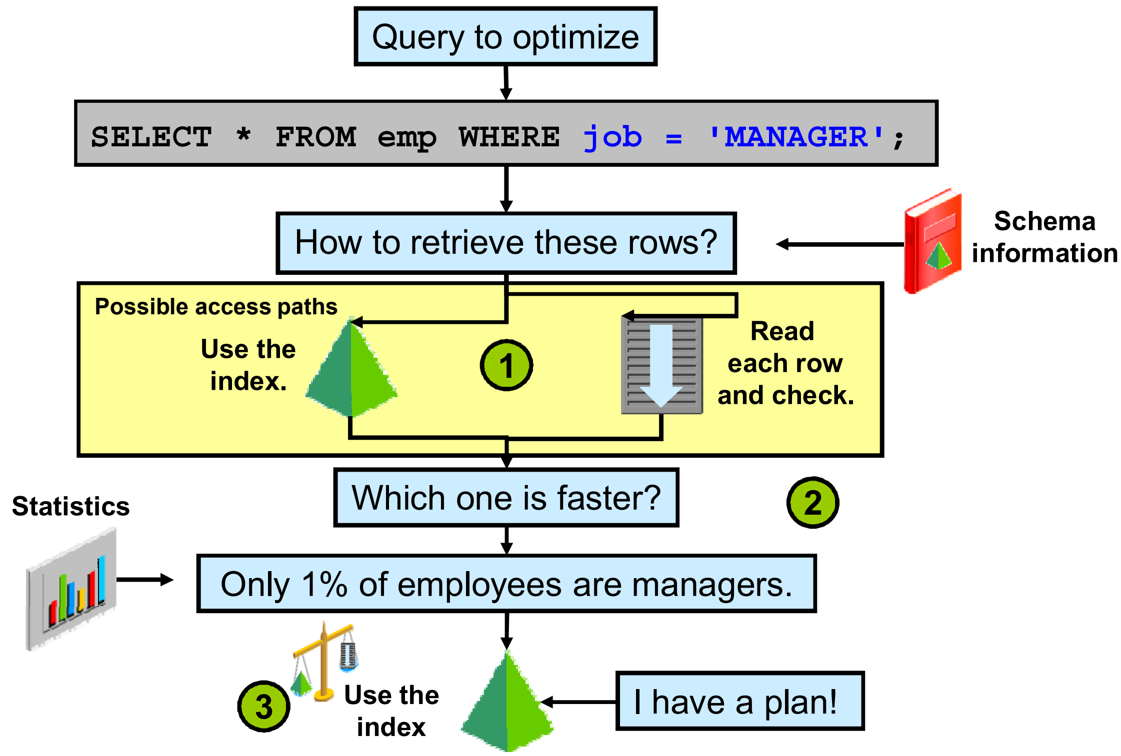
a. Parsing
b. Optimization
c. Row source generation
d. Execution

Answer: a

Lesson Agenda

- SQL Statement Representation
- SQL Statement Processing
- Why Do You Need an Optimizer?
- Query Transformer
- Estimator: Selectivity and Cardinality
- Plan Generator
- Adaptive Query Optimization
- Controlling the Behavior of the Optimizer

Why Do You Need an Optimizer?



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

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

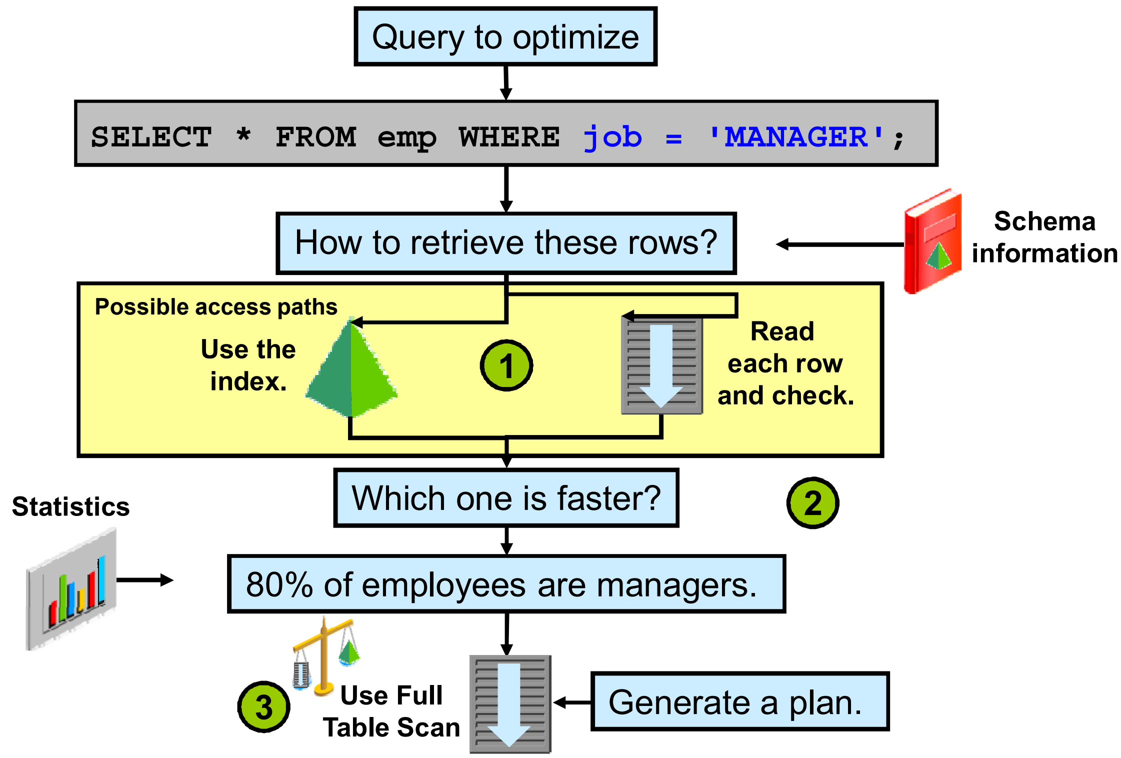
The query optimizer performs the following steps:

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.

Consider a user who queries records for employees who are managers. Here, the database statistics indicate that few employees are managers, therefore reading an index followed by a table access by rowid may be more efficient than a full table scan.

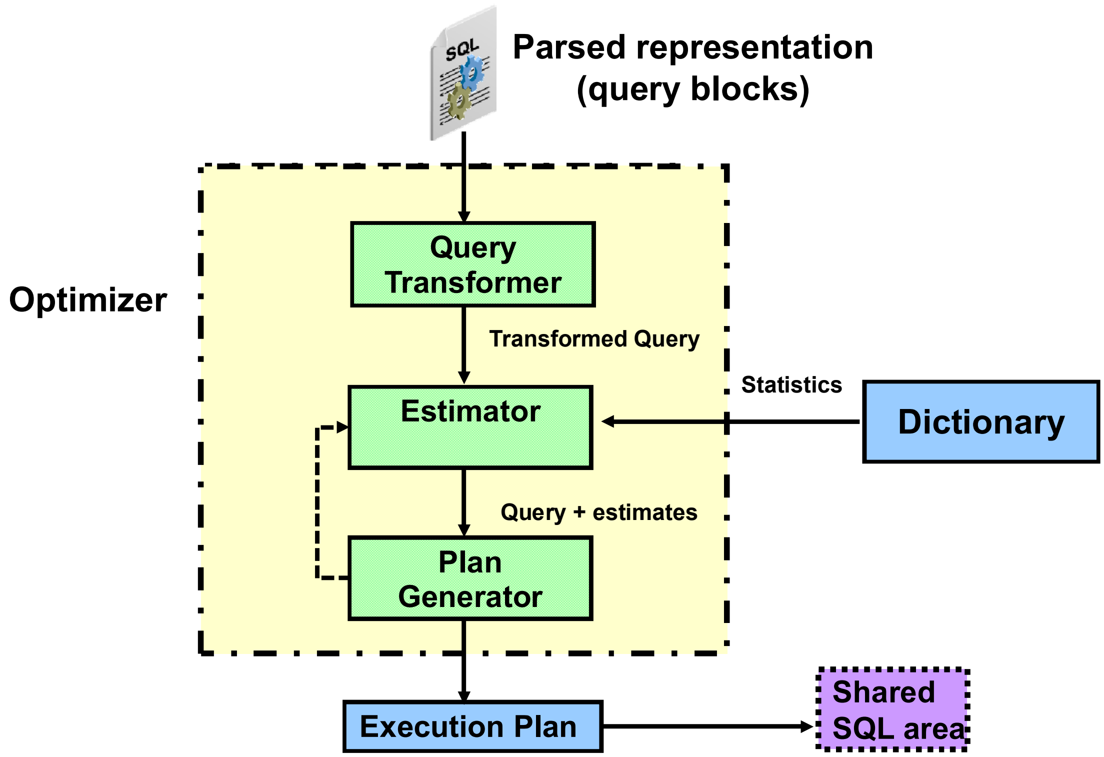
Note: Because of the complexity of finding the best possible execution plan for a particular query, the optimizer's goal is to find a "good" plan that is generally called the best cost plan.

Why Do You Need an Optimizer?



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 use of the index.

Components of the Optimizer



The optimizer creates the execution plan for a SQL statement. The optimizer tests various access paths, join orders, and join methods to find the best plan.

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.

Statistics are retrieved from the dictionary, and the query and estimates are sent to the plan generator. The plan generator either returns the plan to the estimator for comparison with other plans, or sends the query plan to the row-source generator.

Lesson Agenda

- SQL Statement Representation
- SQL Statement Processing
- Why Do You Need an Optimizer?
- Query Transformer
- Estimator: Selectivity and Cardinality
- Plan Generator
- Adaptive Query Optimization
- Controlling the Behavior of the Optimizer

Query Transformer

- Determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement that can be processed more efficiently
- Possible query transformation techniques:
  - `OR` expansion
  - Subquery unnesting (SU)
  - Complex view merging (CVM)
  - Join predicate push down (JPPD)
  - Transitive closure
  - `IN` into `EXISTS` (semijoins)
  - `NOT IN` into `NOT EXISTS` (antijoins)
  - Filter push down (FPD)

The query transformer determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement that can be processed more efficiently. Several query transformation techniques are used by the query transformer, such as transitivity, view merging, predicate pushing, subquery unnesting, query rewrite, star transformation, and `OR` expansion. The best source to find if a query transformation has taken place is 10053 traces. There are some more query transformation techniques:

- Group-by placement
- Join predicate pushdown extensions
- Null-aware antijoin (NAAJ)
- Native full outer join
- Subquery coalescing
- Disjunctive subquery unnesting

Several examples of query transformations are covered in the subsequent slides. For more examples, see MOS note 1082127.1, *Optimizer Cost Based Query Transformation*.

Transformer: OR Expansion Example

B*-tree Index

- Original query:

```
SELECT *
  FROM emp
  WHERE job = 'CLERK' OR deptno = 10;
```

- Equivalent transformed query:

```
SELECT *
  FROM emp
  WHERE job = 'CLERK'
UNION ALL
SELECT *
  FROM emp
  WHERE deptno = 10 AND job <> 'CLERK';
```

ORACLE

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

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

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

Transformer: Subquery Unnesting Example

- Original query:

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

- Equivalent transformed query:

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

Primary or unique key

ORACLE

In subquery unnesting, the optimizer transforms a nested query into an equivalent join statement, and then optimizes the join. This transformation enables the optimizer to consider the subquery tables during access path, join method, and join order selection.

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

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

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

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

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

Transformer: View Merging Example

- Original query:

Index

```
CREATE VIEW emp_10 AS
     SELECT empno, ename, job, sal, comm, deptno
     FROM emp
     WHERE deptno = 10;
```

```
SELECT empno FROM emp_10 WHERE empno > 7800;
```

- Equivalent transformed query:

```
SELECT empno
  FROM emp
  WHERE deptno = 10 AND empno > 7800;
```


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

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

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

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

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

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

Transformer: Predicate Pushing Example

- Original query:

Index

```
CREATE VIEW two_emp_tables AS
SELECT empno, ename, job, sal, comm, deptno FROM emp1
 UNION
SELECT empno, ename, job, sal, comm, deptno FROM emp2;
```

```
SELECT ename FROM two_emp_tables WHERE deptno = 20;
```

- Equivalent transformed query:

```
SELECT ename
  FROM ( SELECT empno, ename, job,sal, comm, deptno
           FROM emp1 WHERE deptno = 20
          UNION
         SELECT empno, ename, job,sal, comm, deptno
           FROM emp2 WHERE deptno = 20 );
```


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

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

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

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

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

Transformer: Transitivity Example

- Original query:

Index

```
SELECT *
 FROM emp, dept
 WHERE emp.deptno = 20 AND emp.deptno = dept.deptno;
```

- Equivalent transformed query:

```
SELECT *
 FROM emp, dept
 WHERE emp.deptno = 20 AND emp.deptno = dept.deptno
   AND dept.deptno = 20;
```


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

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

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

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

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

Hints for Query Transformation

The following hints instruct the optimizer to use a specific SQL query transformation:

- `NO_QUERY_TRANSFORMATION`
- `USE_CONCAT`
- `NO_EXPAND`
- `REWRITE` and `NO_REWRITE`
- `MERGE` and `NO_MERGE`
- `STAR_TRANSFORMATION` and `NO_STAR_TRANSFORMATION`
- `FACT` and `NO_FACT`
- `UNNEST` and `NO_UNNEST`

`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 by 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 and `OR`-expands all disjunctions, including `IN`-lists.

`NO_EXPAND`

The `NO_EXPAND` hint prevents the CBO 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. If you use the REWRITE hint with a view list and the list contains an eligible materialized view, Oracle Database uses that view regardless of its cost.

NO_REWRITE

Oracle Database does not consider views outside of the list. If you do not specify a view list, Oracle Database searches for an eligible materialized view and always uses it regardless of the cost of the final plan.

The NO_REWRITE hint instructs the optimizer to disable query rewrite for the query block, overriding the setting of the QUERY_REWRITE_ENABLED parameter.

MERGE

The MERGE hint lets you merge a view for each query.

If a view's query contains a GROUP BY clause or DISTINCT operator in the SELECT list, the optimizer can merge the view's query into the accessing statement only if complex view merging is enabled. Complex merging can also be used to merge an IN subquery into the accessing statement if the subquery is not correlated.

For correlated subqueries the query transformation can occur many times.

Complex merging is not cost based; that is, the accessing query block must include the MERGE hint. Without this hint, the optimizer uses another approach.

NO_MERGE

The NO_MERGE hint causes the Oracle server to not merge mergeable views. This hint lets the user have 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.

STAR_TRANSFORMATION

The STAR_TRANSFORMATION hint causes the optimizer to use the best plan in which the transformation has been used. Without the hint, the optimizer could make a cost-based decision to use the best plan that is generated without the transformation, instead of the best plan for the transformed query.

Even if the hint is given, there is no guarantee that the transformation will take place. The optimizer generates the subqueries only if it seems reasonable to do so. If no subqueries are generated, there is no transformed query, and the best plan for the untransformed query is used regardless of the hint.

FACT

The `FACT` hint is used in the context of the star transformation to indicate to the transformation that the hinted table should be considered a fact table.

NO_FACT

The `NO_FACT` hint is used in the context of the star transformation to indicate to the transformation that the hinted table should not be considered as a fact table.

Note: The `NO_INDEX` hint is useful if you use distributed query optimization. It applies to function-based, B*-tree, bitmap, and domain indexes. If this hint does not specify an index name, the optimizer does not consider a scan on any index on the table.

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.

Before a subquery is unnested, the optimizer first verifies whether the statement is valid. The statement must then pass heuristic and query optimization tests. The `UNNEST` hint instructs the optimizer to check the subquery block only for validity. If the subquery block is valid, subquery unnesting is enabled without checking the heuristics or costs.

NO_UNNEST

Use of the `NO_UNNEST` hint turns off unnesting.

Quiz

The view merging optimization applies to views that contain only selections, projections, and joins.

a. True
b. False

Answer: a

Quiz

Review the following query and execution plan. Which statements are true in 12*c*?

```
SELECT p.prod_id, v1.cnt
FROM products p,(SELECT s.prod_id, count(*) cnt
                 FROM sales s
                 WHERE s.quantity_sold BETWEEN 1 AND 47
                 GROUP BY s.prod_id) v1
WHERE p.supplier_id = 12
AND p.prod_id = v1.prod_id(+);
```

```
-------------------------------------------------------------------------------------------
| Operation                                  | Name          | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------
| SELECT STATEMENT                           |               |     1 |    20 |  420    (0)|
|  NESTED LOOPS OUTER                        |               |     1 |    20 |  420    (0)|
|   TABLE ACCESS FULL                        | PRODUCTS      |     1 |     7 |    3    (0)|
|   VIEW PUSHED PREDICATE                    |               |     1 |    13 |  417    (0)|
|    FILTER                                  |               |       |       |            |
|     SORT AGGREGATE                         |               |     1 |     7 |            |
|      PARTITION RANGE ALL                   |               | 12762 | 89334 |  417    (0)|
|       TABLE ACCESS BY LOCAL INDEX ROWID    | SALES         | 12762 | 89334 |  417    (0)|
|        BITMAP CONVERSION TO ROWIDS         |               |       |       |            |
|         BITMAP INDEX SINGLE VALUE          | SALES_PROD_BIX|       |       |            |
-------------------------------------------------------------------------------------------
```


Note

See the following page for the question and answer option related to this quiz.

Quiz

a. A query transformation has taken place.

b. The optimizer considered a nested loops join when table `p` and view `v1` are joined even with the `GROUP BY` clause in the view.

c. The optimizer considered only two possible join methods: a hash join and sort-merge join to join table `p` and view `v1`.

d. A join predicate pushdown has become possible, and you are now taking advantage of the index on the `SALES` table to do a nested loops join instead of a hash join.

Answer: a, b, d

For more information, review the white paper titled *Upgrading from Oracle Database 10g to 11g: What to expect from the Optimizer* by accessing the following link:

http://www.oracle.com/technetwork/database/focus-areas/bi-datawarehousing/twp-upgrading-10*g*-to-11*g*-what-to-ex-133707.pdf

Lesson Agenda

- SQL Statement Representation
- SQL Statement Processing
- Why Do You Need an Optimizer?
- Query Transformer
- Estimator: Selectivity and Cardinality
- Plan Generator
- Adaptive Query Optimization
- Controlling the Behavior of the Optimizer

Query Estimator: Selectivity and Cardinality

- Selectivity is the estimated proportion of a row set retrieved by a particular predicate or combination of predicates.

$$\text{Selectivity} = \frac{\text{Number of rows satisfying a condition}}{\text{Total number of rows}}$$

- Expected number of rows retrieved by a particular operation in the execution plan:

$$\text{Cardinality} = \text{Total number of rows} * \text{Selectivity}$$

- Selectivity is expressed as a value between 0.0 and 1.0:
  - High selectivity: Small proportion of rows
  - Low selectivity: Big proportion of rows
- Selectivity computation:
  - If no statistics: Use dynamic statistics.
  - If no histograms: Assume even distribution of rows.

ORACLE

Query estimator determines the overall cost of a given execution plan.

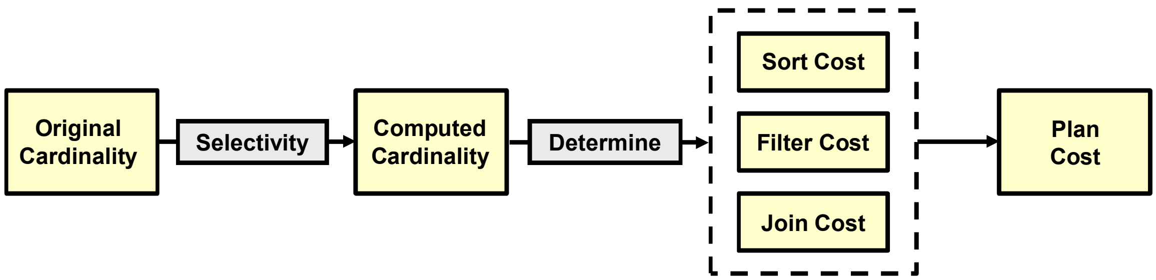
Selectivity represents a fraction of rows from a row set. The row set can be a base table, a view, or the result of a join or a `GROUP BY` operator. The selectivity is tied to a query predicate, such as `last_name = 'Smith'`, or a combination of predicates, such as `last_name = 'Smith' AND job_type = '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 uses either dynamic statistics or an internal default value, depending on the value of the `OPTIMIZER_DYNAMIC_STATICTICS` 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).

Importance of Selectivity and Cardinality

- Selectivity affects the estimates of I/O cost.
- Selectivity affects the sort cost.
- Cardinality is important to determine join, filter, and sort costs.
- If incorrect selectivity and cardinality are used, the optimizer estimates the plan cost incorrectly.



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.

Selectivity and Cardinality: Example

Facts:

- The number of rows in the `CUSTOMERS` table is `55500`.
- The number of distinct values in:
  - `CUST_CITY: 620`
  - `CUST_STATE_PROVINCE: 145`
  - `COUNTRY_ID: 19`

```
SELECT * FROM customers
WHERE cust_city = 'EDISON'
AND cust_state_province = 'NJ'
AND country_id = 12345;
```

Questions:

- What is the selectivity of the `WHERE` predicate?
- What is the computed cardinality for the same predicate?

In the example in the slide, 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.

Based on this assumption, the optimizer deduces that the selectivity of the `WHERE` predicate is (1/620) * (1/145) * (1/19) (assuming there are no histograms) and that the cardinality of the query is (selectivity)*55500.

Selectivity and Cardinality: Example

Facts:

- The number of rows in the `CUSTOMERS` table is `55500`.
- The number of distinct values in:
  - `CUST_CITY: 620`
  - `CUST_STATE_PROVINCE: 145`
  - `COUNTRY_ID: 19`

```
SELECT * FROM customers
WHERE cust_city = 'EDISON' AND
  cust_state_province = 'NJ' AND country_id = 12345;
```

Questions:

- Is the estimated selectivity the same as the actual selectivity? If not, describe why it is different.

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 always results in underestimation of the selectivity. Starting with Oracle Database 11*g*, you can 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. This topic is covered in the lesson titled “Introduction to Optimizer Statistics Concepts.”

Query Estimator: Cost

- Cost is the optimizer's best estimate of the number of standardized I/Os it takes to execute a particular statement.
- Cost unit is a standardized single-block random read:
  - 1 cost unit = 1 SRd
- Example:
  - If a plan costs 1,000, the optimizer computes that it should take as long as 1,000 single-block reads.
  - Remember that it is an estimation.

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

Query Estimator: Cost Components

- The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.
  - The operations may be scanning a table, accessing rows from a table by using an index, joining two tables together, or sorting a row set.
- The cost formula combines three different costs units into standard cost units.

Single-block I/O cost		Multiblock I/O cost		CPU cost
`#SRds*sreadtim`	+	`#MRds*mreadtim`	+	`#CPUCycles/cpuspeed`

```
Cost= (#SRds*sreadtim + #MRds*mreadtim + #CPUCycles/cpuspeed) / sreadtim
```

`#SRds`: Number of single-block reads
`#MRds`: Number of multiblock reads
`#CPUCycles`: Number of CPU Cycles

`Sreadtim`: Single-block read time
`Mreadtim`: Multiblock read time
`Cpuspeed`: Millions instructions per second

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

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

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

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

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

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

Lesson Agenda

- SQL Statement Representation
- SQL Statement Processing
- Why Do You Need an Optimizer?
- Query Transformer
- Estimator: Selectivity and Cardinality
- Plan Generator
- Adaptive Query Optimization
- Controlling the Behavior of the Optimizer

Plan Generator

```
select e.last_name, d.department_name
from   employees e, departments d
where  e.department_id = d.department_id;
```

```
Join order[1]:  DEPARTMENTS[D]#0  EMPLOYEES[E]#1
 NL Join:  Cost: 41.13  Resp: 41.13  Degree: 1
 SM cost: 8.01
 HA cost: 6.51
Best:: JoinMethod: Hash
Cost: 6.51  Degree: 1  Resp: 6.51  Card: 106.00
Join order[2]:  EMPLOYEES[E]#1  DEPARTMENTS[D]#0
 NL Join:  Cost: 121.24  Resp: 121.24  Degree: 1
 SM cost: 8.01
 HA cost: 6.51
Join order aborted
Final cost for query block SEL$1 (#0)
All Rows Plan:
Best join order: 1
+-----------------------------------------------------------------+
| Id  | Operation              | Name         | Rows  | Bytes  | Cost  |
+-----------------------------------------------------------------+
| 0   | SELECT STATEMENT       |              |       |        |     7 |
| 1   |  HASH JOIN             |              |   106 |   6042 |     7 |
| 2   |   TABLE ACCESS FULL    | DEPARTMENTS|     27 |    810 |     3 |
| 3   |   TABLE ACCESS FULL    | EMPLOYEES  |    107 |   2889 |     3 |
+-----------------------------------------------------------------+
```


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 loops, 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 (that is, explores more alternative 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 Access Paths,” “Optimizer Join Operations,” and “Generating and Displaying Execution Plans.”

Quiz

Background

- Suppose that a customer reported a problem query that takes 10 minutes to execute. The explain plan for that query is about the same as the row source plan from the `TKPROF` showing 10 minutes. The cost of that explain plan is 2,000.

Questions

- Was the optimizer quite accurate when it computed the cost as 2,000?

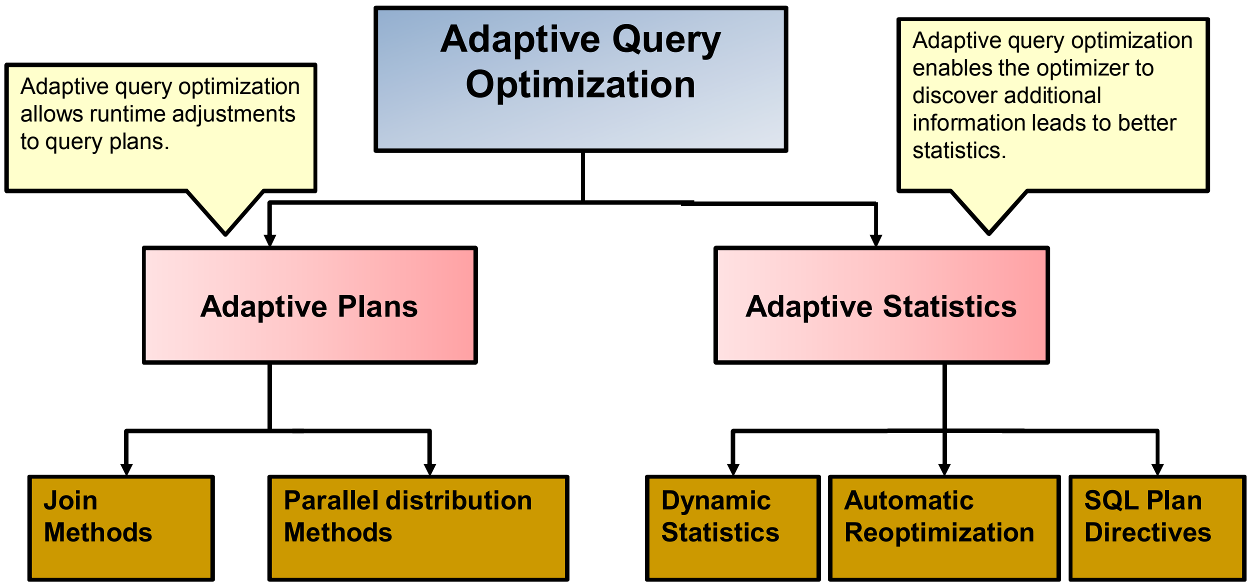
The plan cost of 2,000 means that the optimizer computed that it should take as long as 2,000 single-block reads, because Oracle Database assumes that single-block read time (`sreadtim`) is 12 ms by default.

So, a rough estimate of elapsed time is 2,000 * 12 ms (24 sec), which means the optimizer underestimated the cost. Actual performance could be off by one order of magnitude and still be all right. If it is off by several orders, it raises a concern. You might need to check the contents of optimizer trace.

Lesson Agenda

- SQL Statement Representation
- SQL Statement Processing
- Why Do You Need an Optimizer?
- Query Transformer
- Estimator: Selectivity and Cardinality
- Plan Generator
- Adaptive Query Optimization
- Controlling the Behavior of the Optimizer

Adaptive Query Optimization



In Oracle Database, adaptive query optimization is a set of capabilities that enables the optimizer to make runtime adjustments to execution plans and discover additional information that can lead to better statistics. Adaptive optimization is helpful when existing statistics are not sufficient to generate an optimal plan.

The following graphic shows the feature set for adaptive query optimization.

Note: Adaptive plans are discussed in more detail in the lessons titled "Interpreting Execution Plans and Enhancements" and adaptive statistics are discussed in Appendix F titled "Gathering and Managing Optimizer Statistics."

Lesson Agenda

- SQL Statement Representation
- SQL Statement Processing
- Why Do You Need an Optimizer?
- Query Transformer
- Estimator: Selectivity and Cardinality
- Plan Generator
- Adaptive Query Optimization
- Controlling the Behavior of the Optimizer

Controlling the Behavior of the Optimizer

- **CURSOR_SHARING**: EXACT, FORCE
- **DB_FILE_MULTIBLOCK_READ_COUNT**
- **PGA_AGGREGATE_TARGET**
- **STAR_TRANSFORMATION_ENABLED**
- **RESULT_CACHE_MODE**: MANUAL, FORCE
- **RESULT_CACHE_MAX_SIZE**
- **RESULT_CACHE_MAX_RESULT**
- **RESULT_CACHE_REMOTE_EXPIRATION**

These parameters control optimizer behavior:

- CURSOR_SHARING determines what kind of SQL statements can share the same cursors:
  - **FORCE:** Forces statements that may differ in some literals, but are otherwise identical to share a cursor, unless the literals affect the meaning of the statement
  - **EXACT:** Allows only statements with identical text to share the same cursor. This is the default.
- DB_FILE_MULTIBLOCK_READ_COUNT is one of the parameters you can use to minimize I/O during table scans or index fast full scan. It specifies the maximum number of blocks read in one I/O operation during a sequential scan. The total number of I/Os needed to perform a full table scan or an index fast full scan depends on such factors as the size of the segment, the multiblock read count, and whether parallel execution is being utilized for the operation. As of Oracle Database 10*g* Release 2, the default value of this parameter is a value that corresponds to the maximum I/O size that can be performed efficiently. This value is platform-dependent and is calculated at instance startup for most platforms.

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

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

- The memory size allocated to the result cache depends on the memory size of the SGA as well as the memory management system. You can change the memory allocated to the result cache by setting the `RESULT_CACHE_MAX_SIZE` parameter. The result cache is disabled if you set its value to 0. The value of this parameter is rounded to the largest multiple of 32 KB that is not greater than the specified value. If the rounded value is 0, the feature is disabled.
- Use the `RESULT_CACHE_MAX_RESULT` parameter to specify the maximum amount of cache memory that can be used by any single result. The default value is 5 percent, but you can specify any percentage value between 1 and 100.
- Use the `RESULT_CACHE_REMOTE_EXPIRATION` parameter to specify the time (in number of minutes) for which a result that depends on remote database objects remains valid. The default value is 0, which implies that results that use remote objects should not be cached. Setting this parameter to a nonzero value can produce stale answers; for example, if the remote table used by a result is modified at the remote database.

Controlling the Behavior of the Optimizer

- **OPTIMIZER_INDEX_CACHING**
- **OPTIMIZER_INDEX_COST_ADJ**
- **OPTIMIZER_FEATURES_ENABLED**
- **OPTIMIZER_MODE**: ALL_ROWS, FIRST_ROWS, FIRST_ROWS_*n*
- **OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES**
- **OPTIMIZER_USE_SQL_PLAN_BASELINES**
- **OPTIMIZER_DYNAMIC_SAMPLING**
- **OPTIMIZER_USE_INVISIBLE_INDEXES**
- **OPTIMIZER_USE_PENDING_STATISTICS**

ORACLE

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

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

- `OPTIMIZER_MODE` establishes the default behavior for selecting an optimization approach for either the instance or your session. The possible values are:
  - `ALL_ROWS`: The optimizer uses a cost-based approach for all SQL statements in the session, regardless of the presence of statistics, and optimizes with a goal of best throughput (minimum resource use to complete the entire statement). This is the default value.
  - `FIRST_ROWS_n`: The optimizer uses a cost-based approach, regardless of the presence of statistics, and optimizes with a goal of best response time to return the first *n* number of rows; *n* can equal 1, 10, 100, or 1000.
  - `FIRST_ROWS`: The optimizer uses a mix of cost and heuristics to find the best plan for fast delivery of the first few rows. Using heuristics sometimes leads the query optimizer to generate a plan with a cost that is significantly larger than the cost of a plan without applying the heuristic. `FIRST_ROWS` is available for backward compatibility and plan stability; use `FIRST_ROWS_n` instead.
- `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` enables or disables the automatic recognition of repeatable SQL statements, as well as the generation of SQL plan baselines for such statements.
- `OPTIMIZER_USE_SQL_PLAN_BASELINES` enables or disables the use of SQL plan baselines stored in SQL Management Base. When enabled, the optimizer looks for a SQL plan baseline for the SQL statement being compiled. If one is found in SQL Management Base, the optimizer costs each of the baseline plans and picks one with the lowest cost.
- `OPTIMIZER_DYNAMIC_SAMPLING` controls the level of dynamic statistics performed by the optimizer. If `OPTIMIZER_FEATURES_ENABLE` is set to:
  - 10.0.0 or later, the default value is 2
  - 9.2.0, the default value is 1
  - 9.0.1 or earlier, the default value is 0
  - When this parameter is set to 11, the optimizer will use dynamic statistics to verify cardinality estimates for all SQL operators, and it will determine an internal time limit to spend verifying the estimates.
- `OPTIMIZER_USE_INVISIBLE_INDEXES` enables or disables the use of invisible indexes.
- `OPTIMIZER_USE_PENDING_STATISTICS` specifies whether or not the optimizer uses pending statistics when compiling SQL statements.

Note: Invisible indexes, pending statistics, and dynamic statistics are discussed in the lesson titled "Introduction to Optimizer Statistics Concepts".

Optimizer Features and Oracle Database Releases

`OPTIMIZER_FEATURES_ENABLED`

Features	9.0.0 to 9.2.0	10.1.0 to 10.1.0.5	10.2.0 to 10.2.0.2	11.1.0.6	12.1.0
Index fast full scan	✓	✓	✓	✓	✓
Consideration of bitmap access to paths for tables with only B*-tree indexes	✓	✓	✓	✓	✓
Complex view merging	✓	✓	✓	✓	✓
Peeking into user-defined bind variables	✓	✓	✓	✓	✓
Index joins	✓	✓	✓	✓	✓
Dynamic statistics		✓	✓	✓	✓
Query rewrite enables		✓	✓	✓	✓
Skip unusable indexes		✓	✓	✓	✓
Automatically compute index statistics as part of creation		✓	✓	✓	✓
Cost-based query transformations		✓	✓	✓	✓
Allow rewrites with multiple materialized views (MVs) and/or base tables			✓	✓	✓
Adaptive cursor sharing				✓	✓
Use extended statistics to estimate selectivity				✓	✓
Use native implementation for full outer joins				✓	✓
Partition pruning using join filtering				✓	✓
Group-by-placement optimization				✓	
Null-aware antijoins				✓	

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

Summary

In this lesson, you should have learned how to:

- Describe each phase of SQL statement processing
- Discuss the need for an optimizer
- Explain the various phases of optimization
- Control the behavior of the optimizer

Practice 5: Overview

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

6

Generating and Displaying Execution Plans

Objectives

After completing this lesson, you should be able to:

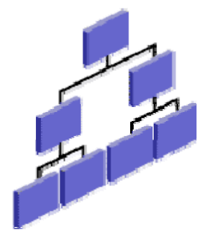
- Discuss execution plans
- Gather execution plans
- Display execution plans

Lesson Agenda

- Execution Plan
  - What Is an Execution Plan?
  - Reading an Execution Plan
  - Reviewing an Execution Plan
  - Viewing Execution Plans
- The EXPLAIN PLAN Command
- PLAN_TABLE
- AUTOTRACE
- Using the V$SQL_PLAN View
- Automatic Workload Repository
- SQL Monitoring

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.
- By using parent-child relationships, the execution plan can be displayed in a tree-like structure (text or graphical).
- An execution plan includes an access path for each table that the statement accesses and an ordering of the tables (the join order) with the appropriate join method.



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 that it must perform for most efficiently retrieving the data required by a query. The combination of row sources for a statement is called the execution plan.

The `EXPLAIN PLAN` statement displays the 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 to determine whether the optimizer selects a particular execution plan, such as nested loops joins.

Reading an Execution Plan

- The operations in the execution plan are read from right-to-left and from top-to-bottom.
- The following statement displays the `EXPLAIN PLAN`:

```
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY(NULL,
'statement_id','BASIC'));
```

- Example:

```
EXPLAIN PLAN
SET statement_id = 'ex_plan1'
FOR SELECT phone_number FROM employees
WHERE phone_number LIKE '650%';
```

```
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY(NULL,
'ex_plan1','BASIC'));
```

```
PLAN_TABLE_OUTPUT
---------------------------------------------------
Plan hash value: 1445457117

---------------------------------------------
| Id  | Operation           | Name      |
---------------------------------------------
|   0 | SELECT STATEMENT    |           |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |
---------------------------------------------

8 rows selected.
```


The plan in the slide shows the execution of a `SELECT` statement. The table `EMPLOYEES` is accessed using a full table scan.

- Every row in the table `EMPLOYEES` is accessed, and the `WHERE` clause criteria is evaluated for every row.
- The `SELECT` statement returns the rows meeting the `WHERE` clause criteria.

Reviewing the Execution Plan

By reviewing execution plans, you should be able to assess if the suboptimal plan is caused by the optimizer's wrong assumptions, calculations, or other factors:

- Drive from the table that has the most selective filter.
- Check to confirm the following:
  - The driving table has the best filter.
  - The fewest number of rows are returned to the next step.
  - The join method is appropriate for the number of rows returned.
  - Views are correctly used.
  - There are any unintentional Cartesian products.
  - Tables are accessed efficiently.

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, check to confirm 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 loops 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 scan on tables with a 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.

Where to Find Execution Plans

- `EXPLAIN PLAN` Command
- `V$SQL_PLAN` (Library Cache)
- `V$SQL_PLAN_MONITOR` (11*g*)
- `DBA_HIST_SQL_PLAN` (AWR)
- `STATS$SQL_PLAN` (Statspack)
- SQL management base (SQL plan baselines)
- SQL tuning set
- Trace files generated by `DBMS_MONITOR`
- Event 10053 trace file
- Process state dump trace file since 10*g*R2

There are many ways to retrieve execution plans in 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 Oracle 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, whereas `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 AWR infrastructure and Statspack store execution plans of top SQL statements. Plans are recorded into `DBA_HIST_SQL_PLAN` or `STATS$SQL_PLAN`.

- Plan and row source operations are dumped in trace files generated by DBMS_MONITOR.
- The SQL management base (SMB) is a part of the data dictionary that resides in the SYSAUX tablespace. It stores statement logs, plan histories, and SQL plan baselines, as well as SQL profiles.
- The event 10053, which is used to dump CBO computations, may include a plan.
- Starting with Oracle Database 10*g*, Release 2, when you dump a process state (or errorstack from a process), execution plans are included in the trace file that is generated.

Viewing Execution Plans

- The `EXPLAIN PLAN` command followed by:
  - `SELECT from PLAN_TABLE(DBMS_XPLAN.DISPLAY());`
- Oracle SQL*Plus Autotrace: `SET AUTOTRACE ON`
- `DBMS_XPLAN.DISPLAY_CURSOR()`
- `DBMS_XPLAN.DISPLAY_AWR()`
- `DBMS_XPLAN.DISPLAY_SQLSET()`
- `DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE()`

If you execute the `EXPLAIN PLAN` Oracle SQL*Plus command, you can `SELECT` from `PLAN_TABLE` to view the execution plan. There are several Oracle 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 Automatic Workload Repository (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 a 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.

Lesson Agenda

- Execution Plan
  - What Is an Execution Plan?
  - Reading an Execution Plan
  - Reviewing an Execution Plan
  - Viewing Execution Plans
- The `EXPLAIN PLAN` Command
- `PLAN_TABLE`
- `AUTOTRACE`
- Using the `V$SQL_PLAN` View
- Automatic Workload Repository
- SQL Monitoring

The EXPLAIN PLAN Command: Overview

- Generates an optimizer execution plan
- Stores the plan in PLAN_TABLE
- Does not execute the statement itself

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.

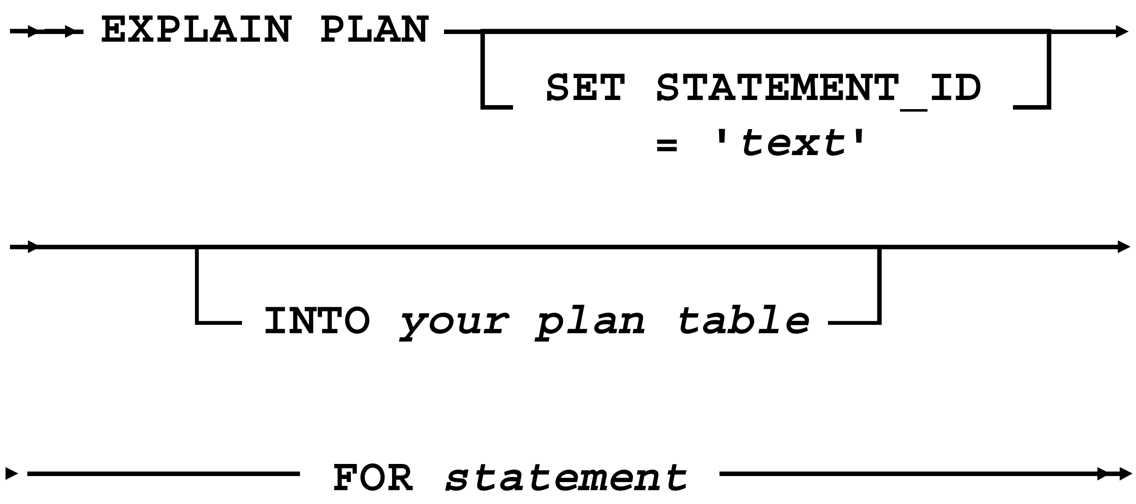
Using EXPLAIN PLAN

1. Use the EXPLAIN PLAN command to explain a SQL statement.
2. 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 that describe execution plans.

Note: You can create your own PLAN_TABLE by 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: Syntax



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

```
SQL> EXPLAIN PLAN
  2   SET STATEMENT_ID = 'demo01' FOR
  3   SELECT e.last_name, d.department_name
  4   FROM hr.employees e, hr.departments d
  5   WHERE e.department_id = d.department_id;

Explained.

SQL>
```

Note: The EXPLAIN PLAN command does not actually execute the statement.

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;
```

Note: If you run the EXPLIAN PLAN command in SQL*Plus, you will get the result as given in the slide and if you execute the statement using SQL Developer, the result is displayed as “plan SET succeeded.”

Lesson Agenda

- Execution Plan
  - What Is an Execution Plan?
  - Reading an Execution Plan
  - Reviewing an Execution Plan
  - Viewing Execution Plans
- The `EXPLAIN PLAN` Command
- `PLAN_TABLE`
- `AUTOTRACE`
- Using the `V$SQL_PLAN` View
- Automatic Workload Repository
- SQL Monitoring

PLAN_TABLE

- PLAN_TABLE:
  - This is automatically created to hold the EXPLAIN PLAN output.
  - You can create your own using utlxplan.sql.
  - The advantage is that SQL is not executed.
  - The disadvantage is it may not be the actual execution plan.
- PLAN_TABLE is hierarchical.
- Hierarchy is established with the ID and PARENT_ID columns.

Various methods are available to gather execution plans. Now, you are being 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 execution plan, the basic format and goal are the same. However, PLAN_TABLE 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 that describe the execution plans.
- It 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.
- Although 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

```
SQL> EXPLAIN PLAN SET STATEMENT_ID = 'demo01' FOR SELECT * FROM emp
  2  WHERE ename = 'KING';

Explained.

SQL> SET LINESIZE 130
SQL> SET PAGESIZE 0
SQL> select * from table(DBMS_XPLAN.DISPLAY());

Plan hash value: 3956160932
--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    38 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    38 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ENAME"='KING')
```


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 the value is null (the default), the displayed plan 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.

Here, you use the same EXPLAIN PLAN command example as in the previous slide.

```
SQL> select * from table(DBMS_XPLAN.DISPLAY(null,null,'ALL'));

Plan hash value: 3956160932
---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    38 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    38 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$1 / EMP@SEL$1
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ENAME"='KING')
Column Projection Information (identified by operation id):
-----------------------------------------------------------
 1 - "EMP"."EMPNO"[NUMBER,22], "ENAME"[VARCHAR2,10], "EMP"."JOB"[VARCHAR2,9],
     "EMP"."MGR"[NUMBER,22], "EMP"."HIREDATE"[DATE,7], "EMP"."SAL"[NUMBER,22],
     "EMP"."COMM"[NUMBER,22], "EMP"."DEPTNO"[NUMBER,22]
```

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, along 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 represents either 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

The ADVANCED format is available only from Oracle Database 10*g*, Release 2 and later versions.

```
select plan_table_output from table(DBMS_XPLAN.DISPLAY(null,null,'ADVANCED
 -PROJECTION -PREDICATE -ALIAS'));

Plan hash value: 3956160932

---------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    38 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EMP  |     1 |    38 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      FULL(@"SEL$1" "EMP"@"SEL$1")
      OUTLINE_LEAF(@"SEL$1")
      ALL_ROWS
      DB_VERSION('12.1.0.1')
      OPTIMIZER_FEATURES_ENABLE('12.1.0.1')
      IGNORE_OPTIM_EMBEDDED_HINTS
      END_OUTLINE_DATA
  */
```

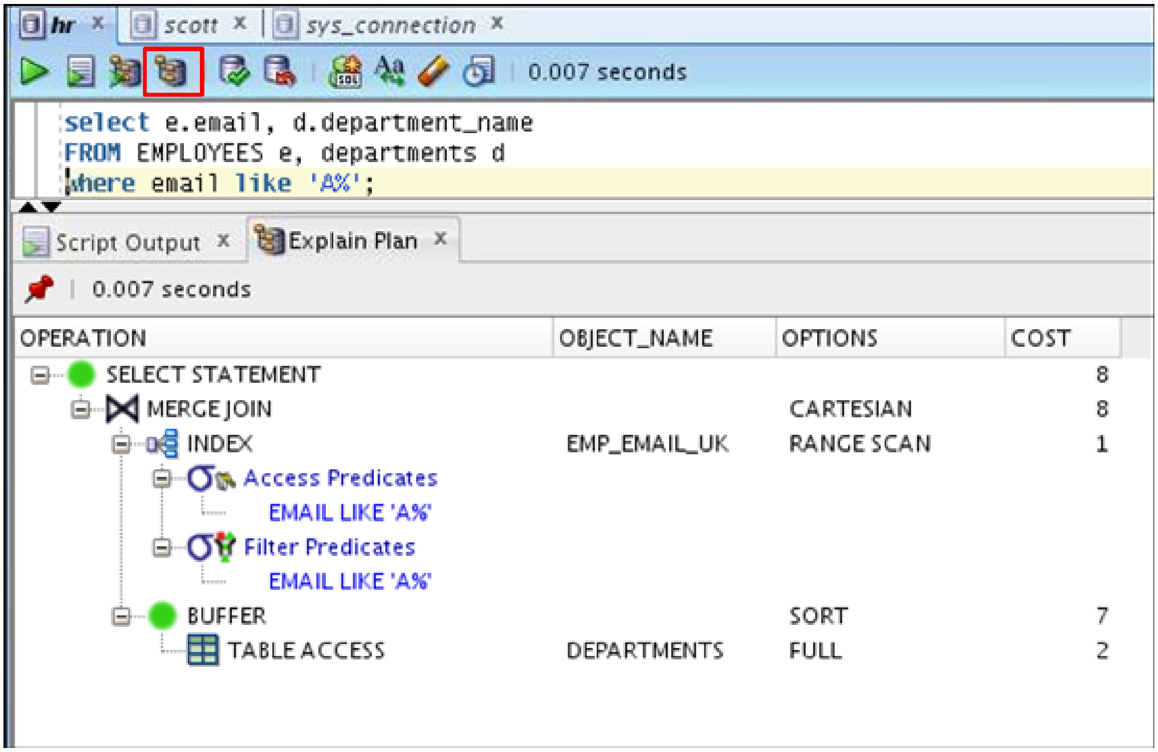
This output format includes all sections from the ALL format plus the outline data that represents a set of hints to reproduce that particular plan.

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 Oracle SQL Developer



The Explain Plan icon generates the execution plan, which you can see on 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, the access method for each table mentioned in the statement, the 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).

Quiz

An `EXPLAIN PLAN` command executes the statement and inserts the plan used by the optimizer into a table.

a. True

b. False

Answer: b

Quiz

Which of the following is NOT true about a `PLAN_TABLE`?

a. The `PLAN_TABLE` is automatically created to hold the `EXPLAIN PLAN` output.
b. You cannot create your own `PLAN_TABLE`.
c. The actual SQL command is not executed.
d. The plan in the `PLAN_TABLE` may not be the actual execution plan.

Answer: b

Lesson Agenda

- Execution Plan
  - What is an Execution Plan
  - Reading an Execution Plan
  - Reviewing an Execution Plan
  - Viewing Execution Plans
- The `EXPLAIN PLAN` Command
- `PLAN_TABLE`
- `AUTOTRACE`
- Using the `V$SQL_PLAN` View
- Automatic Workload Repository
- SQL Monitoring

AUTOTRACE

- Is an Oracle SQL*Plus and SQL Developer facility
- Was introduced with Oracle 7.3
- Needs a `PLAN_TABLE`
- Needs the `PLUSTRACE` role to retrieve statistics from some `V$` views
- Produces the execution plan and statistics by default after running the query
- May not be the execution plan used by the optimizer when it uses bind peeking (recursive `EXPLAIN PLAN`)

When running SQL statements in 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. 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.

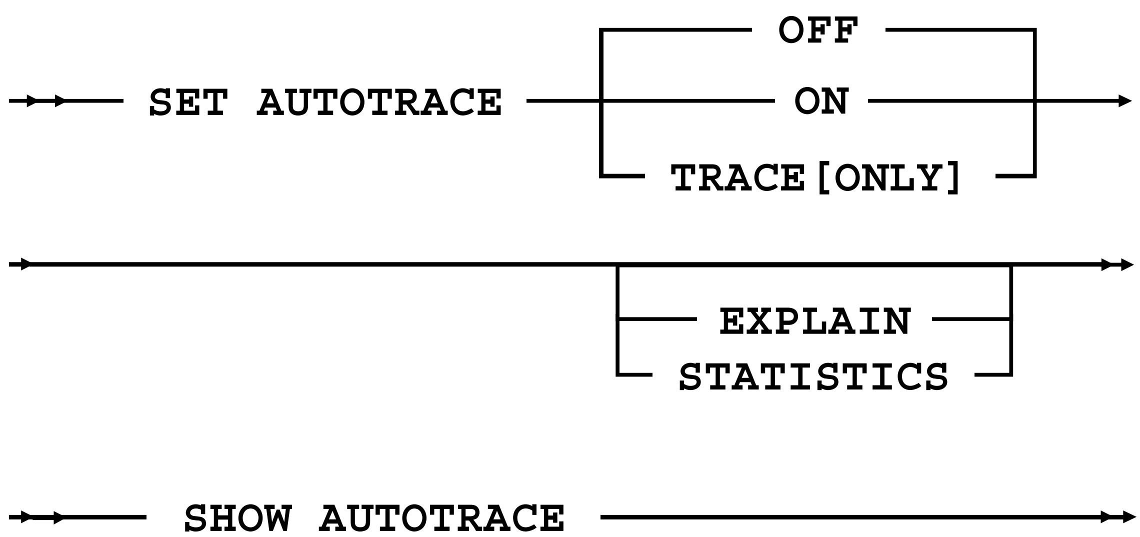
On some versions and platforms, this is run by the database creation scripts. If this is not the case on your platform, connect as `SYSDBA` and run the `plustrce.sql` script.

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 that perform 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 by 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

- To start tracing statements using AUTOTRACE:

```
SQL> set autotrace on
```

- To only display the execution plan without execution:

```
SQL> set autotrace traceonly explain
```

- To display rows and statistics:

```
SQL> set autotrace on statistics
```

- To get only the plan and the statistics (suppress rows):

```
SQL> set autotrace traceonly
```


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 it is not printed.
- SET AUTOTRACE OFF: No AUTOTRACE report is generated. This is the default.

AUTOTRACE: Statistics

```
SQL> show autotrace
autotrace OFF
SQL> set autotrace traceonly statistics
SQL> SELECT * FROM oe.products;

288 rows selected.

Statistics
--------------------------------------------------------
         15  recursive calls
          0  db block gets
        413  consistent gets
          0  physical reads
          0  redo size
     105032  bytes sent via SQL*Net to client
        753  bytes received via SQL*Net from client
         21  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        288  rows processed
```


The statistics are recorded by the server when your statement executes, and they 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 levels. Oracle Database maintains the tables used for internal processing. When Oracle Database needs to make a change to these tables, it internally generates a 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.

- `SQL*Net roundtrips to/from client` is the total number of Oracle Net messages sent to and received from the client.

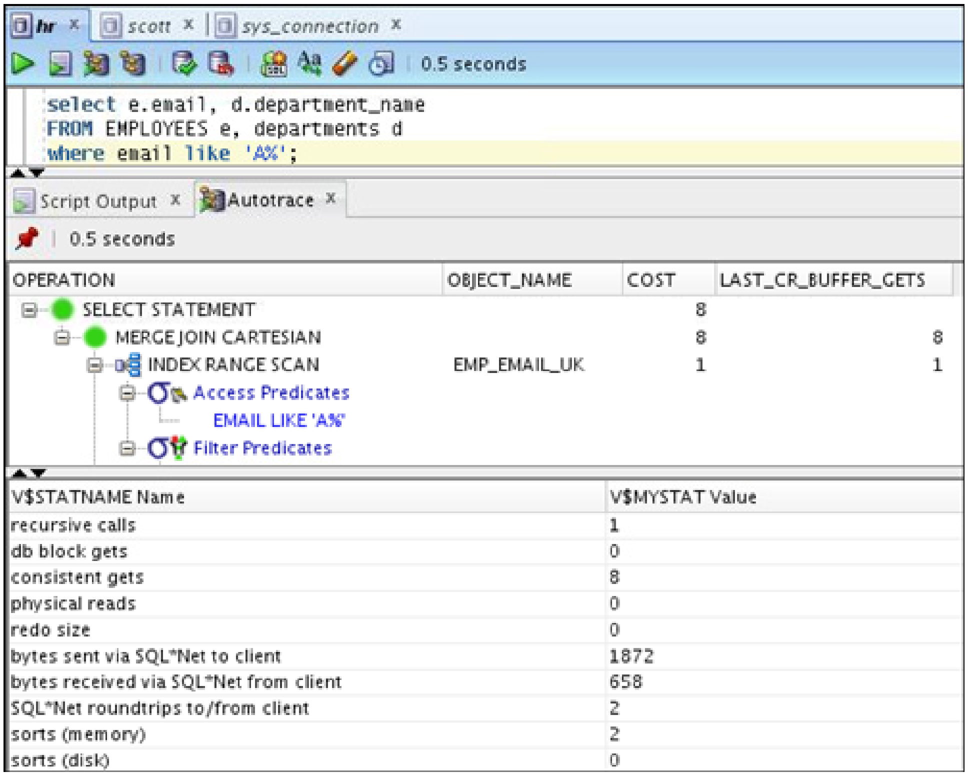
Note: The statistics printed by `AUTOTRACE` are retrieved from `V$SESSTAT`.

- `sorts (memory)` is the number of sort operations that were performed completely in memory and did not require any disk writes.
- `sorts (disk)` is the number of sort operations that required at least one disk write.
- `rows processed` is the number of rows processed during the operation.

The client referred to in the statistics is Oracle SQL*Plus. Oracle Net refers to the generic process communication between Oracle SQL*Plus and the server, regardless of whether Oracle Net is installed. You cannot change the default format of the statistics report.

Note: `db block gets` indicates reads of the current block from the database. `consistent gets` is reads of blocks that must satisfy a particular system change number (SCN). `physical reads` indicates reads of blocks from disk. `db block gets` and `consistent gets` are the two statistics that are usually monitored. They should be low compared to the number of rows retrieved. Sorts should be performed in memory rather than on disk.

`AUTOTRACE` by 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.

Quiz

A user needs to be granted some specialized privileges to generate `AUTOTRACE` statistics.

a. True

b. False

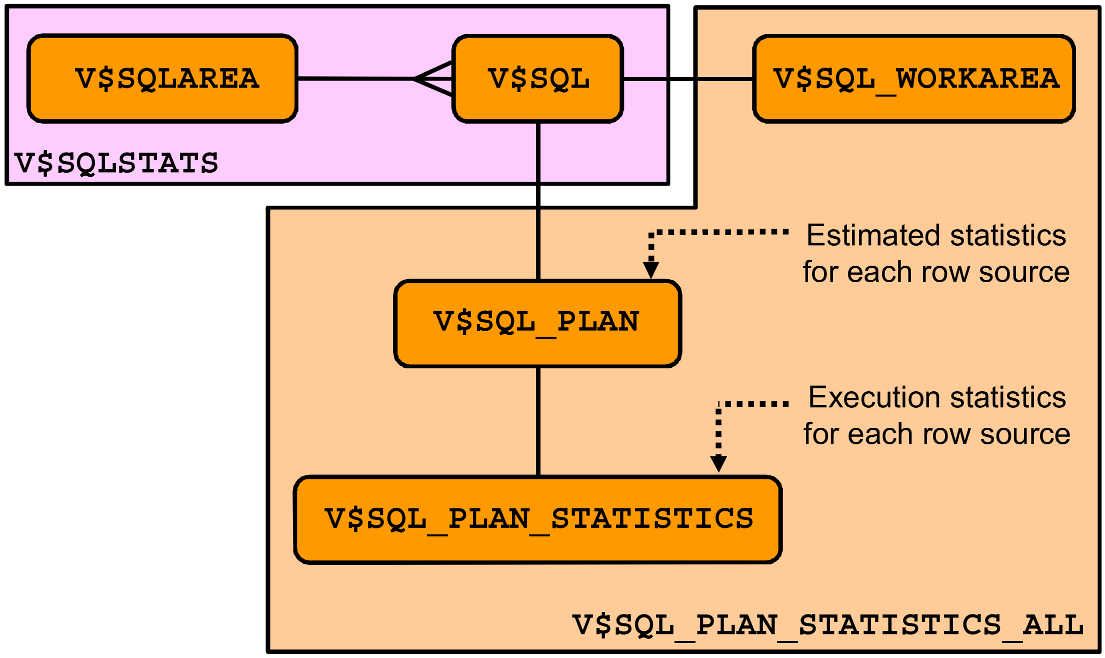
Answer: a

Lesson Agenda

- Execution Plan
  - What is an Execution Plan
  - Reading an Execution Plan
  - Reviewing an Execution Plan
  - Viewing Execution Plans
- The `EXPLAIN PLAN` Command
- `PLAN_TABLE`
- `AUTOTRACE`
- Using the `V$SQL_PLAN` View
- Automatic Workload Repository
- SQL Monitoring

ORACLE

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.

Using the V$SQL_PLAN View

- V$SQL_PLAN provides a way of examining the execution plan for cursors that are still in the library cache.
- V$SQL_PLAN is very similar to PLAN_TABLE:
  – PLAN_TABLE shows a theoretical plan that can be used if this statement were to be executed.
  – V$SQL_PLAN contains the actual plan used.
- It contains the execution plan of every cursor in the library cache (including child).
- The link to V$SQL is:
  – ADDRESS, HASH_VALUE, and CHILD_NUMBER

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, 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 the 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.

Here are a few examples:

- The same statement executed by different users has different cursors associated with it if the object that is referenced is in a different schema.
- Even if the user is the same but in a different session with different bind values, you can get a different plan.
- Using adaptive cursors, you can also have different child plans with different CHILD_NUMBERs.
- 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.

The V$SQL_PLAN Columns

HASH_VALUE	Hash value of the parent statement in the library cache
ADDRESS	Address of the handle to the parent for this cursor
CHILD_NUMBER	Child cursor number using this execution plan
POSITION	Order of processing for all operations that have the same PARENT_ID
PARENT_ID	ID of the next execution step that operates on the output of the current step
ID	Number assigned to each step in the execution plan
PLAN_HASH_VALUE	Numerical representation of the SQL plan for the cursor

Note: This is only a partial listing of the 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 cursor-specific information.

The ADDRESS, HASH_VALUE, and CHILD_NUMBER columns can be used to join with V$SQL to add 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 10*g*, 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, and its bottom 32 bits are SQL_HASH_VALUE. It is normally represented as a character string to make it more manageable.

The V$SQL_PLAN_STATISTICS View

- V$SQL_PLAN_STATISTICS provides actual execution statistics:
  - STATISTICS_LEVEL set to ALL
  - The GATHER_PLAN_STATISTICS hint
- V$SQL_PLAN_STATISTICS_ALL enables side-by-side comparisons of optimizer estimates with the actual execution statistics.

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.

V$SQL_PLAN contains the execution plan information for each child cursor loaded in the library cache. The ADDRESS, HASH_VALUE, and CHILD_NUMBER columns can be used to join with V$SQL to add the child cursor–specific information.

V$SQL_PLAN_STATISTICS provides execution statistics at the row source level for each child cursor. The ADDRESS and HASH_VALUE columns can be used to join with V$SQLAREA to locate the parent cursor. The ADDRESS, HASH_VALUE, and CHILD_NUMBER columns can be used to join with V$SQL to locate the child cursor that is using this area.

V$SQL_PLAN_STATISTICS_ALL contains memory usage statistics for the row sources that use SQL memory (sort or hash join). This view concatenates information in V$SQL_PLAN with the execution statistics from V$SQL_PLAN_STATISTICS and V$SQL_WORKAREA.

V$SQL_WORKAREA displays information about the work areas used by SQL cursors. Each SQL statement stored in the shared pool has one or more child cursors that are listed in the V$SQL view. V$SQL_WORKAREA lists all work areas needed by these child cursors. V$SQL_WORKAREA can be joined with V$SQLAREA on (ADDRESS, HASH_VALUE) and with V$SQL on (ADDRESS, HASH_VALUE, CHILD_NUMBER).

You can use this view to find answers to the following questions:

- What are the top 10 work areas that require the most cache area?
- For work areas allocated in the AUTO mode, what percentage of work areas run using maximum memory?

V$SQLSTATS displays basic performance statistics for SQL cursors, with each row representing the data for a unique combination of SQL text and optimizer plan (that is, unique combination of SQL_ID and PLAN_HASH_VALUE). The column definitions for columns in V$SQLSTATS are identical to those in the V$SQL and V$SQLAREA views. However, the V$SQLSTATS view differs from V$SQL and V$SQLAREA in that it is faster, more scalable, and has greater data retention (the statistics may still appear in this view, even after the cursor has been aged out of the shared pool). Note that V$SQLSTATS contains a subset of columns that appear in V$SQL and V$SQLAREA.

Querying V$SQL_PLAN

```
SELECT PLAN_TABLE_OUTPUT FROM
TABLE(DBMS_XPLAN.DISPLAY_CURSOR('d71cudzrcqsa2'));
```

```
PLAN_TABLE_OUTPUT
SELECT e.last_name, d.department_name FROM hr.employees e,
hr.departments d WHERE  e.department_id =d.department_id

Plan hash value: 4254995515

----------------------------------------------------------------------------------------------
| Id  | Operation                  | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                   |       |       |     2 (100)|          |
|   1 |  NESTED LOOPS              |                   |       |       |            |          |
|   2 |   NESTED LOOPS             |                   |   106 |  2862 |     2   (0)| 00:00:01 |
|   3 |    VIEW                    | index$_join$_001  |   107 |  1177 |     2   (0)| 00:00:01 |
|*  4 |     HASH JOIN              |                   |       |       |            |          |
|   5 |      INDEX FAST FULL SCAN  | EMP_DEPARTMENT_IX |   107 |  1177 |     1   (0)| 00:00:01 |
|   6 |      INDEX FAST FULL SCAN  | EMP_NAME_IX       |   107 |  1177 |     1   (0)| 00:00:01 |
|*  7 |    INDEX UNIQUE SCAN       | DEPT_ID_PK        |     1 |       |     0   (0)|          |
|   8 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS     |     1 |    16 |     0   (0)|          |
----------------------------------------------------------------------------------------------
```


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

CHILD_NUMBER is the child number of the cursor to display. If it is not supplied, the execution plan of all cursors that matches the supplied SQL_ID parameter is displayed. CHILD_NUMBER can be specified only if SQL_ID is specified.

The FORMAT parameter controls the level of detail for the plan. In addition to standard values (BASIC, TYPICAL, SERIAL, ALL, and ADVANCED), there are additional supported values to display runtime statistics for the cursor:

- **IOSTATS:** Assuming that basic plan statistics are collected when SQL statements are executed (either by using the GATHER_PLAN_STATISTICS hint or by setting the statistics_level parameter to ALL), this format shows I/O statistics for ALL (or only for LAST) executions of the cursor.
- **MEMSTATS:** Assuming that PGA memory management is enabled (that is, the pga_aggregate_target parameter is set to a nonzero value), this format allows you to display memory management statistics (for example, execution mode of the operator, how much memory was used, and number of bytes spilled to disk). These statistics apply only to memory-intensive operations, such as hash joins, sort, or some bitmap operators.
- **ALLSTATS:** This is a shortcut for 'IOSTATS MEMSTATS'.
- **LAST:** By default, plan statistics are shown for all executions of the cursor. The LAST keyword can be specified to see only the statistics for the last execution.

Note: Starting with 10*g*R2, a useful format option (+PEEKED_BINDS) can be used to display the values of bind variables when using dbms.xplan.display_cursor().

For example:

```
select plan_table_output
from table(dbms_xplan.display_cursor(null,null,'typical+PEEKED_BINDS'));
```

Lesson Agenda

- Execution Plan
  - What is an Execution Plan
  - Reading an Execution Plan
  - Reviewing an Execution Plan
  - Viewing Execution Plans
- The `EXPLAIN PLAN` Command
- `PLAN_TABLE`
- `AUTOTRACE`
- Using the `V$SQL_PLAN` View
- Automatic Workload Repository
- SQL Monitoring

Automatic Workload Repository

- Collects, processes, and maintains performance statistics for problem-detection and self-tuning purposes
- Includes the following statistics:
  - Object statistics
  - Time-model statistics
  - Some system and session statistics
  - Active Session History (ASH) statistics
- Automatically generates snapshots of the performance data

AWR is part of the intelligent infrastructure introduced with Oracle Database. This infrastructure is used by many components, such as Automatic Database Diagnostic Monitor (ADDM) for analysis. 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 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, and buffer gets
- ASH statistics, representing the history of recent sessions

The database automatically generates snapshots of the performance data once every hour and collects the statistics in the workload repository. The data in the snapshot interval is then analyzed by ADDM. ADDM compares the differences between snapshots to determine which SQL statements to capture based on the effect on the system load. This reduces the number of SQL statements that need to be captured over time.

Note: By using PL/SQL packages, such as `DBMS_WORKLOAD_REPOSITORY` or Enterprise Manager (EM), you can manage the frequency and retention period of the SQL that is stored in AWR.

Important AWR Views

- V$ACTIVE_SESSION_HISTORY
- V$ metric views
- DBA_HIST views:
  - DBA_HIST_ACTIVE_SESS_HISTORY
  - DBA_HIST_BASELINE DBA_HIST_DATABASE_INSTANCE
  - DBA_HIST_SNAPSHOT
  - DBA_HIST_SQL_PLAN
  - DBA_HIST_WR_CONTROL

You can view AWR data on EM screens or in AWR reports. You can also view the statistics directly from the following views:

The V$ACTIVE_SESSION_HISTORY view displays active database session activity, sampled once every second.

V$ metric views provide metric data to track the performance of the system. The metric views are organized into various groups, such as event, event class, system, session, service, file, and tablespace metrics. These groups are identified in the V$METRICGROUP view.

The DBA_HIST views contain historical data stored in the database. This group of views includes:

- DBA_HIST_ACTIVE_SESS_HISTORY: Displays the history of the contents of the sampled in-memory active session history for recent system activity
- DBA_HIST_BASELINE: Displays information about the baselines captured in the system
- DBA_HIST_DATABASE_INSTANCE: Displays information about the database environment
- DBA_HIST_SNAPSHOT: Displays information about the snapshots in the system
- DBA_HIST_SQL_PLAN: Displays SQL execution plans
- DBA_HIST_WR_CONTROL: Displays the settings for controlling AWR

Comparing Execution Plans by Using AWR

- Identify a problem SQL statement and retrieve `SQL_ID`.

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

SQL_ID        SQL_TEXT
------------- ------------------------------------------
454rug2yva18w select /* example */ * from …
```

- Retrieve all execution plans stored for a particular `SQL_ID`.

```
SELECT PLAN_TABLE_OUTPUT
FROM TABLE (DBMS_XPLAN.DISPLAY_AWR('454rug2yva18w'));

Plan hash value: 4179021502

---------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |       |       |     6 (100)|          |
|   1 |  HASH JOIN         |             |    11 |   968 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPARTMENTS |    11 |   220 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMPLOYEES   |   107 |  7276 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------------
```


At times, users report slow performance of a SQL statement, that it had been running fine, but not today. It is not an uncommon situation. You can use the `DBMS_XPLAN.DISPLAY_AWR()` function to display all stored plans in AWR to see if a plan has changed. 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 …
```

3. Using the `SQL_ID`, verify that this statement has been captured in the `DBA_HIST_SQLTEXT` dictionary view. If the query does not return rows, it indicates that the statement has not yet been loaded in AWR.

```
SQL> SELECT SQL_ID, SQL_TEXT FROM dba_hist_sqltext WHERE SQL_ID ='
454rug2yva18w';
no rows selected
```

You can take a manual AWR snapshot rather than wait for the next snapshot (which occurs every hour). Then check to see if it has been captured in `DBA_HIST_SQLTEXT`:

```
SQL> exec dbms_workload_repository.create_snapshot;

PL/SQL procedure successfully completed.

SQL> SELECT SQL_ID, SQL_TEXT FROM dba_hist_sqltext WHERE SQL_ID ='
454rug2yva18w';
SQL_ID          SQL_TEXT
-------------   ------------------------------
454rug2yva18w   select /* example */ * from …
```

4. Use the `DBMS_XPLAN.DISPLAY_AWR ()` function to retrieve the execution plan:

```
SQL>SELECT PLAN_TABLE_OUTPUT FROM TABLE
(DBMS_XPLAN.DISPLAY_AWR('454rug2yva18w'));
```

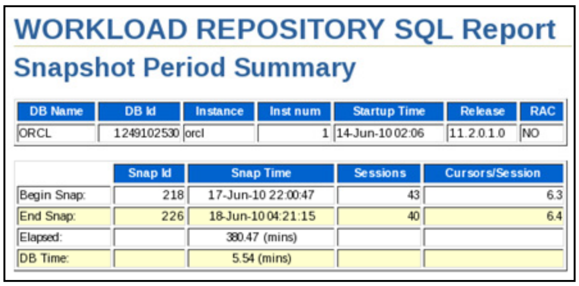
Note: If you have SQLT available in your environment, you can compare the historical execution plans for a particular SQL statement easily.

For more information, see MOS note 215187.1, “SQLT (SQLXPLAIN).”

Generating SQL Reports from AWR Data

```
SQL> @$ORACLE_HOME/rdbms/admin/awrsqrpt

Specify the Report Type …
Would you like an HTML report, or a plain text report?
Specify the number of days of snapshots to choose from
Specify the Begin and End Snapshot Ids …
Specify the SQL Id …
Enter value for sql_id: dvza55c7zu0yv
Specify the Report Name …
```

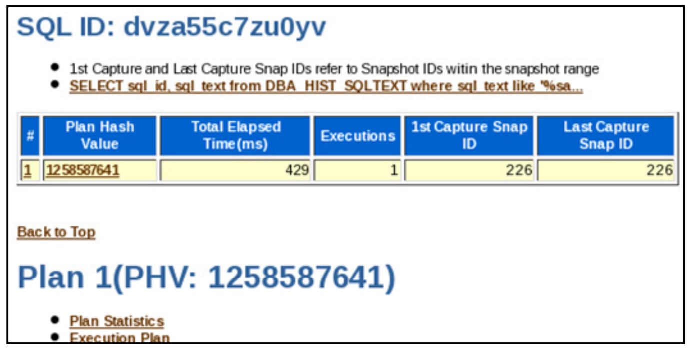


WORKLOAD REPOSITORY SQL Report

Snapshot Period Summary

DB Name	DB Id	Instance	Inst num	Startup Time	Release	RAC
ORCL	1249102530	orcl	1	14-Jun-10 02:06	11.2.0.1.0	NO

	Snap Id	Snap Time	Sessions	Cursors/Session
Begin Snap:	218	17-Jun-10 22:00:47	43	6.3
End Snap:	226	18-Jun-10 04:21:15	40	6.4
Elapsed:		380.47 (mins)		
DB Time:		5.54 (mins)		



SQL ID: dvza55c7zu0yv

- 1st Capture and Last Capture Snap IDs refer to Snapshot IDs witin the snapshot range
- SELECT sql_id, sql_text from DBA_HIST_SQLTEXT where sql_text like '%sa...

#	Plan Hash Value	Total Elapsed Time(ms)	Executions	1st Capture Snap ID	Last Capture Snap ID
1	1258587641	429	1	226	226

Back to Top

Plan 1(PHV: 1258587641)

- Plan Statistics
- Execution Plan

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, the following displays the plan information for a `SQL_ID` in AWR:

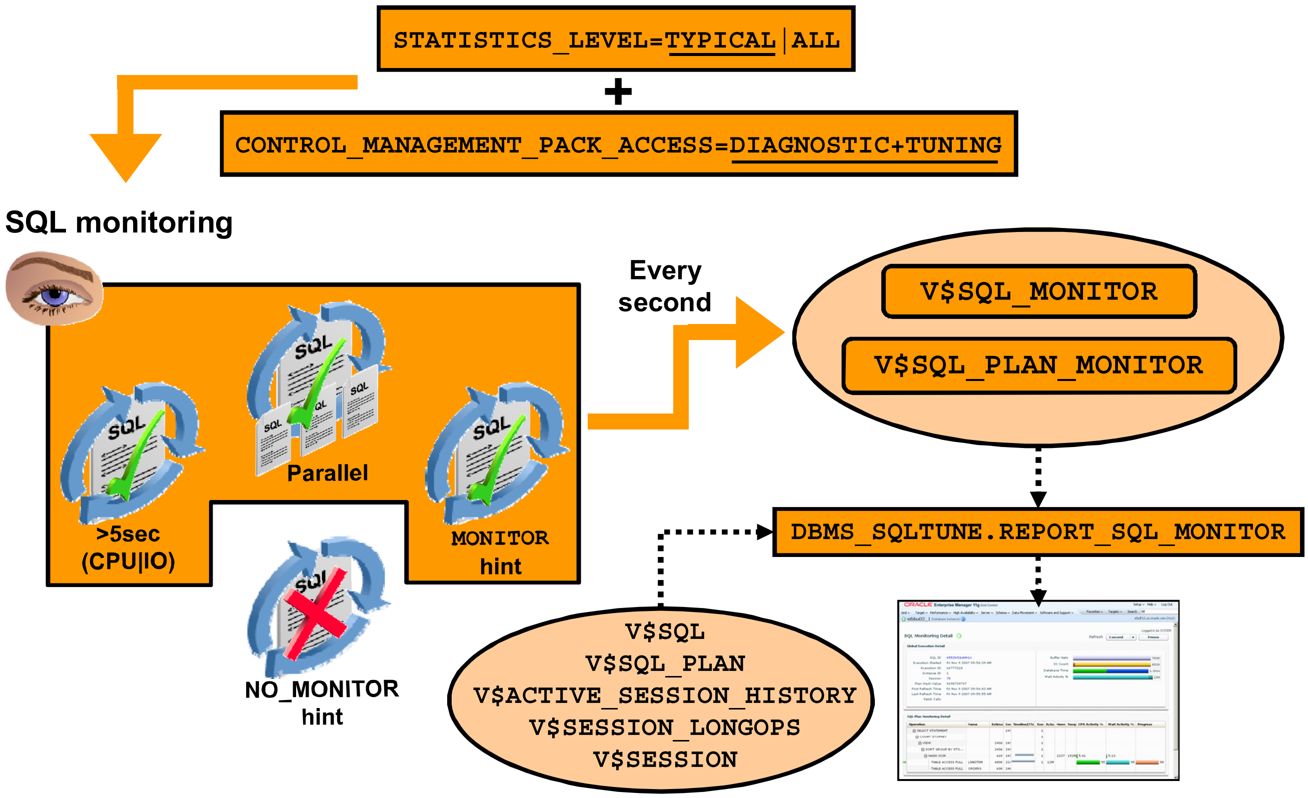
```
select * from table(dbms_xplan.display_awr('dvza55c7zu0yv'));
```

You can retrieve the appropriate value for a SQL statement of interest by querying `SQL_ID` in the `DBA_HIST_SQLTEXT` column.

Lesson Agenda

- Execution Plan
  - What is an Execution Plan
  - Reading an Execution Plan
  - Reviewing an Execution Plan
  - Viewing Execution Plans
- The `EXPLAIN PLAN` Command
- `PLAN_TABLE`
- `AUTOTRACE`
- Using the `V$SQL_PLAN` View
- Automatic Workload Repository
- SQL Monitoring

SQL Monitoring: Overview



The SQL monitoring feature is enabled by default when the `STATISTICS_LEVEL` initialization parameter is set either to `ALL` or to `TYPICAL` (the default value).

Additionally, the `CONTROL_MANAGEMENT_PACK_ACCESS` parameter must be set to `DIAGNOSTIC+TUNING` (the default value) because SQL monitoring is a feature of the Oracle Database Tuning Pack.

By default, SQL monitoring is automatically started when a SQL statement runs parallel, or when it has consumed at least five seconds of the CPU or I/O time in a single execution.

As mentioned, SQL monitoring is active by default. However, two statement-level hints are available to force or prevent a SQL statement from being monitored. To force SQL monitoring, use the `MONITOR` hint. To prevent the hinted SQL statement from being monitored, use the `NO_MONITOR` hint.

You can monitor the statistics for SQL statement execution by using the `V$SQL_MONITOR` and `V$SQL_PLAN_MONITOR` views.

After monitoring is initiated, an entry is added to the dynamic performance `V$SQL_MONITOR` view. This entry tracks the key performance metrics collected for the execution, including the elapsed time, CPU time, number of reads and writes, I/O wait time, and various other wait times. These statistics are refreshed in near real time as the statement executes, generally once every second.

After the execution ends, monitoring information is not deleted immediately, but is kept in the V$SQL_MONITOR view for at least one minute. The entry is eventually deleted so that its space can be reclaimed as new statements are monitored.

The V$SQL_MONITOR and V$SQL_PLAN_MONITOR views can be used in conjunction with the following views to get additional information about the execution that is monitored:

VSQL, VSQL_PLAN, V$ACTIVE_SESSION_HISTORY, V$SESSION_LONGOPS, and V$SESSION

Alternatively, you can use the SQL monitoring report to view SQL monitoring data.

The SQL monitoring report is also available in a GUI version through EM and Oracle SQL Developer.

SQL Monitoring Report: Example

```
SQL> set long 10000000
SQL> set longchunksize 10000000
SQL> set linesize 200
SQL> select dbms_sqltune.report_sql_monitor from dual;

SQL Monitoring Report

SQL Text
--------------------------
select count(*) from sales

Global Information
 Status                :  EXECUTING
 Instance ID           :  1
 Session ID            :  125
 SQL ID                :  fazrk33ng71km
 SQL Execution ID      :  16777216
 Plan Hash Value       :  1047182207
 Execution Started     :  02/19/2008 21:01:18
 First Refresh Time    :  02/19/2008 21:01:22
 Last Refresh Time     :  02/19/2008 21:01:42
-------------------------------------------------------------
| Elapsed |   Cpu   |    IO     |   Other   | Buffer | Reads |
| Time(s) | Time(s) | Waits(s)  | Waits(s)  |  Gets  |       |
-------------------------------------------------------------
|      22 |    3.36 |      0.01 |        19 |   259K |  199K |
-------------------------------------------------------------
```

In a different session

```
SQL> select count(*) from sales;
```


In this example, it is assumed that you SELECT from SALES from a different session than the one used to print the SQL monitoring report.

The DBMS_SQLTUNE.REPORT_SQL_MONITOR function accepts several input parameters to specify the execution, level of detail in the report, and report type (TEXT, HTML, or XML). By default, a text report is generated for the last execution that was monitored if no parameters are specified, as shown in the slide example.

After the SELECT statement is started, and while it executes, you print the SQL monitoring report from a second session.

From the report, you can see that the SELECT statement executes currently.

The Global Information section gives you some important information. To uniquely identify two executions of the same SQL statement, a composite key called an execution key is generated. This execution key consists of three attributes, each corresponding to a column in V$SQL_MONITOR:

- SQL identifier to identify the SQL statement (SQL_ID)
- An internally generated identifier to ensure that this primary key is truly unique (SQL_EXEC_ID)
- A start execution time stamp (SQL_EXEC_START)

The report also shows you some important statistics calculated so far.

SQL Monitoring Report: Example

```
SQL Plan Monitoring Details
==================================================================================
| Id     |         Operation       | Name  |  Rows   | Cost  |   Time     | Start  |
|        |                         |       | (Estim) |       | Active(s)  | Active |
==================================================================================
|      0 | SELECT STATEMENT        |       |         | 78139 |            |        |
|      1 |   SORT AGGREGATE        |       |       1 |       |            |        |
|  ->  2 |    TABLE ACCESS FULL    | SALES |  53984K | 78139 |         23 |     +1 |
|        |                         |       |         |       |            |        |
==================================================================================


================================================================================
Starts |   Rows    | Activity  |            Activity Detail            | Progress |
       | (Actual)  | (percent) |              (sample #)               |          |
     1 |           |           |                                       |          |
     1 |           |           |                                       |          |
     1 |   42081K  |    100.00 | Cpu (4)                               | 74%      |
================================================================================
```


The report then displays the execution path currently used by your statement. SQL monitoring gives you the display of the current operation that executes in the plan. This enables you to detect parts of the plan that are the most time consuming, so that you can focus your analysis on those parts. The running operation is marked by an arrow in the `Id` column of the report.

The `Time Active(s)` column shows how long the operation has been active (the delta in seconds between the first and the last active time).

The `Start Active` column shows, in seconds, when the operation in the execution plan started, relative to the SQL statement execution start time. In the report in the slide, the table access full operation at Id 2 was the first to start (+1s Start Active) and ran for the first 23 seconds so far.

The `Starts` column shows the number of times the operation in the execution plan was executed.

The `Rows (Actual)` column indicates the number of rows produced, and the `Rows (Estim)` column shows the estimated cardinality from the optimizer.

The Activity (percent) and Activity Detail (sample #) columns are derived by joining the V$SQL_PLAN_MONITOR and V$ACTIVE_SESSION_HISTORY views. Activity (percent) shows the percentage of database time consumed by each operation of the execution plan. Activity Detail (sample#) shows the nature of that activity (such as CPU or wait event).

In this report, the Activity Detail (sample #) column shows that most of the database time, 100 percent, is consumed by operation Id 2 (TABLE ACCESS FULL of SALES). So far, this activity consists of four samples, which are attributed only to CPU.

The last column, Progress, shows progress monitoring information for the operation from the V$SESSION_LONGOPS view. In this report, it shows that, so far, the TABLE ACCESS FULL operation is 74 percent complete. This column appears in the report only after a certain amount of time and only for the instrumented row sources.

Note: Not shown by this particular report, the Memory and Temp columns indicate the amount of memory and temporary space consumed by the corresponding operation of the execution plan.

Quiz

After monitoring is initiated, an entry is added to the ________view. This entry tracks the key performance metrics collected for an execution.

a. `V$SQL_MONITOR`
b. `V$PLAN_MONITOR`
c. `ALL_SQL_MONITOR`
d. `ALL_SQL_PLAN_MONITOR`

Answer: b

Summary

In this lesson, you should have learned how to:

- Gather execution plans
- Display execution plans
- Interpret execution plans

Practice 6: Overview

This practice covers the following topics:

- Using different techniques to extract execution plans
- Using SQL monitoring

7

Interpreting Execution Plans and Enhancements

Objectives

After completing this lesson, you should be able to:

- Interpret execution plans
- Discuss adaptive optimizations

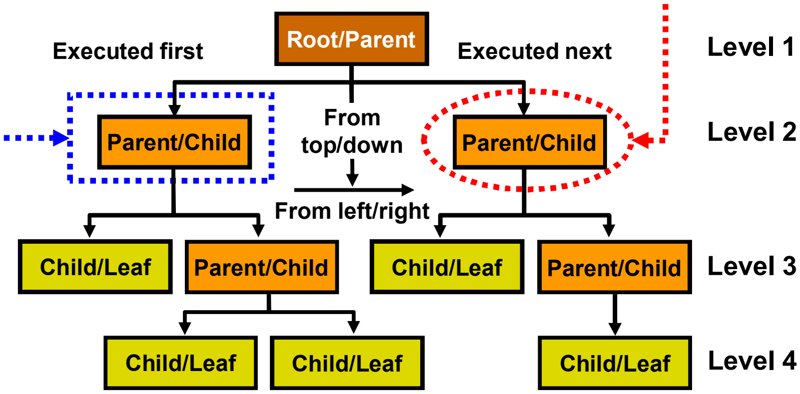
Lesson Agenda

- Interpreting a Serial Execution Plan
- Reading More Complex Execution Plans
- Looking Beyond Execution Plans
- Adaptive Query Optimizations: Overview
- Adaptive Plans: Join Method
  - Adaptive Join Method: Example
  - Displaying the default, final, and full adaptive plans
- Adaptive Plans: Parallel Distribution Method
  - Parallel Distribution Method: Example

Interpreting a Serial Execution Plan

```
id= 1 (pid= )              root/parent
id= 2 (pid=1) (pos=1)     parent/child
id= 3 (pid=2) (pos=1)       child/leaf
id= 4 (pid=2) (pos=2)     parent/child
id= 5 (pid=4) (pos=1)       child/leaf
id= 6 (pid=4) (pos=2)       child/leaf
id= 7 (pid=1) (pos=2)      parent/child
id= 8 (pid=7) (pos=1)       child/leaf
id= 9 (pid=7) (pos=2)     parent/child
id=10 (pid=9) (pos=1)       child/leaf
```

Transform it into a tree.



Explain plan output is a representation of a tree of row sources.

Each step (line in the execution plan or node in the tree) represents a row source.

The explain plan utility indents nodes to indicate that they are the children of the parent above it.

The order of the nodes under the parent indicates the order of execution of the nodes within that level. If two steps are indented at the same level, the first one is executed first.

In the tree format, the leaf at the left on each level of the tree is where the execution starts.

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.

In `PLAN_TABLE` and `V$SQL_PLAN`, the important elements to retrieve the tree structure are the `ID`, `PARENT_ID`, and `POSITION` columns. In a trace file, these columns correspond to the `id`, `pid`, and `pos` fields, respectively.

One way to read an execution plan is by converting it into a graph that has a tree structure. You can start from the top, with `id=1`, which is the root node in the tree. Next, you must find the operations that feed this root node. This is accomplished by operations that have `parent_id` or `pid` with value 1.

Note: The course focuses on serial plans and does not discuss parallel execution plans.

To draw the plan as a tree, perform the following:

1. Take the ID with the lowest number and place it at the top.
2. Look for rows that have a process identifier (PID; parent) equal to this value.
3. Place these in the tree below the Parent according to their POSITION values from the lowest to the highest, ordered from left to right.
4. After finding all the IDs for a parent, move down to the next ID and repeat the process, finding new rows with the same PID.

The first thing to determine in an explain plan is which node is executed first. The method in the slide explains this, but sometimes with complicated plans, it is difficult to do this and it is also difficult to follow the steps through to the end. Large plans are exactly the same as smaller ones, but with more entries. The same basic rules apply. You can always collapse the plan to hide a branch of the tree that does not consume much of the resources.

Standard tree interpretation:

1. Start at the top.
2. Move down the tree to the left until you reach the left node. This is executed first.
3. Look at the siblings of this row source. These row sources are executed next.
4. After the children are executed, the parent is executed next.
5. Now that this parent and its children are completed, work back up the tree and look at the siblings of the parent row source and its parents. Execute as before.
6. Move back up the tree until all row sources are exhausted.

If you remember the few basic rules of explain plans, you can read most plans easily after you gain experience.

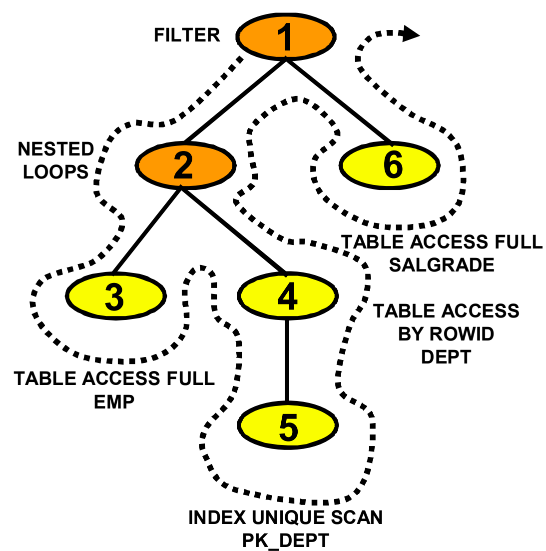
Execution Plan Interpretation: Example 1

```
SELECT /*+ RULE */ ename,job,sal,dname
FROM emp,dept
WHERE dept.deptno=emp.deptno and not exists(SELECT *
                                            FROM salgrade
                                            WHERE emp.sal between losal and hisal);
```

```
--------------------------------------------------
| Id  | Operation                    | Name     |
--------------------------------------------------
|   0 | SELECT STATEMENT             |          |
|*  1 |  FILTER                      |          |
|   2 |   NESTED LOOPS               |          |
|   3 |    TABLE ACCESS FULL         | EMP      |
|   4 |    TABLE ACCESS BY INDEX ROWID| DEPT     |
|*  5 |     INDEX UNIQUE SCAN        | PK_DEPT  |
|*  6 |   TABLE ACCESS FULL          | SALGRADE |
--------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter( NOT EXISTS
     (SELECT 0 FROM "SALGRADE" "SALGRADE" WHERE
      "HISAL">=:B1 AND "LOSAL"<=:B2))
   5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
   6 - filter("HISAL">=:B1 AND "LOSAL"<=:B2)
```



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 `ROWID`s from the index are used to look up 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`.

Note that children are executed before parents. So, although structures for joins must be set up before child execution, the children are notated as executed first. Probably the easiest way is to consider it as the order in which execution is completed; therefore, for the `NESTED LOOPS` join at `ID=2`, the two children {`ID=3` and `ID=4` (together with its child)} must have completed their execution before `ID=2` can be completed.

Execution Plan Interpretation: Example 1

```
SQL> alter session set statistics_level=ALL;

Session altered.

SQL> select /*+ RULE to make sure it reproduces 100% */ ename,job,sal,dname
from emp,dept where dept.deptno = emp.deptno and not exists (select * from salgrade
where emp.sal between losal and hisal);

no rows selected

SQL> select * from table(dbms_xplan.display_cursor(null,null,'TYPICAL IOSTATS
LAST'));

SQL_ID  274019myw3vuf, child number 0
-------------------------------------
…
Plan hash value: 1175760222
--------------------------------------------------------------------------------
| Id  | Operation                      | Name       | Starts | A-Rows | Buffers |
--------------------------------------------------------------------------------
|*  1 |  FILTER                        |            |      1 |      0 |      61 |
|   2 |   NESTED LOOPS                 |            |      1 |     14 |      25 |
|   3 |    TABLE ACCESS FULL           | EMP        |      1 |     14 |       7 |
|   4 |    TABLE ACCESS BY INDEX ROWID| DEPT       |     14 |     14 |      18 |
|*  5 |     INDEX UNIQUE SCAN          | PK_DEPT    |     14 |     14 |       4 |
|*  6 |   TABLE ACCESS FULL            | SALGRADE   |     12 |     12 |      36 |
--------------------------------------------------------------------------------
…
```


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.

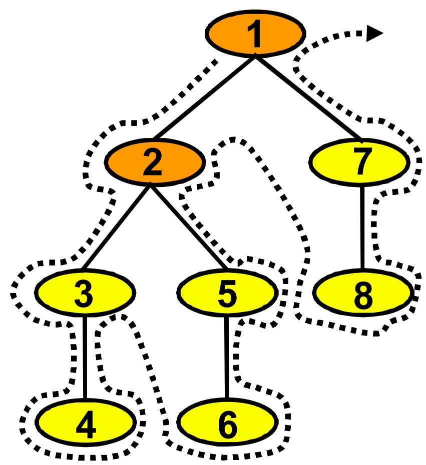
If you observe the statistics closely, the `TABLE ACCESS FULL` operation on the `EMP` table (`ID=3`) is started once. However, operations from `ID 5` and `4` are started 14 times and once for each `EMP` row. At this step (`ID=2`), the system gets all `ENAME`, `SAL`, `JOB`, and `DNAME`.

The system must now 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 needs to be done only 12 times in this case, because the system does the check for each distinct salary at run time, and there are 12 distinct salaries in the `EMP` table.

Execution Plan Interpretation: Example 2

```
SQL> select /*+ USE_NL(d) use_nl(m) */ m.last_name as dept_manager
  2  ,       d.department_name
  3  ,       l.street_address
  4  from   hr.employees m   join
  5          hr.departments d on (d.manager_id = m.employee_id)
  6          natural join
  7          hr.locations l
  8  where  l.city = 'Seattle';
```

```
0    SELECT STATEMENT
1 0    NESTED LOOPS
2 1     NESTED LOOPS
3 2      TABLE ACCESS BY INDEX ROWID LOCATIONS
4 3       INDEX RANGE SCAN            LOC_CITY_IX
5 2      TABLE ACCESS BY INDEX ROWID DEPARTMENTS
6 5       INDEX RANGE SCAN            DEPT_LOCATION_IX
7 1     TABLE ACCESS BY INDEX ROWID  EMPLOYEES
8 7      INDEX UNIQUE SCAN            EMP_EMP_ID_PK
```



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 `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 that produces data but that does not consume any data. 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 `ROWID`s, which are used to look up 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 `ROWID`s, which are used to look up in the `DEPARTMENTS` table in `ID=5`.

4. After the children operation, the parent operation is next. The NESTED LOOPS join at ID=2 is executed next, bringing together the underlying data.
5. Now that this parent and its children are completed, go back up the tree, and look at the siblings of the parent row source and their parents. Execute as before. The sibling of ID=2 at the same level in the plan is ID=7. This has a child ID=8, which is executed first. The index unique scan produces ROWIDs, which are used to look up in the EMPLOYEES table in ID=7.
6. Move back up the plan until all row sources are exhausted. Finally the plan is brought together with NESTED LOOPS at ID=1, which passes the results back to ID=0.
7. The execution order is: 4 – 3 – 6 – 5 – 2 – 8 – 7 – 1 – 0.

A complete description of this plan is as follows:

The inner nested loops is executed first using LOCATIONS as the driving table, using an index access on the CITY column. This is because you searched for departments only in Seattle.

The result is joined with the DEPARTMENTS table, using an index on the LOCATION_ID join column; the result of this first join operation is the driving row source for the second nested loops join.

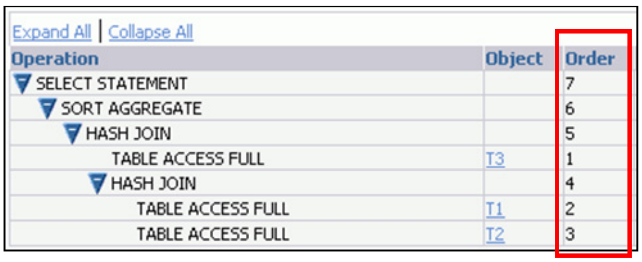
The second join probes the index on the EMPLOYEE_ID column of the EMPLOYEES table. The system can do that because it knows (from the first join) the employee ID of all managers of departments in Seattle. Note that this is a unique scan because it is based on the primary key.

Finally, the EMPLOYEES table is accessed to retrieve the last name.

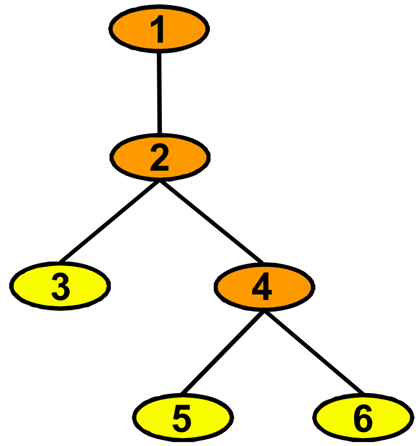
Execution Plan Interpretation: Example 3

```
select /*+ ORDERED USE_HASH(b) SWAP_JOIN_INPUTS(c) */ max(a.i)
from t1 a, t2 b, t3 c
where a.i = b.i and a.i = c.i;
```

```
0    SELECT STATEMENT
1     SORT AGGREGATE
2 1    HASH JOIN
3 2     TABLE ACCESS FULL T3
4 2     HASH JOIN
5 4      TABLE ACCESS FULL T1
6 4      TABLE ACCESS FULL T2
```



Operation	Object	Order
SELECT STATEMENT		7
SORT AGGREGATE		6
HASH JOIN		5
TABLE ACCESS FULL	T3	1
HASH JOIN		4
TABLE ACCESS FULL	T1	2
TABLE ACCESS FULL	T2	3



Join order is: T1 - T2 - T3

ORACLE

For the execution plan in the slide, try to find the order in which the plan is executed and deduce the join order (order in which the system joins tables). Again, `ID` is in the first column and `PID` is in the second column. The position is reflected by the indentation. It is important to recognize the join order of an execution plan so that you can find your plan in a 10053 event trace file.

An interpretation of this plan is as follows:

1. The system first hashes the `T3` table (Operation `ID=3`) into memory.
2. Then it hashes the `T1` table (Operation `ID=5`) into memory.
3. Then the scan of the `T2` table begins (Operation `ID=6`).
4. The system picks a row from `T2` and probes `T1` (`T1.i=T2.i`).
5. If the row survives, the system probes `T3` (`T1.i=T3.i`).
6. If the row survives, the system sends it to the next operation.
7. The system outputs the maximum value from the previous result set.

In conclusion, the execution order is: `3 – 5 – 6 – 4 – 2 – 1`.

The join order is: `T1 – T2 – T3`

You can also use Enterprise Manager to understand execution plans, especially because it displays the Order column.

Note: A special hint was used to make sure that `T3` would be first in the plan.

Execution Plan Interpretation: Example 4

```
select /*+ GATHER_PLAN_STATISTICS */ *
from oe.inventories where
warehouse_id = 1;

select plan_table_output
  from table(dbms_xplan.display_cursor(format=> 'allstats last'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------
SQL_ID  6260b91rbnn4n, child number 0
-------------------------------------
select /*+ GATHER_PLAN_STATISTICS */ * from oe.inventories where
warehouse_id = 1

Plan hash value: 791134270

-------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |      1 |        |     36 |00:00:00.01 |       9 |
|   1 |  TABLE ACCESS BY INDEX ROWID| INVENTORIES    |      1 |    124 |     36 |00:00:00.01 |       9 |
|*  2 |   INDEX RANGE SCAN           | INVENTORY_IX   |      1 |    124 |     36 |00:00:00.01 |       5 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("WAREHOUSE_ID"=1)
```

Compare actual rows returned by each operation (A-Rows) with the Optimizer estimate (E-Rows).

See the execution plan in the slide. You can use the `GATHER_PLAN_STATISTICS` hint in the SQL statement to automatically obtain more comprehensive runtime statistics, such as the actual cardinality for each operation. Using the hint, you can compare the actual rows returned by each operation (A-Rows) with the Optimizer estimate (E-Rows). In the example, the actual and estimated rows are quite different. You might need to check column statistics, called histogram, to resolve the issue. Histogram is covered in the lesson titled “Introduction to Optimizer Statistics Concepts.”

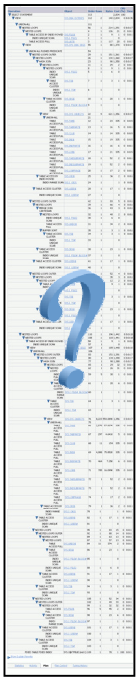
Note: Using the hint has an impact on the execution time of a SQL statement, so you should use it only for the purpose of analysis.

Lesson Agenda

- Interpreting a Serial Execution Plan
- Reading More Complex Execution Plans
- Looking Beyond Execution Plans
- Adaptive Query Optimizations: Overview
- Adaptive Plans: Join Method
  - Adaptive Join Method: Example
  - Displaying the default, final, and full adaptive plans
- Adaptive Plans: Parallel Distribution Method
  - Parallel Distribution Method: Example

Reading More Complex Execution Plans

```
SELECT owner , segment_name , segment_type
FROM dba_extents
WHERE file_id = 1
AND   123213 BETWEEN block_id AND block_id + blocks -1;
```



Expand All | Collapse All

Operation	Object	Order	Rows	Bytes	Cost	CPU (%)	Time
SELECT STATEMENT		113			2,834	100	
VIEW	SYS.DBA_EXTENTS	112	2	140	2,834	0	0:0:35
UNION-ALL		111					
NESTED LOOPS		56	1	214	1,391	0	0:0:17
NESTED LOOPS		110	1	196	1,442	0	0:0:18

Collapse using indentation and focus on operations consuming most resources.

In the slide, the plan on the left comes from the query on the data dictionary. It is so long that it is very difficult to apply the previous method to interpret it and locate the first operation.

You can always collapse a plan to make it readable, as illustrated by the collapsed plan on the right. As shown, this is easy to do when using the EM or Oracle SQL Developer graphical interface. You can clearly see that this plan is a `UNION ALL` of two branches. Your knowledge of the data dictionary enables you to understand that the two branches correspond to dictionary-managed and locally managed tablespaces. Your knowledge of your database enables you to know that there are no dictionary-managed tablespaces.

Therefore, if there is a problem, it must be on the second branch. To get confirmation, you must look at the plan information and execution statistics of each row source to locate the part of the plan that consumes most resources. Then, you just need to expand the branch that you want to investigate (where time is being spent). To use this method, you must look at the execution statistics that are generally found in `V$SQL_PLAN_STATISTICS` or in the `TKProf` Reports generated from trace files. For example, for each parent operation, `tkprof` cumulates the time it takes to execute itself plus the sum of the time taken for all of its child operations.

Lesson Agenda

- Interpreting a Serial Execution Plan
- Reading More Complex Execution Plans
- Looking Beyond Execution Plans
- Adaptive Query Optimizations: Overview
- Adaptive Plans: Join Method
  - Adaptive Join Method: Example
  - Displaying the default, final, and full adaptive plans
- Adaptive Plans: Parallel Distribution Method
  - Parallel Distribution Method: Example

Looking Beyond Execution Plans

- An execution plan alone cannot tell you whether a plan is good or not.
- It may need additional testing and tuning:
  - SQL Tuning Advisor
  - SQL Access Advisor
  - SQL Performance Analyzer
  - SQL Monitoring
  - Tracing
  - Adaptive Execution Plans

An execution plan alone cannot differentiate between well-tuned statements and those that perform poorly. For example, an `EXPLAIN PLAN` output that shows that a statement uses an index does not necessarily mean that the statement runs efficiently. Sometimes, indexes can be extremely inefficient.

It is best to use `EXPLAIN PLAN` to determine an access plan, and then later prove that it is the optimal plan through testing. When evaluating a plan, you should examine the statement's actual resource consumption.

The rest of this course is intended to show you various methods to achieve this.

Quiz

```
SELECT * FROM oe.inventories WHERE warehouse_id = 1;

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |   124 |  1240 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| INVENTORIES  |   124 |  1240 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | INVENTORY_IX |   124 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

SELECT count(*) FROM oe.inventories WHERE warehouse_id = 1;

COUNT(*)
--------
      36
```

Q1. According to the execution plan, is the number of rows returned in step 2 quite accurate?

Q2. What is the selectivity of the predicate and computed cardinality (total rows in the table: 1112 rows, NDV: 9)?

Q3. Has the optimizer made a good estimation?

When you review the execution plan, many components can be assessed, such as cardinality, access method, join method, join type, join order, and partition pruning. This quiz focuses on the computed cardinality.

The value of the Rows column in step 2 is based on the following calculation:

- Computed cardinality = total rows (1112) * selectivity (1/9), which is about 124.
- Only 36 rows meet the condition (`warehouse_id= 1`).

The optimizer makes a wrong assumption about many factors. This example shows the issue caused by data distribution. A discussion on how to resolve this issue is covered in the lesson titled "Other Optimizer Operators."

Lesson Agenda

- Interpreting a Serial Execution Plan
- Reading More Complex Execution Plans
- Looking Beyond Execution Plans
- Adaptive Query Optimization: Overview
  - Adaptive Plans: Join Method
  - Adaptive Join Method: Example
  - Displaying the default, final, and full adaptive plans
  - Adaptive Plans: Parallel Distribution Method
  - Parallel Distribution Method: Example

Adaptive Query Optimization: Overview

- Adaptive Query Optimization is a set of capabilities that enable the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics.
- Adaptive Query Optimization is extremely helpful when existing statistics are not sufficient to generate an optimal plan.
- The database uses adaptive plans when `OPTIMIZER_FEATURES_ENABLE` is set to 12.1.0.1 or later, and the `OPTIMIZER_ADAPTIVE_REPORTING_ONLY` initialization parameter is set to the default of `FALSE`.

Adaptive Query Optimization enables the optimizer to automatically adapt a poorly performing execution plan at run time and prevent a bad plan from being chosen on subsequent executions.

The optimizer will instrument its chosen plan so that at run time, it can be detected if the optimizer's estimates are bad. Then the plan can be automatically adapted to the actual conditions.

An adaptive plan is referred to as a plan that changes after optimization when optimizer estimates prove inaccurate.

The optimizer can adapt plans based on statistics that are collected during statement execution. All adaptive mechanisms can execute a plan that differs from the plan, which was originally determined during hard parse. An adaptive plan improves the ability of the query-processing engine (compilation and execution) to generate better execution plans.

Adaptive plans are useful because the optimizer occasionally picks a suboptimal default plan because of cardinality misestimate. The ability to adapt the plan at run time based on actual execution statistics results in a more optimal final plan. After choosing the final plan, the optimizer uses it for subsequent executions, thus ensuring that the suboptimal plan is not reused.

Adaptive Plans: Join Method

- The join method decision is deferred until run time.
  - A default plan is computed using available statistics.
  - Alternate subplans are pre-computed and stored in the cursor.
  - Statistic collectors are inserted at key points in the plan.
- The final decision is based on the statistics collected during execution.
  - The default plan and subplans have valid ranges for the statistics collected.
  - If the statistics prove to be out of range, the subplans are swapped.
  - This requires buffering near the swap point to avoid returning rows to the user.

An adaptive plan is an execution plan that has different built-in plan options. During the first execution, before a specific subplan becomes active, the optimizer makes a final decision about which option to use. The optimizer bases its choice on observations made during the execution up to this point. Adaptive plan enables the final plan for a statement to differ from the default plan, thereby potentially improving query performance.

A subplan is a portion of a plan that the optimizer can switch to as an alternative at run time.

During statement execution, the statistics collector buffers a portion of rows. Portions of the plan preceding the statistics collector can have alternate subplans, each of which is valid for a subset of possible values returned by the collector.

Often the set of values for a subplan is a range. If a statistic falls in the range of valid values for a subplan that was not the default plan, the optimizer chooses the alternative subplan. After the optimizer chooses a subplan, buffering is disabled. The statistics collector stops collecting rows, and passes them through instead. On subsequent executions of the child cursor, the optimizer disables buffering, and chooses the same final plan.

With dynamic plans, the execution plan adapts to the optimizer's poor plan choices, and correct decisions can be made during the first execution.

Note: The new column in `V$SQL IS_RESOLVED_DYNAMIC_PLAN` indicates if the final plan was not the default plan. Information found via dynamic plans is persisted as SQL plan directives.

Adaptive Join Method: Example

```
SELECT product_name
FROM order_items o, product_information p
WHERE o.unit_price = 15
AND o.quantity > 1
AND p.product_id = o.product_id
```

Query: Find all products with a unit price of 15 that were sold more than once.

An adaptive plan for this statement shows two possible plans:

- Nested Loops
- Hash Join

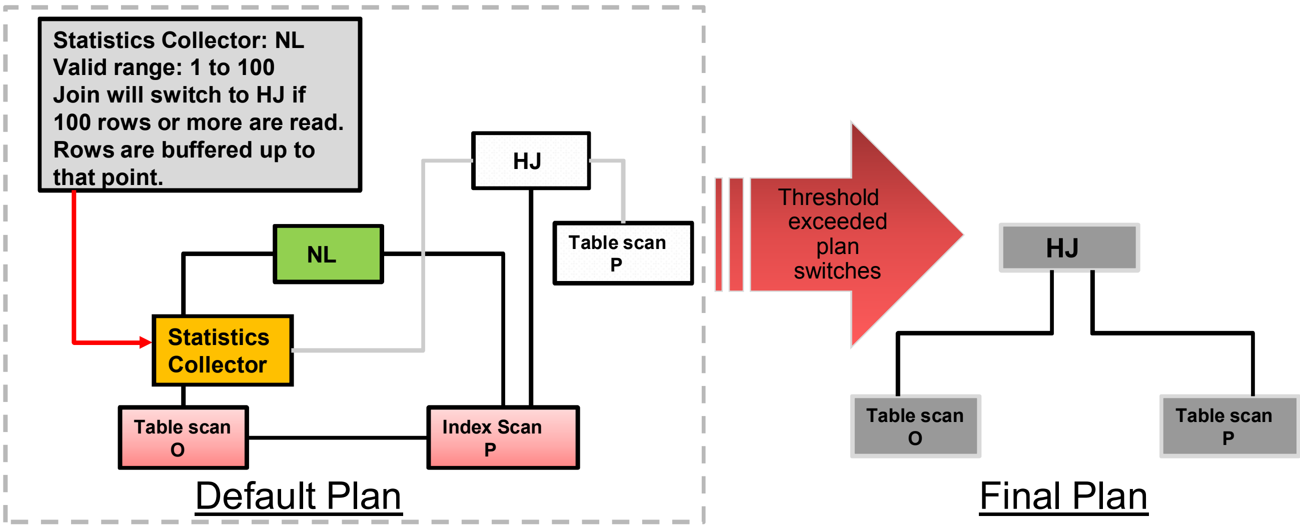
The example in the slide shows a join of the `order_items` and `product_information` tables.

An adaptive plan for this statement shows two possible plans, one with a nested loops join and the other with a hash join.

Adaptive Join Method: Working

Alternate subplans are pre-computed and stored in the cursor.

- In this case, a nested loops join is replaced by a hash join if the number of rows processed exceeds a valid range.



A nested loops join is preferable if the database can avoid scanning a significant portion of `product_information` because its rows are filtered by the join predicate. If few rows are filtered, however, scanning the right table in a hash join is preferable.

The graphic in the slide shows the adaptive process. For the query in the previous slide, the adaptive portion of the default plan contains two subplans, each of which uses a different join method. The optimizer automatically determines when each join method is optimal, depending on the cardinality of the left side of the join.

The statistics collector buffers enough rows coming from the `order_items` table to determine which join method to use. If the row count is below the threshold determined by the optimizer, the optimizer chooses the nested loops join; otherwise, the optimizer chooses the hash join. In this case, the row count coming from the `order_items` table is above the threshold, so the optimizer chooses a hash join for the final plan, and disables buffering.

Displaying the Default Plan

- An explain plan command always shows a default plan.
- The following example shows a nested loops join as the default plan.
- However, there is no statistics collector shown in the plan.

```
SQL> explain plan for
  2  select /*+ gather_plan_statistics*/ product_name
  3  from order_items  o, product_information p
  4  where o.unit_price = 15
  5    and o.quantity > 1
  6    and p.product_id = o.product_id;

Explained.

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

PLAN_TABLE_OUTPUT
------------------------------------------------------------------
Plan hash value: 389188998

------------------------------------------------------------------
| Id  | Operation                     | Name                   |
------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                        |
|   1 |  NESTED LOOPS                 |                        |
|   2 |   NESTED LOOPS                |                        |
|*  3 |    TABLE ACCESS FULL          | ORDER_ITEMS            |
|*  4 |    INDEX UNIQUE SCAN          | PRODUCT_INFORMATION_PK |
|   5 |   TABLE ACCESS BY INDEX ROWID| PRODUCT_INFORMATION    |
------------------------------------------------------------------
```

ORACLE

A statement that has a dynamic plan could show two possible plans: one plan with a nested loops before execution and another with a hash join after execution. This information is displayed in the plan output table.

The plan in the slide shows a nested loops join before execution.

Displaying the Final Plan

- After the statement has completed, use `DBMS_XPLAN.DISPLAY_CURSOR` to see the final plan that was selected.
- The following example shows a nested loops join as the default plan.
- Again the statistics collector is not visible in the plan.

```
SQL> select * from table(dbms_xplan.display_cursor('1b07h971622mm'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
SQL_ID  1b07h971622mm, child number 0
-------------------------------------
select /*+ gather_plan_statistics*/ product_name from order_items  o,
product_information p where o.unit_price = 15   and o.quantity > 1
and p.product_id = o.product_id

Plan hash value: 2326172301

--------------------------------------------------------------------------------
| Id  | Operation          | Name                | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                     |       |       |     7 (100)|
|*  1 |  HASH JOIN         |                     |     4 |   128 |     7   (0)|
|*  2 |   TABLE ACCESS FULL| ORDER_ITEMS         |     4 |    48 |     3   (0)|
|   3 |   TABLE ACCESS FULL| PRODUCT_INFORMATION |     1 |    20 |     1   (0)|
--------------------------------------------------------------------------------
```

ORACLE

After the optimizer determines the final plan, `DBMS_XPLAN.DISPLAY_CURSOR` displays the hash join.

The plan changed because the optimizer realized during execution that the number of rows actually returned from the `order_items` table is much larger than expected. Multiple single-column predicates on the `order_items` table caused the initial cardinality estimate to be incorrect. The mis-estimation cannot be corrected by extended statistics because one of the predicates is a non-equality predicate.

Displaying the Full Adaptive Plan

The new adaptive optimization section is shown when the format parameter `+adaptive` is set.

```
SQL> select* from table(dbms_xplan.display_cursor(format=>' +adaptive'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
SQL_ID  9ht2704s2s5s9, child number 0
-------------------------------------
select /*+ monitor*/ product_name from order_items o,
product_information p where o.unit_price = 15 and quantity > 1 and
p.product_id = o.product_id

Plan hash value: 1553478007

------------------------------------------------------------------------------------------------
|   Id  | Operation                      | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
|     0 | SELECT STATEMENT               |                         |       |       |     8 (100)|          |
|  *  1 |  HASH JOIN                     |                         |    13 |   416 |     8   (0)| 00:00:01 |
|-    2 |   NESTED LOOPS                 |                         |       |       |            |          |
|-    3 |    NESTED LOOPS                |                         |    13 |   416 |     8   (0)| 00:00:01 |
|-    4 |     STATISTICS COLLECTOR       |                         |       |       |            |          |
|  *  5 |      TABLE ACCESS FULL         | ORDER_ITEMS             |    13 |   156 |     3   (0)| 00:00:01 |
|- *  6 |     INDEX UNIQUE SCAN          | PRODUCT_INFORMATION_PK  |       |       |            |          |
|-    7 |    TABLE ACCESS BY INDEX ROWID| PRODUCT_INFORMATION     |     1 |    20 |     5   (0)| 00:00:01 |
|     8 |   TABLE ACCESS FULL            | PRODUCT_INFORMATION     |   288 |  5760 |     5   (0)| 00:00:01 |
------------------------------------------------------------------------------------------------


PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")
   5 - filter(("O"."UNIT_PRICE"=15 AND "QUANTITY">1))
   6 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
   - this is an adaptive plan (rows marked '-' are inactive)

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
   - 1 Sql Plan Directive used for this statement
```

ORACLE

View adaptive reports by using `DBMS_XPLAN.DISPLAY_CURSOR`. The format argument passed to `DBMS_XPLAN.DISPLAY_CURSOR` must include `+ADAPTIVE`.

Adaptive Plans: Indicator in V$SQL

- A new column in V$SQL is IS_RESOLVED_ADAPTIVE_PLAN.
- This column shows whether all the adaptive parts of a plan have been resolved to the final plan.
- The resolved plan is then used for subsequent executions.
- Statistics collectors and buffering are disabled.

This column shows whether all the adaptive parts of a plan have been resolved to the final plan. After the plan is resolved, the plan hash value and the plan displayed by DBMS_XPLAN will not change till the end of execution.

The values for this column are:

- **NULL:** If the plan is not adaptive
- **Y:** If the plan is fully resolved
- **N:** If the plan is not yet fully resolved

Lesson Agenda

- Interpreting a Serial Execution Plan
- Reading More Complex Execution Plans
- Looking Beyond Execution Plans
- Adaptive Optimizations: Overview
  - Adaptive Plans: Join Method
  - Adaptive Join Method: Example
  - Displaying the default, final, and full adaptive plans
  - Adaptive Plans: Parallel Distribution Method
  - Parallel Distribution Method: Example

Adaptive Plans: Parallel Distribution Method

- Parallel execution requires data redistribution to perform operations such as parallel sorts, aggregations, and joins.
- Data distribution is necessary when parallel execution is used.
- The decision on distribution method is based on operation and expected number of rows.
- A new adaptive distribution method is HYBRID-HASH.
  - Statistics collectors are inserted in front of the parallel server process on the left side of the join.
  - If the actual number of rows is less than a threshold, there is a switch from hash distribution to broadcast.

Parallel execution requires data redistribution to perform operations such as parallel sorts, aggregations, and joins. Oracle Database can use many different data distribution methods. The database chooses the method based on the number of rows to be distributed and the number of parallel server processes in the operation.

For example, consider the following alternative cases:

Many parallel server processes distribute a few rows.

- The database may choose the broadcast distribution method. In this case, the entire result set is sent to all the parallel server processes.

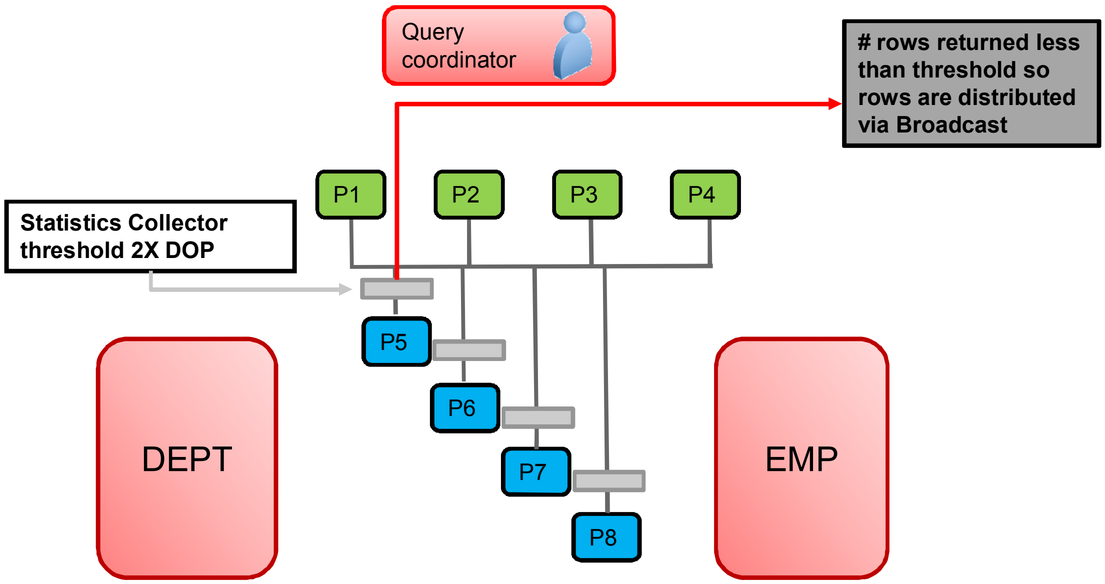
A few parallel server processes distribute many rows.

- If a data skew is encountered during data redistribution, it could adversely affect the performance of the statement. The database is more likely to pick a hash distribution to ensure that each parallel server process receives an equal number of rows.

The HYBRID-HASH distribution technique involves adaptive parallel data distribution that does not decide the final data distribution method until execution time. The optimizer inserts statistics collectors in front of the parallel server processes on the producer side of the operation. If the actual number of rows is less than a threshold, defined as twice the degree of parallelism chosen for the operation, the data distribution method switches from hash to broadcast. Otherwise, the data distribution method is a hash.

Parallel Distribution Method: Example

- If the total number of rows scanned is less than the threshold, switch to Broadcast otherwise use HASH.



The diagram in the slide shows a HYBRID-HASH join between the departments and employees tables. A statistics collector is inserted in front of the parallel server processes that are scanning the departments table. The distribution method is based on the runtime statistics. In this example, the number of rows exceeds the threshold of twice the degree of parallelism, so the optimizer chooses a broadcast technique for the departments table.

HYBRID-HASH Method: Example

HYDRID-HASH is used instead of the traditional HASH distribution.

```
EXPLAIN PLAN FOR SELECT /*+ parallel(8) full(e) full(d) */
department_name, sum(salary) FROM employees e, departments d WHERE
d.department_id=e.department_id
GROUP BY department_name;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY());
```

```
---------------------------------------------------------------------------------------------------------------------
| Id  | Operation                        | Name        | Rows  | Bytes | Cost (%CPU)| Time     |    TQ  |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                 |             |    27 |   621 |     4   (0)| 00:00:01 |        |      |            |
|   1 |  PX COORDINATOR                  |             |       |       |            |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)            | :TQ10003    |    27 |   621 |     4   (0)| 00:00:01 |  Q1,03 | P->S | QC (RAND)  |
|   3 |    HASH GROUP BY                 |             |    27 |   621 |     4   (0)| 00:00:01 |  Q1,03 | PCWP |            |
|   4 |     PX RECEIVE                   |             |    27 |   621 |     4   (0)| 00:00:01 |  Q1,03 | PCWP |            |
|   5 |      PX SEND HASH                | :TQ10002    |    27 |   621 |     4   (0)| 00:00:01 |  Q1,02 | P->P | HASH       |

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------------------------------------------
|   6 |       HASH GROUP BY              |             |    27 |   621 |     4   (0)| 00:00:01 |  Q1,02 | PCWP |            |
|*  7 |        HASH JOIN                 |             |   106 |  2438 |     4   (0)| 00:00:01 |  Q1,02 | PCWP |            |
|   8 |         PX RECEIVE               |             |    27 |   432 |     2   (0)| 00:00:01 |  Q1,02 | PCWP |            |
|   9 |          PX SEND HYBRID HASH     | :TQ10000    |    27 |   432 |     2   (0)| 00:00:01 |  Q1,00 | P->P | HYBRID HASH|
|  10 |           STATISTICS COLLECTOR   |             |       |       |            |          |  Q1,00 | PCWC |            |
|  11 |            PX BLOCK ITERATOR     |             |    27 |   432 |     2   (0)| 00:00:01 |  Q1,00 | PCWC |            |
|  12 |             TABLE ACCESS FULL    | DEPARTMENTS |    27 |   432 |     2   (0)| 00:00:01 |  Q1,00 | PCWP |            |
|  13 |         PX RECEIVE               |             |   107 |   749 |     2   (0)| 00:00:01 |  Q1,02 | PCWP |            |
|  14 |          PX SEND HYBRID HASH (SKEW)| :TQ10001  |   107 |   749 |     2   (0)| 00:00:01 |  Q1,01 | P->P | HYBRID HASH|
|  15 |           PX BLOCK ITERATOR      |             |   107 |   749 |     2   (0)| 00:00:01 |  Q1,01 | PCWC |            |
|  16 |            TABLE ACCESS FULL     | EMPLOYEES   |   107 |   749 |     2   (0)| 00:00:01 |  Q1,01 | PCWP |            |

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------
```

ORACLE

The threshold is 16 (2 X 8). Because the number of rows (27) is greater than the threshold, the optimizer chooses a HYBRID-HASH.

Quiz

Identify the characteristics that must be supported by an application that is designed for SQL execution efficiency.

a. Use of concurrent connections to the database
b. Use of cursors so that SQL statements are parsed once and executed multiple times
c. For data warehousing queries, use of cursor sharing so that you can get the best plan

Answer: b

Quiz

The decision on distribution method is based on operation and expected number of rows.

a. True

b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Interpret execution plans
- Use adaptive optimizations

Practice 7: Overview

These practices cover using adaptive plans.

8

Optimizer: Table and Index Access Paths

Objectives

After completing this lesson, you should be able to:

- Describe the SQL operators for tables and indexes
- List the possible access paths
- Describe common observations

This lesson helps you to understand the execution plans that use operators related to table and index access methods.

Lesson Agenda

- Row Source Operations
- Main Structures and Access Paths
- Table Access Paths
- Indexes: Overview
  - Normal B*-tree Indexes
  - Index Scans
  - Index-Organized Tables
  - Bitmap Indexes
  - Bitmap Operations
  - Composite Indexes
  - Invisible Index: Overview
- Common Observations

Row Source Operations

- A row source is a set of rows returned by a step in the execution plan.
- Row sources can be classified as:
  - Unary operations
    - Access Path
  - Binary operations
    - Joins
  - N-ary operations
- An access path is a way in which a query retrieves rows from a row source.

ORACLE

A row source is a set of rows returned by a step in the execution plan. The row source can be a table, a view, or the result of a join or grouping operation.

You can classify row sources as follows:

- Unary operations: Operations that act on only one input, such as an access path
- Binary operations: Operations that act on two inputs, such as joins
- N-ary operations: Operations that act on several inputs, such as a relational operator

Access paths are ways in which data is retrieved from a database. In general, index access paths should be used for statements that retrieve a small subset of table rows, whereas full scans are more efficient when accessing a large portion of a table. Online transaction processing (OLTP) applications, which consist of short-running SQL statements with high selectivity, are often characterized by the use of index access paths. Decision Support System (DSS), on the other hand, tend to use partitioned tables and perform full scans of the relevant partitions.

Lesson Agenda

- Row Source Operations
- Main Structures and Access Paths
- Table Access Paths
- Indexes: Overview
  - Normal B*-tree Indexes
  - Index Scans
  - Index-Organized Tables
  - Bitmap Indexes
  - Bitmap Operations
  - Composite Indexes
  - Invisible Index: Overview
- Common Observations

Main Structures and Access Paths

Structures	Access Paths
Tables	1. `Full Table Scan` 2. `ROWID Scan` 3. `Sample Table Scan`
Indexes	4. `Index Scan (Unique)` 5. `Index Scan (Range)` 6. `Index Scan (Full)` 7. `Index Scan (Fast Full)` 8. `Index Scan (Skip)` 9. `Index Scan (Index Join)` 10. `Using Bitmap Indexes` 11. `Combining Bitmap Indexes`

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, whereas full scans are more efficient when accessing a large portion of a 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 are not 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 Database physically stores all rows for each department from both the `EMP` and `DEPT` tables in the same data blocks.

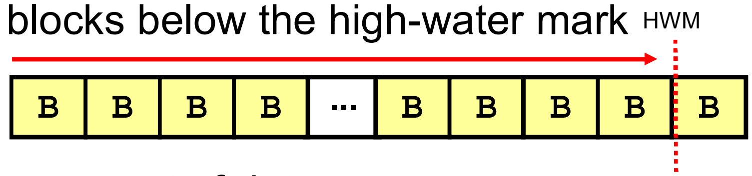
Hash clusters are single-table clusters in which rows with the same hash-key values are stored together. A mathematical hash function is used to select the location of a row within the cluster. All rows with the same key value are stored together on disk.

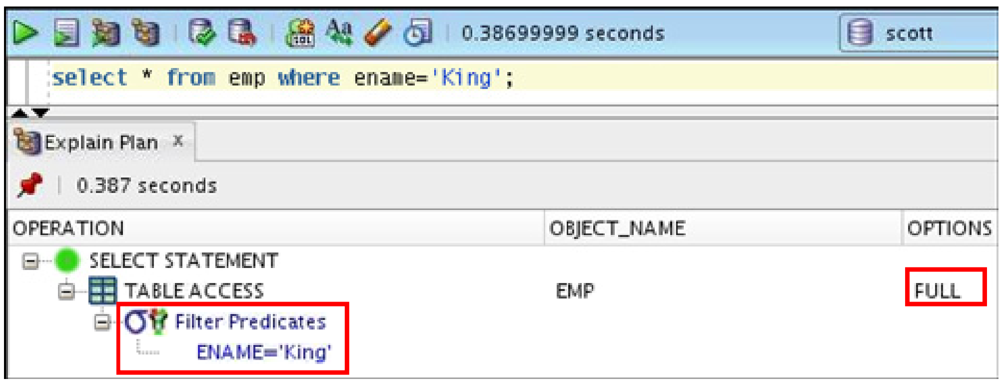
Lesson Agenda

- Row Source Operations
- Main Structures and Access Paths
- Table Access Paths
- Indexes: Overview
  - Normal B*-tree Indexes
  - Index Scans
  - Index-Organized Tables
  - Bitmap Indexes
  - Bitmap Operations
  - Composite Indexes
  - Invisible Index: Overview
- Common Observations

Full Table Scan

- Performs multiblock reads
  (here `DB_FILE_MULTIBLOCK_READ_COUNT = 4`)
- Reads all formatted blocks below the high-water mark
- May filter rows
- Is faster than index range scans for large amount of data





A full table scan (FTS) sequentially reads all rows from a table and filters out those that do not meet the selection criteria. During an FTS, 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 by 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 an FTS 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 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 V6, an 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.

Using Full Table Scan

Common questions:

- Are all full table scans bad?
- At what percentage of data does the optimizer consider a full table scan as the most efficient method to retrieve data from a table (20%+, 30%+, or 50%+, and so on)?

When to use:

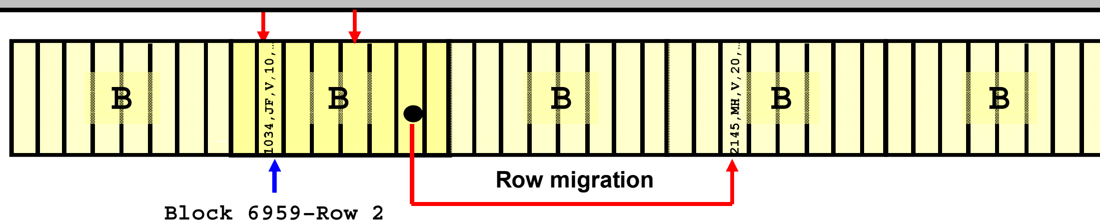
- No suitable index
- Low selectivity filters (or no filters)
- Small table
- High degree of parallelism
- Full table scan hint: `FULL (<table name>)`

The optimizer uses an FTS in any of the following cases:

- **Lack of index:** If the query is unable to use any existing indexes, it uses an FTS (unless a `ROWID` filter or a cluster access path is available). For example, if a function is used on the indexed column in the query, the optimizer cannot use the index and instead uses an FTS. If you need to use the index for case-independent searches, either do not permit mixed-case data in the search columns or create a function-based index, such as `UPPER(last_name)` on the search column.
- **Large amount of data (low selectivity):** If the optimizer thinks that the query accesses enough blocks in a table, it may use an FTS even though indexes might be available.
- **Small table:** If a table contains less than `DB_FILE_MULTIBLOCK_READ_COUNT` blocks under the high-water mark, an FTS might be cheaper than an index range scan, regardless of the fraction of tables being accessed or indexes present.
- **High degree of parallelism:** A high degree of parallelism for a table skews the optimizer towards FTS over range scans. Examine the `DEGREE` column in `ALL_TABLES` for the table to determine the degree of parallelism.
- **FTS hints:** Use the `FULL(table alias)` hint to instruct the optimizer to use an FTS.
- The table statistics are stale.

ROWID Scan

```
SELECT * FROM SCOTT.EMP WHERE rowid='AAAQ+LAAEAAAAAfAAJ';
--------------------------------------------------------------------
| Id | Operation                  | Name | Rows | Bytes | Cost |
--------------------------------------------------------------------
|  0 | SELECT STATEMENT           |      |    1 |    37 |    1|
|  1 |  TABLE ACCESS BY USER ROWID| EMP  |    1 |    37 |    1|
--------------------------------------------------------------------
```




```
SELECT * FROM employees WHERE employee_id > 190;
-----------------------------------------------------------------------
| Id | Operation                  | Name          | Rows | Bytes | Cost|
-----------------------------------------------------------------------
|  0 | SELECT STATEMENT           |               |   16 |  1104 |  2  |
|  1 |  TABLE ACCESS BY USER ROWID|EMPLOYEES      |   16 |  1104 |  2  |
| *2 |   INDEX RANGE SCAN         |EMP_EMP_ID_PK|   16 |       |  1  |
-----------------------------------------------------------------------
```


The ROWID of a row specifies the data file and data block that contains 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.

However, you rarely specify a ROWID directly in a query. Instead, the database gets ROWIDs from an index scan of a table's index (see the slides titled "Index Scans"). Table access might be required for columns in the statement that are not present in the index. 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. The second plan includes an index range scan of the emp_emp_id_pk index on the EMPLOYEES table. The database uses the ROWIDs obtained from the index to retrieve the corresponding rows from the EMPLOYEES table.

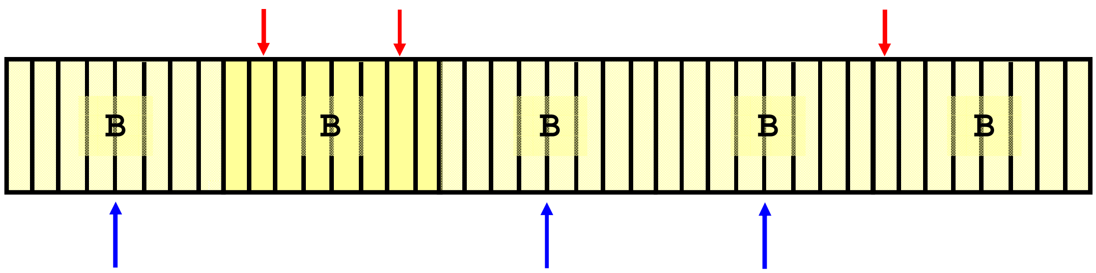
ROWIDs are the system's internal representation of where data is stored. Accessing data based on position is not recommended because rows can move around due to row migration and chaining, and also after export and import.

Note: Because of row migration, a ROWID can sometimes point to an address that is 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

```
SELECT * FROM emp SAMPLE BLOCK (10) SEED (1);

---------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost  (%CPU)|
---------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     1 |    38 |     2   (0)|
|   1 |  TABLE ACCESS SAMPLE | EMP  |     1 |    38 |     2   (0)|
---------------------------------------------------------------------
```



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 the table.

- **`SEED seed_value`:** Specify this clause to instruct the database to attempt to return the same sample from one execution to the next. `seed_value` must be an integer between 0 and 4294967295. If you omit this clause, the resulting sample changes from one execution to the next.

In row sampling, more blocks need to be accessed given a particular sample size, but the results are usually more accurate. Block samples are less costly, but may be inaccurate, more so with smaller samples.

Note: Block sampling is possible only during FTS or index fast full scans. If a more efficient execution path exists, Oracle Database does not perform block sampling. If you want to guarantee block sampling for a particular table or index, use the `FULL` or `INDEX_FFS` hint.

Quiz

A full table scan sequentially reads all rows from a table and filters out those that do not meet the selection criteria.

a. True
b. False

Answer: a

Lesson Agenda

- Row Source Operations
- Main Structures and Access Paths
- Table Access Paths
- Indexes: Overview
  - Normal B*-tree Indexes
  - Index Scans
  - Index-Organized Tables
  - Bitmap Indexes
  - Bitmap Operations
  - Composite Indexes
  - Invisible Index: Overview
- Common Observations

Indexes: Overview

- Storage techniques:
  - B*-tree indexes: The default and the most common
    - Normal
    - Function based: Pre-computed value of a function or an expression
    - Index-organized table (IOT)
  - Bitmap indexes
  - Cluster indexes: Defined specifically for a cluster
- Index attributes:
  - Key compression
  - Reverse key
  - Ascending, descending
- Domain indexes: Specific to an application or cartridge

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 to locate information fast, an Oracle Database index provides a faster access path to table data. 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 non-unique.

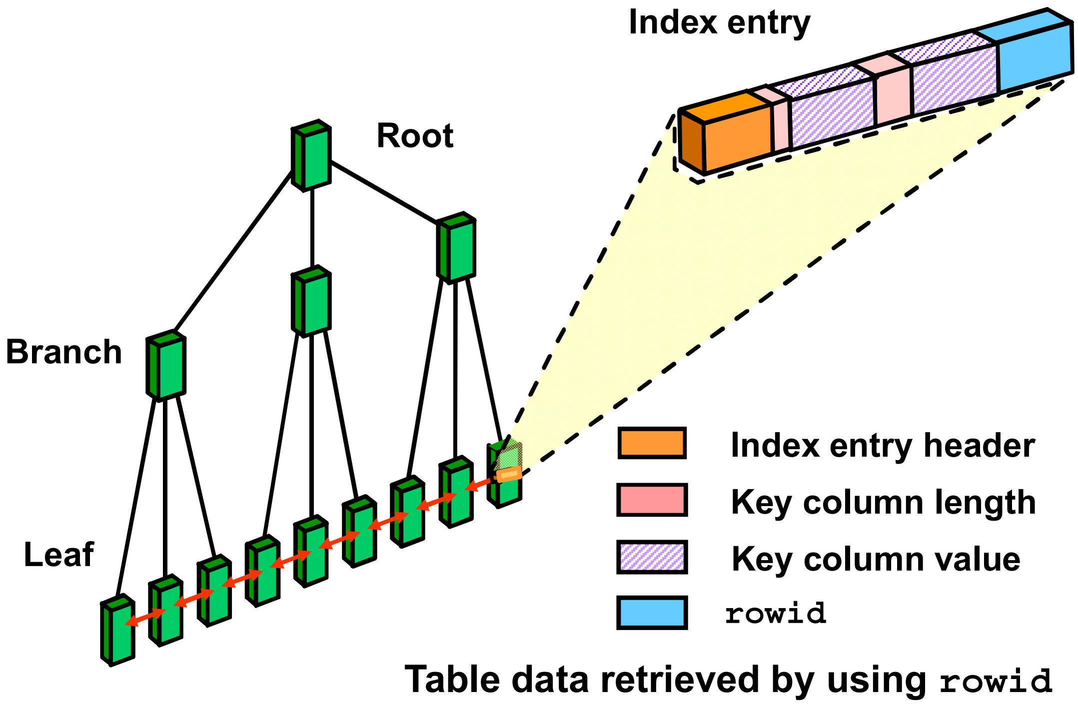
A composite index, also called a concatenated index, is an index that you create on multiple columns (up to 32) in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.

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 a few reads to find the correct row. However, the "B" in "B*-tree" does not stand for "binary," but rather for "balanced."

- **Descending indexes:** Descending indexes allow data to be sorted from “big to small” (descending) instead of from “small to big” (ascending) in the index structure.
- **Reverse key indexes:** These are B*-tree indexes whereby the bytes in the key are reversed. Reverse key indexes can be used to obtain a more even distribution of index entries throughout an index that is populated with increasing values. For example, if you use a sequence to generate a primary key, the sequence generates values such as 987500, 987501, and 987502. With a reverse key index, the database logically indexes 005789, 105789, 205789, and so on, instead of 987500, 987501, and 987502. Because these reverse keys are now likely to be placed in different locations, this placement can reduce contention for particular blocks that may otherwise be targets for contention. However, only equality predicates can benefit from these indexes.
- **Index key compression:** The basic concept behind a compressed key index is that every entry is broken into two—a prefix and a suffix component. The prefix is built on the leading columns of the concatenated index and has many repeating values. The suffix is built on the trailing columns in the index key and is the unique component of the index entry within the prefix. This compression is not the same as ZIP files compression; rather, this optional compression removes redundancies from concatenated (multicolumn) indexes.
- **Function-based indexes:** These B*-tree or bitmap indexes store the computed result of a function on a row’s column or columns, and not the column data itself. You can consider them as indexes on a virtual (derived or hidden) column. In other words, it is a column that is not physically stored in the table. You can gather statistics on this virtual column.
- **Index-organized tables:** These tables are stored in a B*-tree structure. Whereas rows of data in a heap-organized table are stored in an unorganized fashion (data goes wherever there is available space), data in an IOT is stored and sorted by a primary key. IOTs behave like regular tables as far as your application is concerned.
- **Bitmap indexes:** In a normal B*-tree, there is a one-to-one relationship between an index entry and a row; that is, an index entry points to a row. With bitmap indexes, a single index entry uses a bitmap to point to many rows simultaneously. They are appropriate for repetitive data (data with few distinct values relative to the total number of rows in the table) that is mostly read-only. Bitmap indexes should never be considered in an OLTP database for concurrency-related issues.
- **Bitmap join indexes:** A bitmap join index is a bitmap index for the join of two or more tables. A bitmap join index can be used to avoid actual joins of tables or to greatly reduce the volume of data that must be joined, by performing restrictions in advance. Queries using bitmap join indexes can be sped by using bitwise operations.
- **Application domain indexes:** These are indexes that you build with packages and store either in the database or even outside the database. You tell the optimizer how selective your index is and how costly it is to execute, and the optimizer decides whether or not to use your index based on that information.

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 `ROWID`s 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 top down. In certain situations, the balancing algorithm can cause the B*-tree height to increase unnecessarily. It is possible to reorganize indexes with the `ALTER INDEX … REBUILD | COALESCE` command.

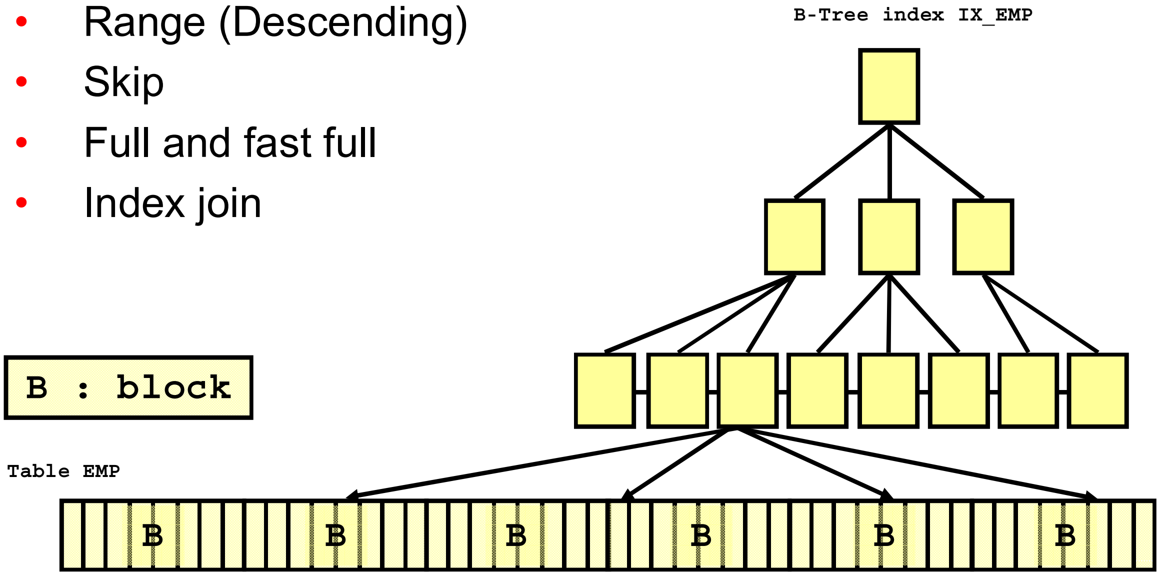
The internal structure of a B*-tree index allows rapid access to the indexed values. The system can directly access rows after it has retrieved the address (the `ROWID`) from the index leaf blocks.

Note: The maximum size of a single-index entry is approximately half of the data block size.

Index Scans

Types of index scans:

- Unique
- Range (Descending)
- Skip
- Full and fast full
- Index join



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 the columns of the index, the system reads the indexed column values directly from the index, rather than from the table.

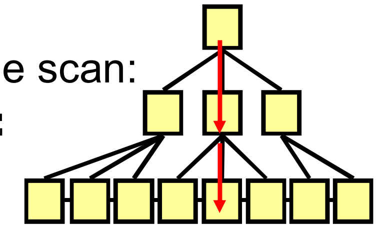
The index contains not only the indexed value but also the `ROWID`s of the rows in the table that have the value. Therefore, if the statement accesses other columns in addition to the indexed columns, the system can find the rows in the table by using either a table access by `ROWID` or a cluster scan.

Note: The graphic in the slide shows a case where four rows are retrieved from the table by using their `ROWID`s that are obtained by the index scan.

Index Unique Scan

The optimizer considers an index unique scan:

- If a SQL statement contains a `UNIQUE` or a `PRIMARY KEY` constraint
- When all the columns of a unique (B*-tree) index are specified with equality conditions



index UNIQUE Scan PK_EMP

```
select * from emp where empno = 9999;
```

```
----------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes  | Cost|
----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |     1 |   38   |   1  |
|   1 |  TABLE ACCESS BY INDEX ROWID |EMP    |     1 |   38   |   1  |
|  *2 |   INDEX UNIQUE SCAN          |PK_EMP |     1 |        |   0  |
----------------------------------------------------------------------
```


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.

In the example in the slide, the database uses the primary key on the `empno` column (`PK_EMP`). Note that the database is likely to perform an index unique scan when you specify all the columns of a unique index as well.

Key values and `ROWID`s are obtained from the index, and table rows are obtained by using `ROWID`s.

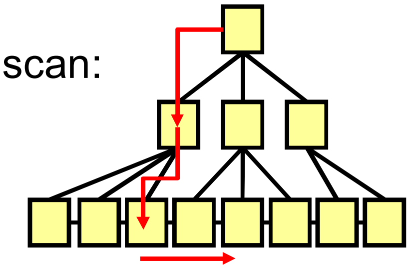
You can look for access conditions in the “Predicate Information” section of the execution plan. (The execution plan is discussed in detail in the lesson titled “Interpreting Execution Plans and Enhancements.”)

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

The optimizer considers an index range scan:
- When the optimizer finds one or more leading columns of an index specified in conditions and any combination of the preceding conditions
- When it can use unique or non-unique indexes
- If it can avoid sorting when index columns constitute the ORDER BY/GROUP BY clause and the indexed columns are NOT NULL because otherwise they are not considered




```
create index I_DEPTNO
on EMP(deptno);

select /*+ INDEX(EMP I_DEPTNO) */ *
from emp
where deptno = 10
and sal > 1000;
```


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.

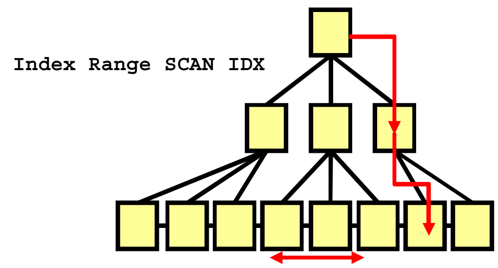
Wildcard searches (col1 like '%ASD') should not be in a leading position, because it does not result in a range scan.

Range scans can use unique or non-unique indexes. Range scans can avoid sorting when index columns constitute the ORDER BY/GROUP BY clause and the indexed columns are NOT NULL because otherwise they are not considered.

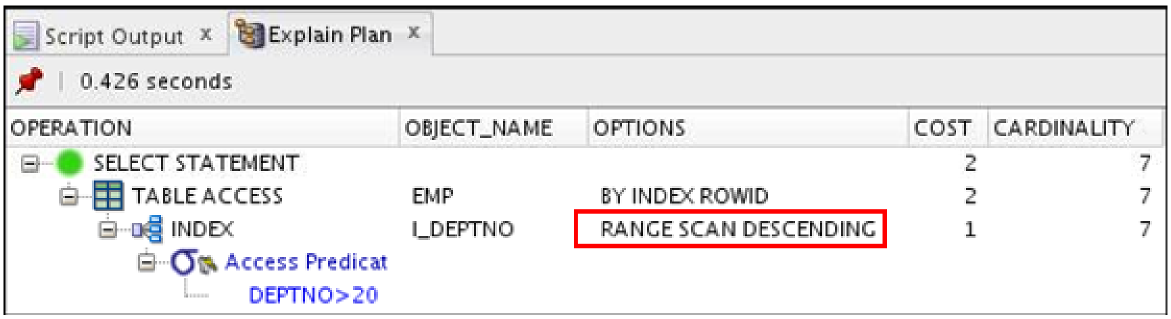
A descending index range scan is identical to an index range scan, except that data is returned in descending order. The optimizer uses a descending index range scan when an order by descending clause can be satisfied by an index.

In the example in the slide, using the I_DEPTNO index, 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




```
select * from emp where deptno>20 order by deptno desc;
```



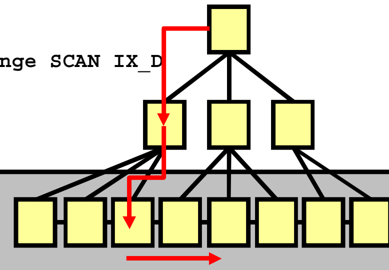
OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	7
TABLE ACCESS	EMP	BY INDEX ROWID	2	7
INDEX	I_DEPTNO	RANGE SCAN DESCENDING	1	7
Access Predicat				
DEPTNO>20				

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 `DEPTNO` column in the `EMP` table in descending order. 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 performed in ascending order.

Descending Index Range Scan




```
drop index I_Deptno;
create index IX_D on EMP(deptno desc);
select * from emp where deptno <30;
```

Script Output × | Explain Plan ×

0.011 seconds

OPERATION	OBJECT_NA...	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	9
TABLE ACCESS	EMP	BY INDEX ROWID	2	9
INDEX	IX_D	RANGE SCAN	1	1
Access Predicates				
SYS_OP_DESCEND(DEPTNO)>HEXTORAW('3EE0FF')				
Filter Predicates				
SYS_OP_UNDESCEND(SYS_OP_DESCEND(DEPTNO))<30				

A descending index range scan is identical to an index range scan, except that data is returned in descending order. Descending indexes allow 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 descending order to return the most recent data first, or when seeking a value that is less than a specified value, as shown in the example in the slide.

The optimizer uses a descending index range scan when an order by descending clause can be satisfied by a descending index.

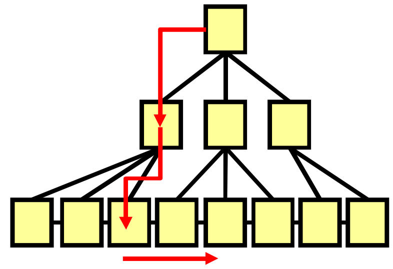
The `INDEX_DESC(table_alias index_name)` hint can be used to force this access path if possible.

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 by using the `SYS_OP_UNDESCEND` function.

Index Range Scan: Function-Based

The optimizer considers this scan:

- When frequently executed SQL statements include transformed columns, or columns in expressions, in a `WHERE` or an `ORDER BY` clause



Index Range SCAN IX_FBI

```
create index IX_FBI on EMP(UPPER(ename));

select * from emp where upper(ENAME) like 'A%';
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	1
TABLE ACCESS	EMP	BY INDEX ROWID BATCHED	2	1
INDEX	IX_FBI	RANGE SCAN	1	1
Access Predicates				
UPPER(ENAME) LIKE 'A%'				
Filter Predicates				
UPPER(ENAME) LIKE 'A%'				

ORACLE

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 an `ORDER BY` clause.

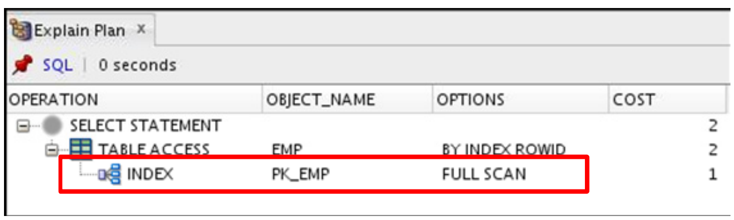
For example, function-based indexes that are defined with the `UPPER(column_name)` or `LOWER(column_name)` keywords allow non-case-sensitive searches, as shown in the slide.

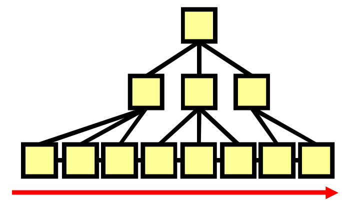
Index Full Scan

The optimizer considers an index full scan when:

- All the columns in the `ORDER BY` clause must be in the index
- The query requires a sort-merge join
- A `GROUP BY` clause is present in the query, and the columns in the `GROUP BY` clause are present in the index

```
select * from emp
order by empno;
```





index Full Scan I_DEPTNO

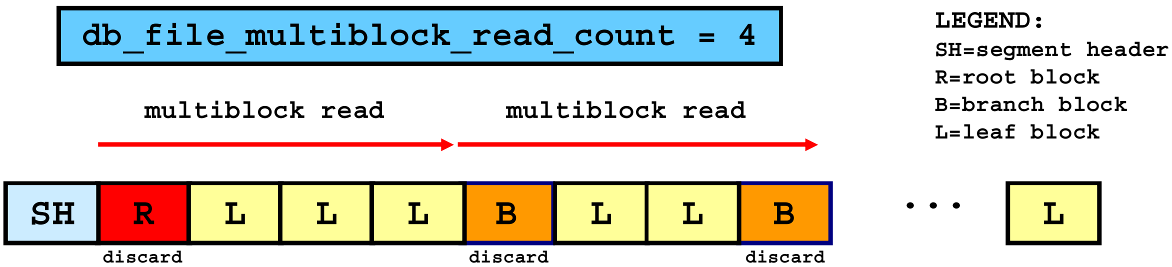
A full scan can be used to eliminate a sort operation because data is ordered by an index key. An index full scan reads the index by using single-block I/O (unlike a fast full index scan).

Oracle Database may use a full scan in any of the following situations:

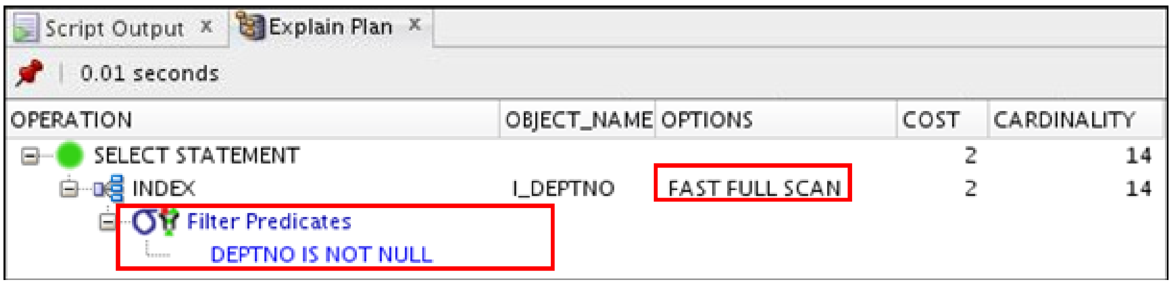
- An `ORDER BY` clause that meets the following requirements is present in the query:
  - All the columns in the `ORDER BY` clause must be in the index.
  - The order of the columns in the `ORDER BY` clause must match the order of the leading index columns.
- The `ORDER BY` clause can contain all the columns in the index or a subset of the columns in the index.
- The query requires a sort-merge join. The database can perform a full index scan instead of doing a full table scan, followed by a sort when the query meets the following requirements:
  - All the columns referenced in the query must be in the index.
  - The order of the columns referenced in the query must match the order of the leading index columns.
  - The query can contain all the columns in the index or a subset of the columns in the index.

- A `GROUP BY` clause is present in the query, and the columns in the `GROUP BY` clause are present in the index. The columns do not need to be in the same order in the index and the `GROUP BY` clause. The `GROUP BY` clause can contain all the columns in the index or a subset of the columns in the index.

Index Fast Full Scan




```
select /*+ INDEX_FFS(EMP I_DEPTNO)  */ deptno from emp
where deptno is not null;
```



Index fast full scans are an alternative to full table scans when the index contains all the columns that are needed for the query. A fast full scan accesses the data in the index itself without accessing the table.

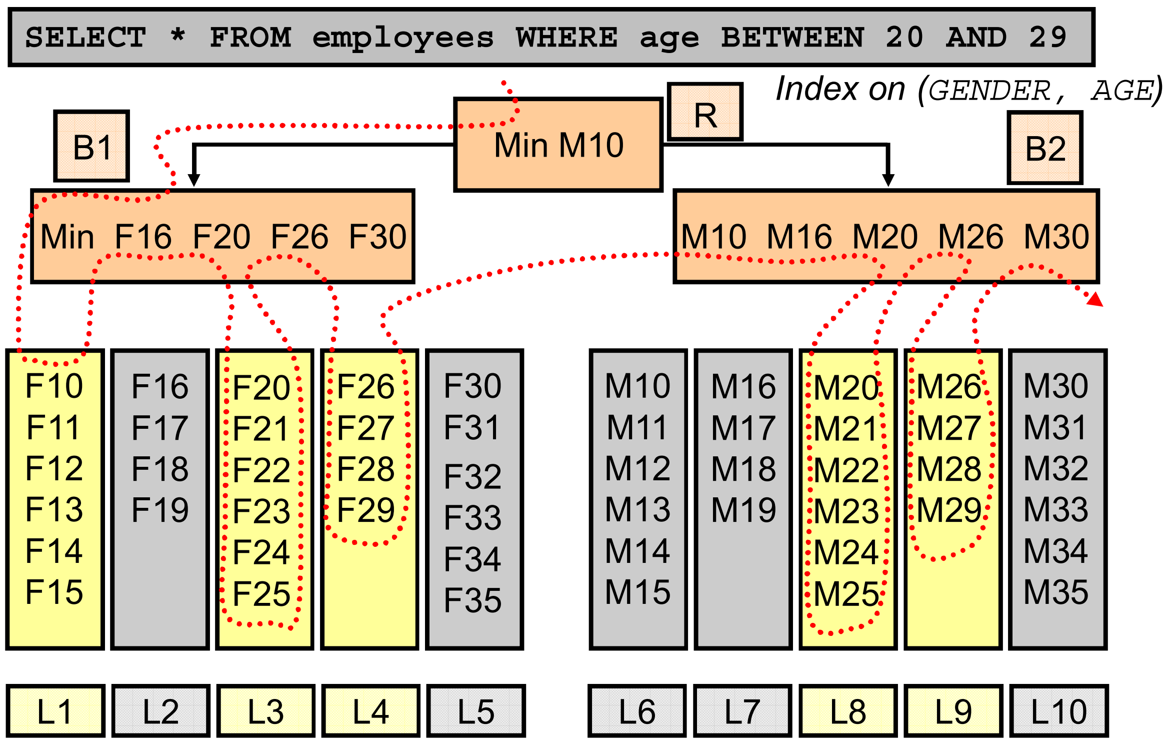
It cannot be used to eliminate a sort operation because data is not ordered by an index key. It can be used for the `min`/`avg`/`sum` aggregate functions. In this case, the optimizer must know that all table rows are represented in the index.

This operation reads the entire index by using multiblock reads (unlike a full index scan). 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 index fast full scans with the `OPTIMIZER_FEATURES_ENABLE` initialization parameter or the `INDEX_FFS` hint, as shown in the example in the slide.

Note: Index fast full scans are used against an index when it is rebuilt offline.

Index Skip Scan



Index skip scans improve index scans by skipping blocks that could never contain keys that match 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 the first entry to be F16, goes to the left leaf [L1], and starts to scan it because it could contain A25 (that is, where “gender” is before “F” in the alphabet). The server identifies that this is not possible because the first entry is F10. It is therefore not possible to find an entry such as A25 in this leaf, and 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.

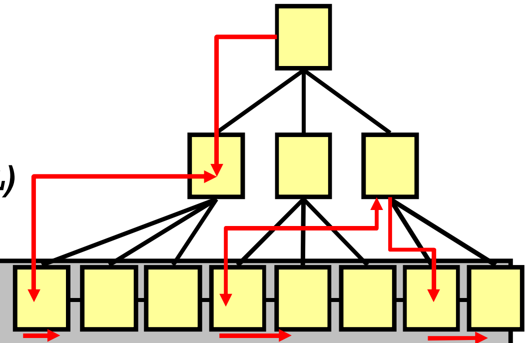
Returning to [B1], the server finds that the next two entries have a common prefix of F2. This identifies possible subtrees to scan. The system knows that these subtrees are ordered by age.

The third and fourth leaf blocks [L3–L4] are scanned, and some values are retrieved. By looking at the fourth entry in the first branch block [B1], the system determines that it is no longer possible to find an F2*x* entry. Thus, it is not necessary to scan that subtree [L5].

The same process continues with the right part of this index. Note that out of a total of 10 leaf blocks, only five are scanned.

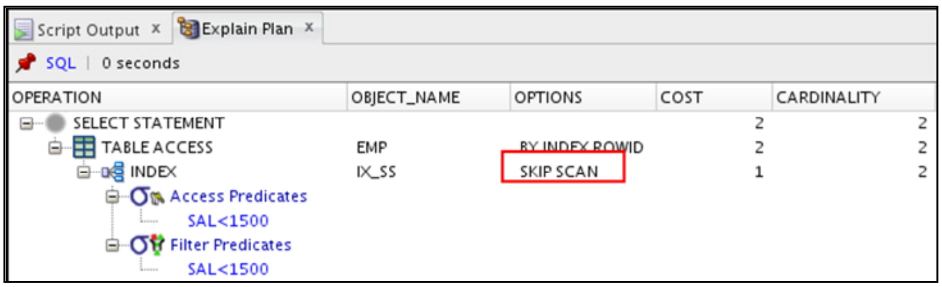
Index Skip Scan: Example

Index on (`DEPTNO, SAL`)



```
create index IX_SS on EMP(DEPTNO,SAL);

select /*+ index_ss(EMP IX_SS) */ * from emp where SAL < 1500;
```



Script Output | Explain Plan

SQL | 0 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	2
TABLE ACCESS	EMP	BY INDEX ROWID	2	2
INDEX	IX_SS	SKIP SCAN	1	2
Access Predicates				
SAL<1500				
Filter Predicates				
SAL<1500				

The example in the slide finds employees who have salary less than 1500, by 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 has only some discrete values; that is, 10, 20, and 30.

Skip scanning lets a composite index be split logically into smaller sub-indexes. The number of logical sub-indexes is determined by the number of distinct values in the initial column.

The system pretends that the index is three little index structures hidden inside a 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 a few distinct values in the leading column of the composite index and many distinct values in the non-leading key of the index.

Index Join Scan

```
alter table emp modify (SAL not null, ENAME not null);
create index I_ENAME on EMP(ename);
create index I_SAL on EMP(sal);
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	14
VIEW	index$_join$_001		2	14
HASH JOIN				
Access Predicates				
ROWID=ROWID				
INDEX	I_ENAME	FAST FULL SCAN	1	14
INDEX	I_SAL	FAST FULL SCAN	1	14

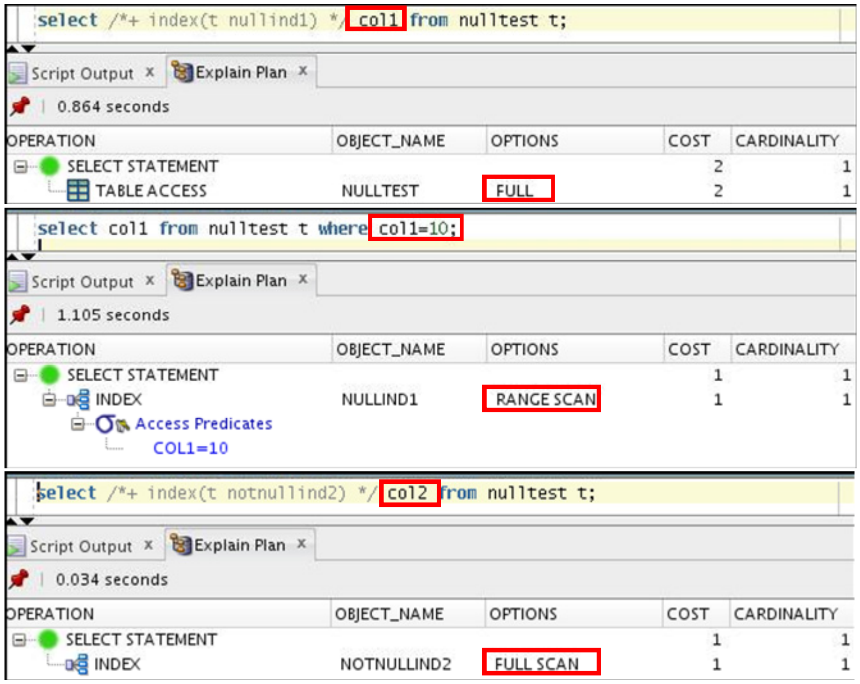
An index join is a hash join of several indexes that together contain all the table columns that are referenced in the query. If an index join is used, no table access is needed because all relevant column values can be retrieved from the indexes. An index join cannot be used to eliminate a sort operation.

The index join is not a real join operation (note that the example in the slide is a single table query), but it is built by using index accesses, followed by a join operation on ROWIDs. 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

```
create table nulltest ( col1 number, col2 number not null);
create index nullind1 on nulltest (col1);
create index notnullind2 on nulltest (col2);
```



ORACLE

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 search only 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 with the `COL1 IS NOT NULL` predicate.

Using Indexes: Considering Nullable Columns

Column	Null?
SSN	Y
FNAME	Y
LNAME	N
⋮	

PERSON

```
CREATE UNIQUE INDEX person_ssn_ix
    ON person(ssn);
```

```
SELECT COUNT(*) FROM person;

  SELECT STATEMENT     |
   SORT AGGREGATE      |
    TABLE ACCESS FULL| PERSON
```

```
DROP INDEX person_ssn_ix;
```

Column	Null?
SSN	N
FNAME	Y
LNAME	N
⋮	

PERSON

```
ALTER TABLE person ADD CONSTRAINT pk_ssn
    PRIMARY KEY (ssn);
```

```
SELECT /*+ INDEX(person) */ COUNT(*) FROM
    person;

  SELECT STATEMENT        |
   SORT AGGREGATE         |
    INDEX FAST FULL SCAN| PK_SSN
```

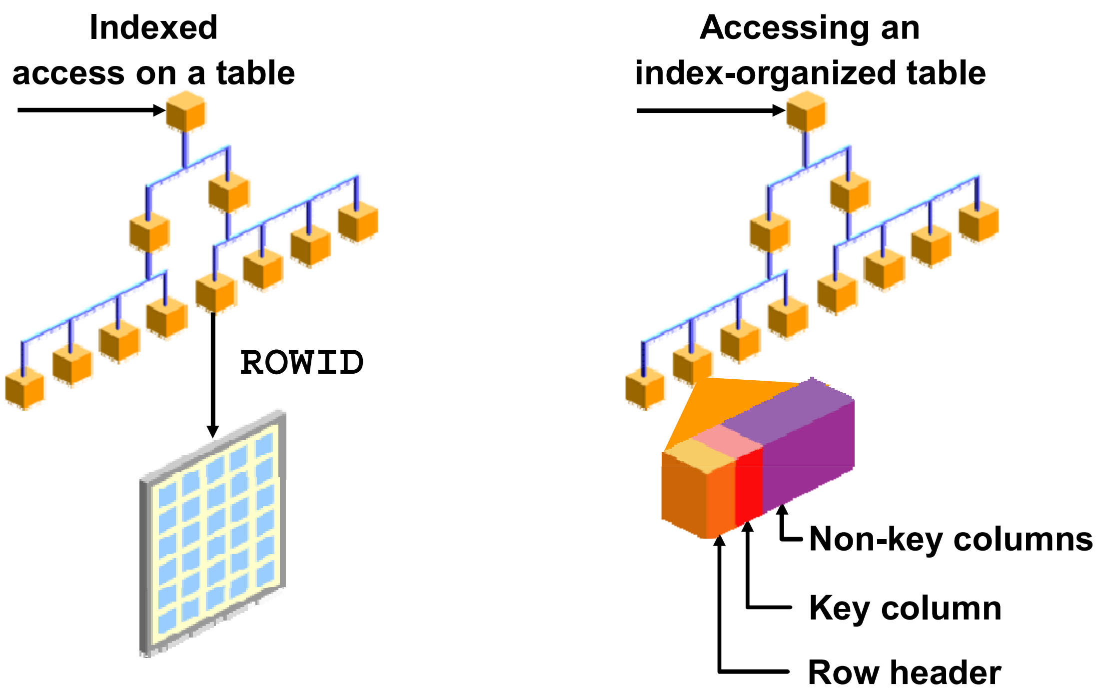
ORACLE

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 than 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 instead 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 IOT is 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 `ROWID`s. 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.

IOTs 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

```
create table iotemp
 ( empno number(4) primary key, ename varchar2(10) not null,
   job varchar2(9), mgr number(4), hiredate date,
   sal number(7,2) not null,  comm number(7,2), deptno number(2))
organization index;
```

```
select * from iotemp where empno=9999;
```

Script Output × | Explain Plan ×

0.734 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			1	1
INDEX	SYS_IOT_TOP_7...	UNIQUE SCAN	1	1
Access Predicates				
EMPNO=9999				

```
select * from iotemp where sal>1000;
```

Script Output × | Explain Plan ×

0.007 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	1
INDEX	SYS_IOT_TOP_7...	FAST FULL SCAN	2	1
Filter Predicates				
SAL>1000				

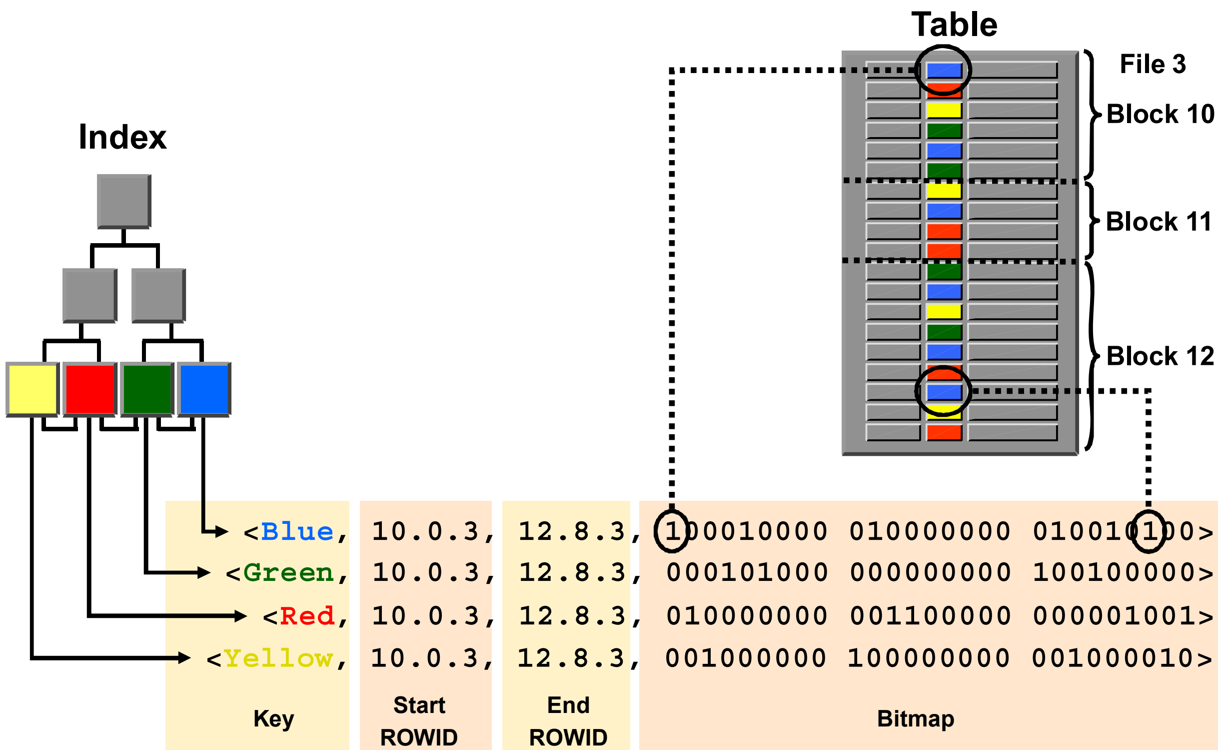
ORACLE

IOTs 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 that is 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 `ROWID`s. Based on these values, the system uses an internal algorithm to map bitmaps onto `ROWID`s. 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 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.

Note: Unlike most other types of indexes, bitmap indexes include rows that have `NULL` values. Indexing of nulls can be useful for some types of SQL statements, such as queries with the aggregate function `COUNT`. The `IS NOT NULL` predicate can also benefit from bitmap indexes. Although bitmaps are compressed internally, they are split in multiple leaves if the number of rows increases.

Bitmap Index Access: Examples

```
SELECT * FROM PERF_TEAM WHERE country='FR';
---------------------------------------------------------------------
| Id | Operation                     | Name      | Rows | Bytes |
---------------------------------------------------------------------
|  0 | SELECT STATEMENT              |           |    1 |    45 |
|  1 |  TABLE ACCESS BY INDEX ROWID  | PERF_TEAM |    1 |    45 |
|  2 |   BITMAP CONVERSION TO ROWIDS |           |      |       |
|  3 |    BITMAP INDEX SINGLE VALUE  | IX_B2     |      |       |
---------------------------------------------------------------------
Predicate:  3 - access("COUNTRY"='FR')
```

```
SELECT * FROM PERF_TEAM WHERE country>'FR';
---------------------------------------------------------------------
| Id | Operation                     | Name      | Rows | Bytes |
---------------------------------------------------------------------
|  0 | SELECT STATEMENT              |           |    1 |    45 |
|  1 |  TABLE ACCESS BY INDEX ROWID  | PERF_TEAM |    1 |    45 |
|  2 |   BITMAP CONVERSION TO ROWIDS |           |      |       |
|  3 |    BITMAP INDEX RANGE SCAN    | IX_B2     |      |       |
---------------------------------------------------------------------
Predicate: 3 - access("COUNTRY">'FR') filter("COUNTRY">'FR')
```

ORACLE

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 `ROWID`s and have their corresponding rows returned for the query.

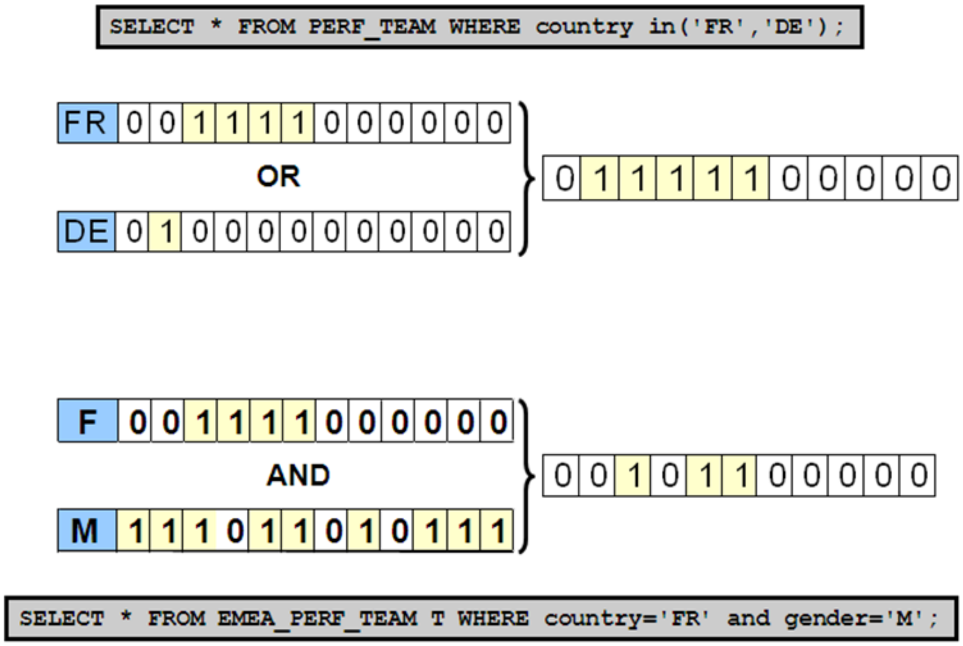
In some cases (such as a query counting the number of rows with `COUNTRY` FR), the query might simply use the bitmap itself and count the number of 1s (not needing the actual rows).

This is illustrated in the following example:

```
SELECT count(*) FROM PERF_TEAM WHERE country>'FR';

---------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)|
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |     1 |     3 |     2   (0)|
|   1 |  SORT AGGREGATE              |       |     1 |     3 |            |
|   2 |   BITMAP CONVERSION COUNT    |       |     1 |     3 |     2   (0)|
|   3 |    BITMAP INDEX RANGE SCAN| IX_B2 |       |       |            |
---------------------------------------------------------------------------
Predicate: 3 - access("COUNTRY">'FR') filter("COUNTRY">'FR')
```

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 filtering improves response time, often dramatically. Because the bitmaps from bitmap indexes can be combined quickly, it is usually best to use single-column bitmap indexes.

Because of fast bit-and, bit-minus, and bit-or operations, bitmap indexes are efficient when:

- You use `IN (value_list)`
- Predicates are combined with `AND` or `OR`

Combining Bitmap Index Access Paths

```
SELECT * FROM PERF_TEAM WHERE country in ('FR','DE');
---------------------------------------------------------------------
| Id | Operation                        | Name      | Rows | Bytes |
|  0 | SELECT STATEMENT                 |           |    1 |    45 |
|  1 |  INLIST ITERATOR                 |           |      |       |
|  2 |   TABLE ACCESS BY INDEX ROWID    | PERF_TEAM |    1 |    45 |
|  3 |    BITMAP CONVERSION TO ROWIDS   |           |      |       |
|  4 |     BITMAP INDEX SINGLE VALUE    | IX_B2     |      |       |
Predicate: 4 - access("COUNTRY"='DE' OR "COUNTRY"='FR')
```

```
SELECT * FROM PERF_TEAM WHERE country='FR' and gender='M';
---------------------------------------------------------------------
| Id | Operation                       | Name      | Rows | Bytes |
|  0 | SELECT STATEMENT                |           |    1 |    45 |
|  1 |  TABLE ACCESS BY INDEX ROWID    | PERF_TEAM |    1 |    45 |
|  2 |   BITMAP CONVERSION TO ROWIDS   |           |      |       |
|  3 |    BITMAP AND                   |           |      |       |
|  4 |     BITMAP INDEX SINGLE VALUE   | IX_B1     |      |       |
|  5 |     BITMAP INDEX SINGLE VALUE   | IX_B2     |      |       |
Predicate: 4 - access("GENDER"='M') 5 - access("COUNTRY"='FR')
```


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 are looking for. The more complex the compound `WHERE` clauses become, the more benefit you get from bitmap indexing.

Bitmap Operations

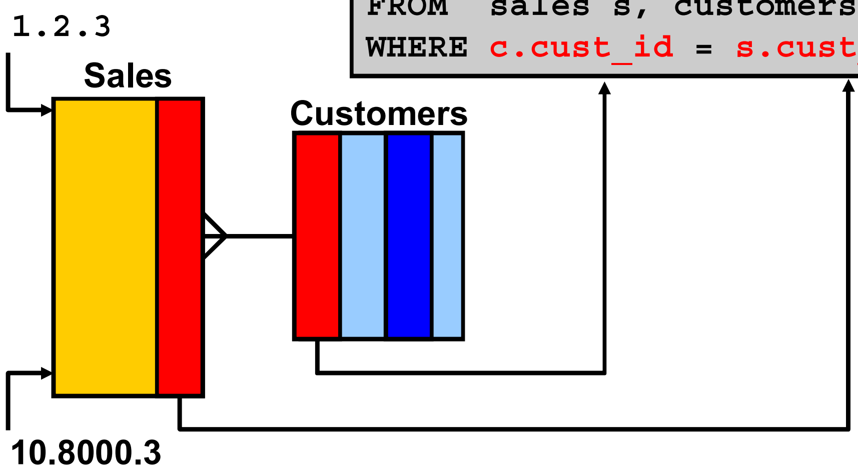
- `BITMAP CONVERSION`:
  - `TO ROWIDs`
  - `FROM ROWIDs`
  - `COUNT`
- `BITMAP INDEX`:
  - `SINGLE VALUE`
  - `RANGE SCAN`
  - `FULL SCAN`
- `BITMAP MERGE`
- `BITMAP AND/OR`
- `BITMAP MINUS`
- `BITMAP KEY ITERATION`

The slide summarizes all possible bitmap operations. The following operations have not been explained so far:

- `BITMAP CONVERSION FROM ROWID`: This refers to a B*-tree index that is converted by the optimizer into `BITMAP` (cost is lower than other methods) to make these efficient bitmap 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 that result 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 by 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

```
CREATE BITMAP INDEX cust_sales_bji
ON     sales(c.cust_city)
FROM   sales s, customers c
WHERE  c.cust_id = s.cust_id;
```




```
       <Rognes, 1.2.3, 10.8000.3, 1000100100101100…>
<Aix-en-Provence, 1.2.3, 10.8000.3, 0001010000100000…>
    <Marseille, 1.2.3, 10.8000.3, 0100000001000001…>
```

ORACLE

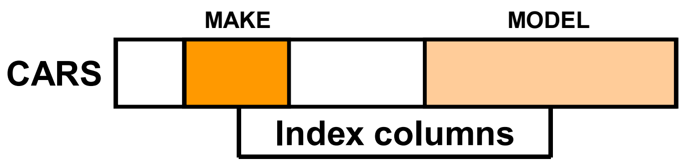
In addition to a bitmap index on a single table, you can create a bitmap join index. A bitmap join index is a bitmap index for the join of two or more tables. A bitmap join index is a space-efficient way of reducing the volume of data that must be joined by performing the join in advance.

Note: Bitmap join indexes are much more efficient in storage than materialized join views.

In the example in the slide, 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 it is also a foreign key inside `SALES`, although it is not required.

The `FROM` and `WHERE` clauses 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 with one particular key. Each bit in a bitmap corresponds to one row in the `SALES` table. In the first key in the slide (Rognes), the first row in the `SALES` table corresponds to a product sold to a Rognes customer, whereas 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




```
create index cars_make_model_idx on cars(make, model);
```

```
select *
from cars
where make = 'CITROËN' and model = '2CV';
```

```
-----------------------------------------------------------------
| Id  | Operation                      | Name                     |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT               |                          |
|   1 |  TABLE ACCESS BY INDEX ROWID| CUSTOMERS                |
|*  2 |   INDEX RANGE SCAN             | CARS_MAKE_MODEL_IDX      |
-----------------------------------------------------------------
```


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.

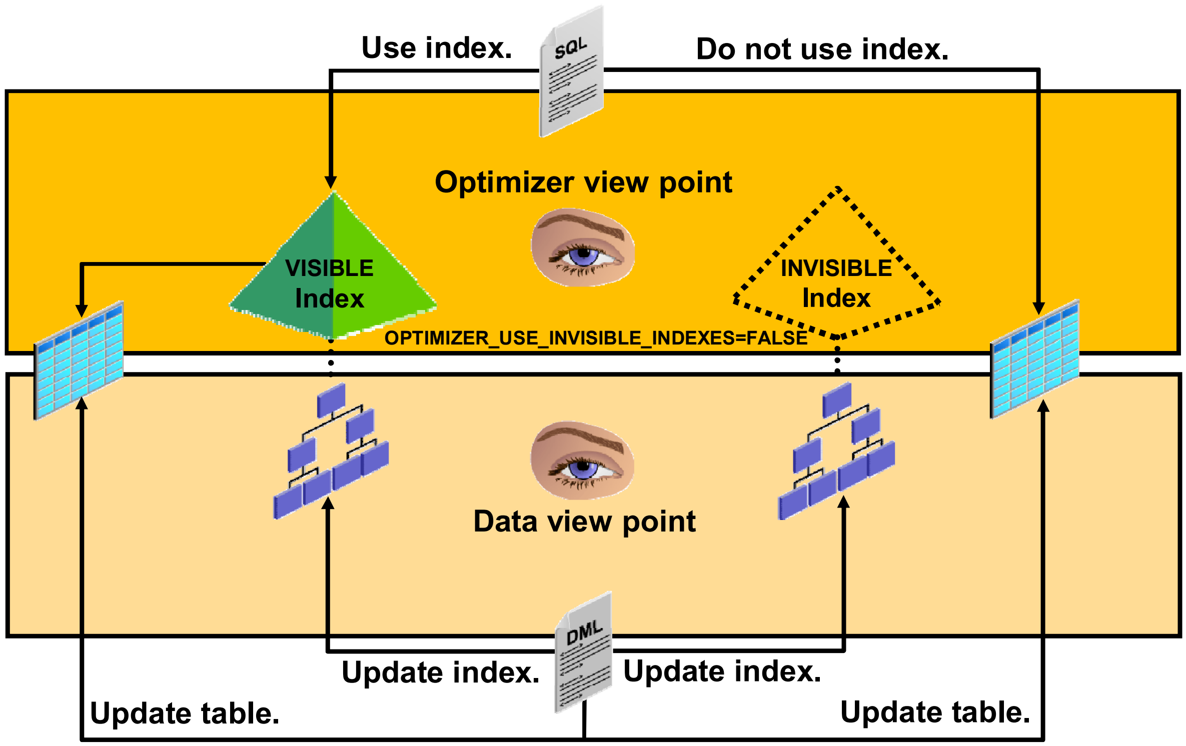
Composite indexes can provide additional advantages over single-column indexes:

- **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 because 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. With 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.

Invisible Indexes: Examples

- An index is altered as not visible to the optimizer:

```
ALTER INDEX ind1 INVISIBLE;
```

- The optimizer does not consider this index:

```
SELECT /*+ index(TAB1 IND1) */ COL1 FROM TAB1 WHERE …;
```

- The optimizer considers this index:

```
ALTER INDEX ind1 VISIBLE;
```

- You initially create an index as invisible:

```
CREATE INDEX IND1 ON TAB1(COL1) INVISIBLE;
```


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 initially create an index 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.
- Index the correct tables and columns.
- Order index columns for performance.
- Limit the number of indexes for each table.
- Drop indexes that are no longer required.
- Specify the tablespace for each index.
- Consider parallelizing index creation.
- Consider creating indexes with `NOLOGGING`.
- Consider the costs and benefits of coalescing or rebuilding indexes.
- Consider cost before disabling or dropping constraints.

- Create indexes after inserting table data: Data is often inserted or loaded into a table by using either the SQL*Loader or an import utility. It is more efficient to create an index for a table after inserting or loading 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 percent of the rows in a large table.
  - To improve performance on joins of multiple tables, index the columns used for the 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.

- Columns not suitable for indexing:
  - There are many nulls in the column, and you do not search on the not null values.
  - The `LONG` and `LONG RAW` columns cannot be indexed.
  - You can create unique or non-unique indexes on virtual columns.
- Order index columns for performance: The order of columns in the `CREATE INDEX` statement can affect query performance. In general, specify the most frequently used columns first.
- Limit the number of indexes for each table: A table can have any number of indexes. However, the more indexes there are, the more overhead is incurred as the table is modified. Thus, there is a trade-off between the speed of retrieving data from a table and the speed of updating the table.
- Drop indexes that are no longer required.
- Specify a tablespace for each index: If you use the same tablespace for a table and its index, it can be more convenient to perform database maintenance, such as tablespace backup.
- Consider parallelizing index creation: To speed up index creation, you can parallelize index creation, just as you can parallelize table creation. However, an index created with an `INITIAL` value of 5M and a parallel degree of 12 consumes at least 60 MB of storage during index creation.
- Consider creating indexes with `NOLOGGING`: You can create an index and generate minimal redo log records by specifying `NOLOGGING` in the `CREATE INDEX` statement. Because indexes created using `NOLOGGING` are not archived, perform a backup after you create the index. Note that `NOLOGGING` is the default in a `NOARCHIVELOG` database.
- Consider the costs and benefits of coalescing or rebuilding indexes: Improper sizing or increased growth can produce index fragmentation. To eliminate or reduce fragmentation, you can rebuild or coalesce the index. But before you perform either task, weigh the costs and benefits of each option, and select the one that works best for your situation.
- Consider cost before disabling or dropping constraints: Because unique and primary keys have associated indexes, you should factor in the cost of dropping and creating indexes when considering whether to disable or drop a `UNIQUE` or `PRIMARY KEY` constraint. If the associated index for a `UNIQUE` key or `PRIMARY KEY` constraint is extremely large, you can save time by leaving the constraint enabled rather than dropping and re-creating the large index. You also have the option of explicitly specifying that you want to keep or drop the index when dropping or disabling a `UNIQUE` or `PRIMARY KEY` constraint.

Quiz

Assuming that the column `EMAIL` has an index, the following query results in an index range scan:

```
Select employee_name from employees where
email like '%A';
```

a. True

b. False

Answer: b

Quiz

To get optimum results from indexes:

a. Create indexes before inserting table data
b. Do not order index columns
c. Limit the number of indexes for each table
d. Do not specify the tablespace for each index

Answer: c

Lesson Agenda

- Row Source Operations
- Main Structures and Access Paths
- Table Access Paths
- Indexes: Overview
  – Normal B*-tree Indexes
  – Index Scans
  – Index-Organized Tables
  – Bitmap Indexes
  – Bitmap Operations
  – Composite Indexes
  – Invisible Index: Overview
- Common Observations

Common Observations

Review the following common observations:

- Why is a full table scan used?
- Why is a full table scan not used?
- Why is an index scan not used?

In the following slides, you learn about common observations for table scans and index scans.

For detailed information, see MOS note 390374.1, "Oracle Performance Diagnostic Guide" (Query Tuning > References).

Why Is a Full Table Scan Used?

Review the following common causes:

- You set parameters that affect the optimizer's cost estimation.
- A large volume of business data must be processed.

The optimizer uses an undesired full table scan over an index scan in any of the following cases:

- **Parameter settings:** A few parameters may influence the optimizer to choose a full table scan. See the following parameter settings:

  `optimizer_index_cost_adj =` <much higher than 100>

  `db_file_multiblock_read_count =` <greater than 1MB/db_block_size>

  `optimizer_mode=`all_rows

  You could improve the query by changing certain non-default parameter settings.
- **Large volume of data (low selectivity):** If the query must indeed process a large volume of data, it may use a full table scan even though indexes might be available. You could tune the query by using parallel executions or materialized views.

Why Is Full Table Scan Not Used?

Review the following common causes:

- `INDEX FULL SCAN` is used to avoid a sort operation.
- You set parameters that affect the optimizer's cost estimation.
- Optimizer mode or hint is set to `FIRST_ROWS` or `FIRST_ROWS_n`.
- Query has a `USE_NL` hint that is not appropriate.
- Query has a `USE_NL`, `FIRST_ROWS`, or `FIRST_ROWS_n` hint that is favoring a nested loop join.
- No parallel slaves are available for the query.

`INDEX FULL SCAN` is used to avoid a sort operation.

`INDEX FULL SCAN` can be used to avoid a sort operation such as `ORDER BY`. Sometimes, `INDEX FULL SCAN` is better than a sort operation, but not always. If the optimizer makes incorrect estimations, it leads to a bad use of the `INDEX FULL SCAN` operation. You could improve the query by using the `NO_INDEX` or `ALL_ROWS` hint or by tuning PGA memory to adjust a sorting cost.

You set parameters that affect the optimizer's cost estimation.

A few parameters may influence the optimizer to choose index scans and nested loop joins. See the following parameter settings:

`optimizer_index_cost_adj` = <much lower than 100>
`db_file_multiblock_read_count` = <smaller than 1MB/db_block_size>
`optimizer_index_caching` = <too high>
`optimizer_mode=first_rows` (or first_rows_N)

You could improve the query by changing certain non-default parameter settings.

The optimizer mode or hint is set to FIRST_ROWS or FIRST_ROWS_n.

Incorrect OPTIMIZER_MODE affects how the optimizer approaches the execution plan and how it estimates the costs of access methods and join types. You could set it to ALL_ROWS or the ALL_ROWS hint when a large number of rows are expected.

The query has a USE_NL hint that is not appropriate.

The query possibly performs better with other join operations, such as a hash join or a merge join. You could examine the query by removing the hints that are influencing the choice of index.

The query has a USE_NL, FIRST_ROWS, or FIRST_ROWS_n hint that is favoring NL.

Same as "query has a USE_NL hint that is not appropriate." Note that nested loop (NL) joins will not be cost competitive when indexes are not available to the optimizer.

No parallel slaves are available for the query.

If no parallel slaves are available, the query runs in serial. It may influence the optimizer to choose an index scan.

Why Is an Index Scan Not Used?

Review the following common causes:

- There are functions being applied to the predicate.
- There is a data type mismatch.
- Statistics are old.
- The column can contain null.
- Using the index would actually be slower than not using it.
- You set parameters that affect the optimizer's cost estimation.
- The optimizer costs a full table scan cheaper than a series of index range scans.
- No index is available for columns in the predicate.

The optimizer mode or hint is set to `FIRST_ROWS` or `FIRST_ROWS_n`.

Incorrect `OPTIMIZER_MODE` affects how the optimizer approaches the execution plan and how it estimates the costs of access methods and join types. You could set it to `ALL_ROWS` or the `ALL_ROWS` hint when a large number of rows are expected.

The query has a `USE_NL` hint that is not appropriate.

The query possibly performs **better** with other join operations, such as a hash join or a merge join. You could examine the query by removing the hints that are influencing the choice of index.

The query has a `USE_NL`, `FIRST_ROWS`, or `FIRST_ROWS_n` hint that is favoring NL.

Same as "query has a `USE_NL` hint that is not appropriate." Note that nested loop joins will not be cost competitive when indexes are not available to the optimizer.

No parallel slaves are available for the query.

If no parallel slaves are available, the query runs in serial. It may influence the optimizer to choose an index scan.

Old Statistics

The statistics are considered by the optimizer when deciding whether to use an index. If they are outdated, they may influence the optimizer to make poor decisions about indexes.

Null Columns

If a column can contain nulls, it may prevent the use of an index on that column. This topic is covered later in this lesson.

Slower Index

Sometimes the use of an index is not efficient. This topic is covered later in this lesson.

Setting parameters

A few parameters may influence the optimizer to choose a full table scan.

See the following parameter settings:

`optimizer_index_cost_adj` = <much higher than 100>

`db_file_multiblock_read_count` = <greater than 1MB/db_block_size>
`optimizer_mode`=all_rows

You could improve the query by changing certain non-default parameter settings.

A full table scan is cheaper than a series of index range scans.

The optimizer determines that it is cheaper to do a full table scan rather than expand the `IN` list or `OR` into separate query blocks where each one uses an index.

You could improve the query by using the `USE_CONCAT` hint. The `USE_CONCAT` hint instructs the optimizer to transform combined `OR` conditions in the `WHERE` clause of a query into a compound query using the `UNION ALL` set operator. Without this hint, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them.

The `USE_CONCAT` hint overrides the cost consideration. For example:

```
SELECT /*+ USE_CONCAT */ *
FROM employees e
WHERE manager_id = 108
         OR department_id = 110;
```

No index available

No indexes are available for one or more columns in the query predicate. Oracle has only a full table scan access available.

You could improve the query by creating a new index or by re-creating the existing index. In order to get recommendations on indexes, the SQL Access Advisor can be used. It recommends bitmap, function-based, and B*-tree indexes. Note that the SQL Access Advisor is a separate option.

Why Is an Index Scan Not Used?

Review the following common causes:

- The available indexes are too unselective.
- The index's cluster factor is too high.
- The query has a hint that is preventing the use of indexes.
- The index hint is being ignored.
- The incorrect `OPTIMIZER_MODE` is being used.
- A filter predicate is missing.

The available indexes are too unselective.

If the available indexes are too unselective, you may need to re-create the existing indexes. For better selectivity, consider a composite index rather than multiple single-column indexes. When creating the composite key index, the index key order is important. The most selective column should be a part of the leading portion. For columns that have few distinct values and are not updated frequently, consider bitmap indexes.

The index's cluster factor is too high.

The index clustering factor measures row order in relation to an indexed value. The more order that exists in row storage for this value, the lower the clustering factor. The clustering factor is useful as a rough measure of the number of required I/Os. If the cluster factor is too high, total index access cost is expensive.

Total index access cost = index cost + table cost

Index cost = # of Levels + (index selectivity * index leaf blocks)

Table cost = table selectivity * cluster factor

The cluster factor is covered in the lesson titled "Other Optimizer Operators."

The query has a hint that is preventing the use of indexes.

The query contains hints that prevent the use of indexes, such as `INDEX_*`, `NO_INDEX`, `NO_INDEX_*`, and `FULL`. You could remove hints to influence the choice of index.

The index hint is being ignored.

Index hints may be ignored because the hints have syntax errors, table aliases were not used, or selected join orders or types may make it semantically impossible to use the index. You must correct common problems with hints.

The incorrect `OPTIMIZER_MODE` is being used.

An incorrect `OPTIMIZER_MODE` affects how the optimizer approaches the execution plan and how it estimates the costs of access methods and join types. You could set it to `FIRST_ROWS` or `FIRST_ROWS_n` when a small number of rows are expected. This often produces better plans for OLTP applications.

A filter predicate is missing.

Examine the predicate to see if any tables are missing a filter condition. Discuss or observe how the data from the query is used by end users.

Summary

In this lesson, you should have learned to:

- Describe the SQL operators for tables and indexes
- List the possible access paths
- Describe common observations

Practice 8: Overview

This practice covers using different access paths for better optimization (case 1 through case 13).

9

Optimizer: Join Operators

Objectives

After completing this lesson, you should be able to:

- Describe the SQL operators for joins
- List the possible access paths

This lesson helps you to understand the execution plans related to join operations.

Lesson Agenda

- Join Methods
  - Nested loops join
  - Sort-merge join
  - Hash join
  - Cartesian join
- Join Operation Types
  - Equijoin/Natural – Nonequijoin
  - Outer join (Full, Left, and Right)
  - Semijoin: `EXISTS` subquery
  - Antijoin: `NOT IN` subquery

How the Query Optimizer Executes Join Statements

The factors considered by the optimizer:

- Access paths
- Join methods
- Join orders

To choose an execution plan for a join statement, the optimizer must make the following interrelated decisions:

- **Access paths:** For simple statements, the optimizer must choose an access path to retrieve data from each table in the join statement.
- **Join methods:** To join each pair of row sources, Oracle Database must perform a join operation. The possible join methods are nested loops, sort-merge, and hash joins. A Cartesian join requires one of the preceding join methods.
- **Join order:** To execute a statement that joins more than two tables, Oracle Database joins two of the tables, and then joins the resulting row source to the next table. This process continues until all tables are joined into the result.

Note: Access paths have been covered in the lesson titled "Optimizer: Table and Index Access Paths."

Join Methods

A join:

- Defines the relationship between two row sources
- Is a method of combining data from two data sources
- Is controlled by join predicates, which define how the objects are related
- Join methods:
  – Nested loops
  – Sort-merge join
  – Hash join
  – Cartesian join

Join predicate

Nonjoin predicate

```
SELECT e.ename, d.dname
FROM dept d JOIN emp e
USING (deptno)
WHERE e.job = 'ANALYST' OR e.empno = 9999;
```


A join is a statement that retrieves data from multiple tables. A row source is a set of data that can be accessed in a query. It can be a table, an index, a non-mergeable 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). In a join, one row set is called inner, and the other is called outer.

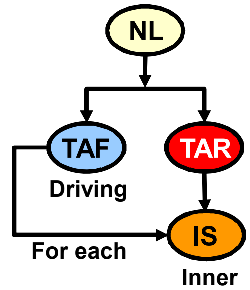
The optimizer supports different join methods such as the following:

- **Nested loops join:** Is 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.

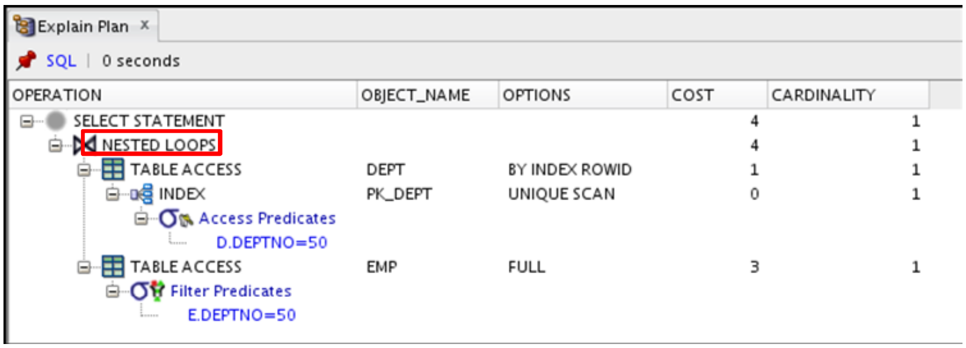
- **Hash join:** Is used for joining large data sets. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows. This method is best used when the smaller table fits in the available memory. The cost is then limited to a single read pass over the data for the two tables.
- **Cartesian join:** Is used when one or more of the tables do not have any join conditions to any other tables in the statement

Nested Loops Join

- The driving row source is scanned.
- Each row returned drives a lookup in the inner row source.
- Joining rows are then returned.




```
select ename, e.deptno, d.deptno, d.dname
from emp e join dept d
on e.deptno = d.deptno and d.deptno=30;
```



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			4	1
NESTED LOOPS			4	1
TABLE ACCESS	DEPT	BY INDEX ROWID	1	1
INDEX	PK_DEPT	UNIQUE SCAN	0	1
Access Predicates				
D.DEPTNO=50				
TABLE ACCESS	EMP	FULL	3	1
Filter Predicates				
E.DEPTNO=50				

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 row ID.

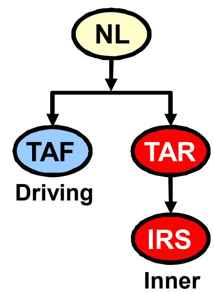
Any nonjoin predicates on the inner table are considered after this initial retrieval, unless a composite index combining both the join and the nonjoin predicate is used.

The code to emulate a nested loops 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 loops joins when joining a small number of rows, with a good driving condition between the two tables. And if the join condition is an efficient way of accessing the second table. 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




```
select ename, e.deptno, d.deptno, d.dname
from emp e join dept d
on e.deptno = d.deptno and ename like 'A%';
---------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |     2 |    84 |     5
|   1 |  TABLE ACCESS BY INDEX ROWID | DEPT   |     1 |    22 |     1
|   2 |   NESTED LOOPS               |        |     2 |    84 |     5
|*  3 |    TABLE ACCESS FULL         | EMP    |     2 |    40 |     3
|*  4 |    INDEX RANGE SCAN          | IDEPT  |     1 |       |     0
---------------------------------------------------------------------
   3 - filter("E"."ENAME" LIKE 'A%')
   4 - access("E"."DEPTNO"="D"."DEPTNO")
```


Oracle 9*i*R2 introduced a mechanism called nested loops prefetching. The idea is to improve I/O utilization, and therefore response time, of index access with table lookup by batching row ID 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 available only when the inner access path is an index range scan and not if the inner access path is an index unique scan.

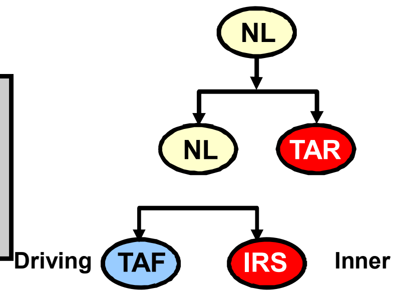
The prefetching mechanism is used by table lookup. When an index access path is chosen and the query cannot be satisfied by the index alone, the data rows indicated by the `ROWID` must also be fetched. This `ROWID` to data row access (table lookup) is improved by using data block prefetching, which involves reading an array of blocks that are pointed at by an array of qualifying `ROWID`s.

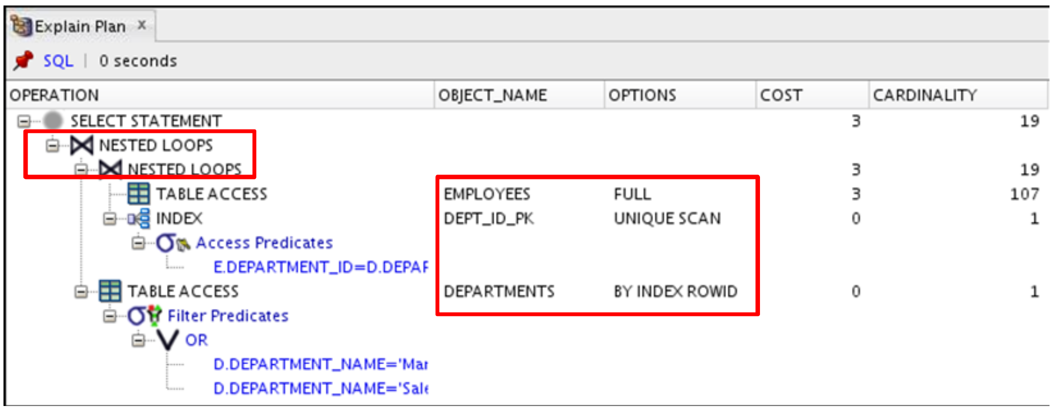
Without data block prefetching, accessing a large number of rows using a poorly clustered B*-tree index could be expensive. Each row accessed by the index would likely be in a separate data block and thus would require a separate I/O operation.

With data block prefetching, the system delays data block reads until multiple rows specified by the underlying index are ready to be accessed. The system then retrieves multiple data blocks simultaneously, rather than reading a single data block at a time.

Nested Loops Join: 11*g* Implementation

```
select e.first_name, e.last_name, e.salary,
d.department_name
from employees e join departments d
on d.department_name IN ('Marketing', 'Sales')
and e.department_id = d.department_id;
```





OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			3	19
NESTED LOOPS				
NESTED LOOPS			3	19
TABLE ACCESS	EMPLOYEES	FULL	3	107
INDEX	DEPT_ID_PK	UNIQUE SCAN	0	1
Access Predicates				
E.DEPARTMENT_ID=D.DEPAF				
TABLE ACCESS	DEPARTMENTS	BY INDEX ROWID	0	1
Filter Predicates				
OR				
D.DEPARTMENT_NAME='Mar				
D.DEPARTMENT_NAME='Sale				

ORACLE

Oracle Database 11*g* introduced a new way of performing joins with NESTED LOOPS operators to reduce overall latency for physical I/O. When an index or a table block is not in the buffer cache and is needed to process a join, a physical I/O is required. The current release of Oracle Database can batch multiple physical I/O requests and process them using a vector I/O instead of processing them one at a time.

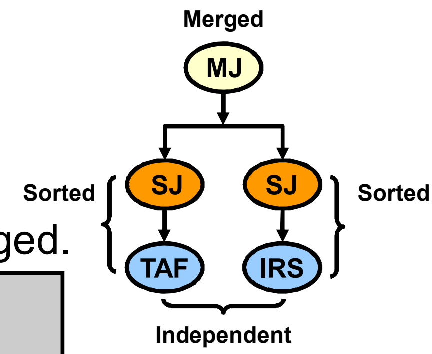
With this NESTED LOOPS implementation, the system first performs a NESTED LOOPS join between the other table and the index. This produces a set of ROWIDs that you can use to look up the corresponding rows from the table with the index. Instead of going to the table for each ROWID produced by the first 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 reads each block in the inner table only once.

Note: In some cases, a second join row source is not allocated, and the execution plan looks the same as it did before Oracle Database 11*g*. The following list describes such cases:

- All the columns needed from the inner side of the join are present in the index, and there is no table access required. In this case, Oracle Database allocates only one join row source.

- The order of the rows returned might be different from the order returned in previous releases. Thus, when Oracle Database tries to preserve a specific ordering of the rows, for example to eliminate the need for an `ORDER BY` sort, Oracle Database might use the original implementation for nested loops joins.
- The `OPTIMIZER_FEATURES_ENABLE` initialization parameter is set to a release before Oracle Database 11*g*. In this case, Oracle Database uses the original implementation for nested loops joins.

Sort-Merge Join



- The first and second row sources are sorted by the same sort key.
- Sorted rows from both tables are merged.

```
select ename, e.deptno, d.deptno, dname
from emp e join dept d
on e.deptno = d.deptno and ename > 'A'
```

Explain Plan ×

SQL | 2.619 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			5	14
MERGE JOIN			5	14
TABLE ACCESS	DEPT	BY INDEX ROWID	2	4
INDEX	PK_DEPT	FULL SCAN	1	4
SORT		JOIN	3	14
Access Predicates				
E.DEPTNO=D.DEPTNO				
Filter Predicates				
E.DEPTNO=D.DEPTNO				
TABLE ACCESS	EMP	FULL	3	14
Filter Predicates				
ENAME>'A'				

ORACLE

Sort-merge joins can join rows from two independent sources. Sort-merge joins perform better than nested loops joins for large data sets.

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 by using any access and filter predicates, and sort it on the join columns.
2. Get the second data set by 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 the first row to the next operation is returned. Sorting could make this join technique expensive, especially if sorting cannot be performed in memory.

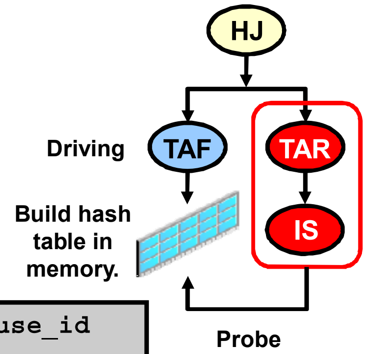
The optimizer can select a sort-merge join over a hash join for joining large amounts of data if any of the following conditions are true:

- The join condition between two tables is not an equijoin.
- Sorts are already required by previous operations.

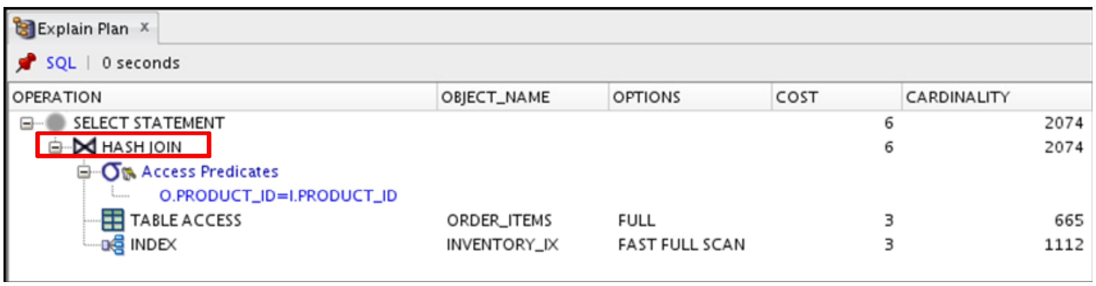
Note: Sort-merge joins are useful when the join condition between two tables is an inequality condition (but not a non-equality), such as `<`, `<=`, `>`, or `>=`.

Hash Join

- The smallest row source is used to build a hash table.
- The second row source is hashed and checked against the hash table.




```
select o.product_id, order_id, quantity, i.warehouse_id
from order_items o join inventories i
on o.product_id=i.product_id;
```


Explain Plan
SQL | 0 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			6	2074
HASH JOIN			6	2074
Access Predicates				
O.PRODUCT_ID=I.PRODUCT_ID				
TABLE ACCESS	ORDER_ITEMS	FULL	3	665
INDEX	INVENTORY_IX	FAST FULL SCAN	3	1112

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 a 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.

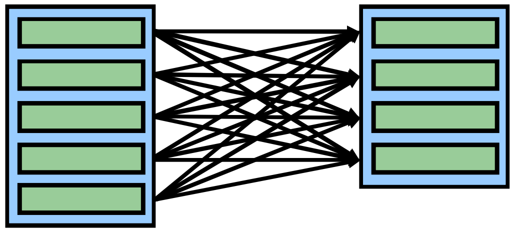
The system starts to read the second set of rows, using whatever access mechanism is most appropriate for acquiring the rows, and it uses the same hash function on the join column or columns to calculate the number of the relevant hash bucket. The system then checks to see if there are any rows in that 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 the rows from the smallest row source to return the first row to the next operation.

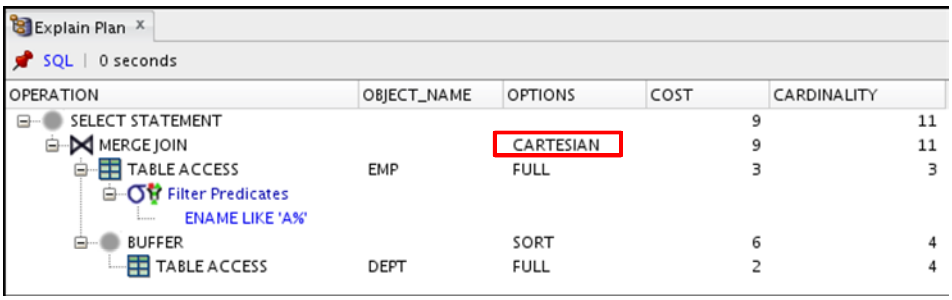
Note: The optimizer uses a hash join to join two tables if they are joined using an equijoin and if either of the following conditions is true:

- A large amount of data must be joined.
- A large fraction of a small table must be joined.

Cartesian Join



```
select ename, e.deptno, d.deptno, dname
from emp e join dept d
on ename like 'A%';
```



Explain Plan ×

SQL | 0 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			9	11
MERGE JOIN		CARTESIAN	9	11
TABLE ACCESS	EMP	FULL	3	3
Filter Predicates				
ENAME LIKE 'A%'				
BUFFER		SORT	6	4
TABLE ACCESS	DEPT	FULL	2	4

ORACLE

A Cartesian join is used when one or more of the tables do 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 loops join 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 a single-row row source (guaranteed by a unique index, for example) joined to some other table.

Lesson Agenda

- Join Methods
  - Nested loops join
  - Sort-merge join
  - Hash join
  - Cartesian join
- Join Operation Types
  - Equijoin/Natural – Nonequijoin
  - Outer join (Full, Left, and Right)
  - Semijoin: `EXISTS` subquery
  - Antijoin: `NOT IN` subquery

Join Types

- A join operation combines the output from two row sources and returns one resulting row source.
- Join operation types include the following:
  - Join (Equijoin/Natural – Nonequijoin)
  - Outer join (Full, Left, and Right)
  - Semijoin: `EXISTS` subquery
  - Antijoin: `NOT IN` subquery
  - Star join (Optimization)

Join operation types include the following:

- **Join (equijoin and nonequijoin):** Returns rows that match the predicate join
- **Outer join:** Returns rows that match the predicate join and row when no match is found
- **Semijoin:** Returns rows that match the `EXISTS` subquery. Find one match in the inner table, and then stop the search.
- **Antijoin:** Returns rows with no match in the `NOT IN` subquery. Stop as soon as one match is found.
- **Star join:** Is not a join type, but only 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 optimization algorithms used to flatten subquery constructs such that they can be resolved in a join-like way.

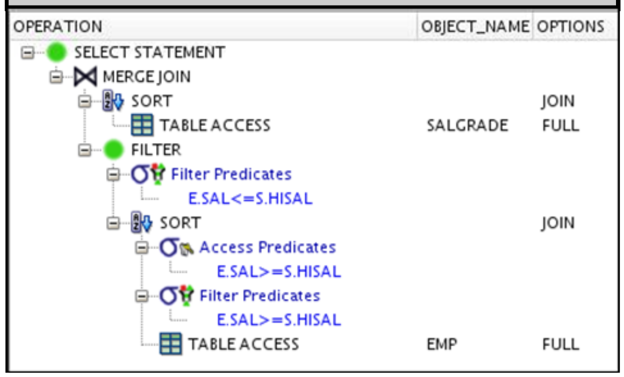
Equijoins and Nonequijoins

Equijoin

```
SELECT e.ename, e.sal, s.grade
FROM   emp e JOIN salgrade s
ON e.sal = s.hisal;
```

Nonequijoin

```
SELECT e.ename, e.sal, s.grade
  FROM emp e INNER JOIN salgrade s
    ON e.sal BETWEEN s.losal and s.hisal;
```



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. Examples 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.

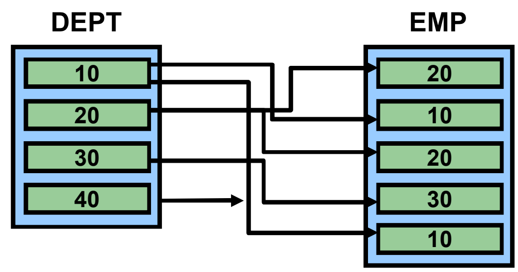
Outer Joins

An outer join returns a row even if no match is found.

1

```
SELECT d.deptno,d.dname,e.empno,e.ename
FROM   emp e RIGHT OUTER JOIN dept d
ON   e.deptno = d.deptno;
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		OUTER
TABLE ACCESS	DEPT	BY INDEX RO
INDEX	PK_DEPT	FULL SCAN
SORT		JOIN
Access Predicates		
E.DEPTNO(+)=D.DEPTNO		
Filter Predicates		
E.DEPTNO(+)=D.DEPTNO		
TABLE ACCESS	EMP	FULL



2

```
SELECT /*+ USE_HASH(e d) */
d.deptno,d.dname,e.empno,e.ename
FROM   emp e RIGHT OUTER JOIN dept d
ON   e.deptno = d.deptno;
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH JOIN		OUTER
Access Predicates		
E.DEPTNO(+)=D.DEPTNO		
TABLE ACCESS	DEPT	FULL
TABLE ACCESS	EMP	FULL

3

```
SELECT /*+ USE_NL(e d) */
d.deptno,d.dname,e.empno,e.ename
FROM   emp e RIGHT OUTER JOIN dept d
ON   e.deptno = d.deptno;
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
NESTED LOOPS		OUTER
TABLE ACCESS	DEPT	FULL
TABLE ACCESS	EMP	FULL
Filter Predicates		
E.DEPTNO(+)=D.DEPTNO		

ORACLE

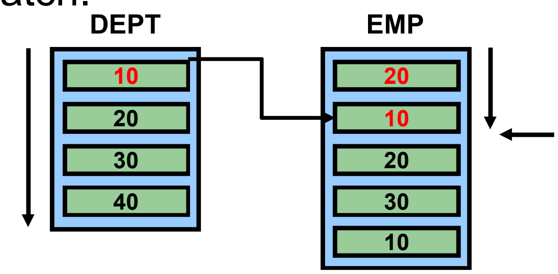
The simple join is the most commonly used join within the system. Other joins open up extra functionality, but have much more specialized uses. The outer join operator is placed on the deficient side of the query; that is, it is placed against the table that has the missing join information. Consider 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 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 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 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 look only for the first match.




```
SELECT deptno, dname
FROM   dept
WHERE  EXISTS (SELECT 1 FROM emp WHERE emp.deptno=dept.deptno);
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		SEMI
TABLE ACCESS	DEPT	BY INDEX RO
INDEX	PK_DEPT	FULL SCAN
SORT		UNIQUE
Access Predicates		
EMP.DEPTNO=DEPT.DEPTNO		
Filter Predicates		
EMP.DEPTNO=DEPT.DEPTNO		
TABLE ACCESS	EMP	FULL

Semijoins return a result when you encounter 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.

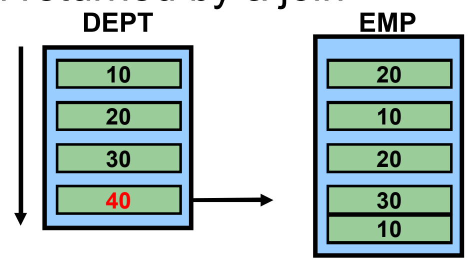
In the slide diagram, for each `DEPT` record, only the first matching `EMP` record is returned as a join result. This prevents scanning huge numbers of duplicate rows in a table when all you are interested in is if there are any matches.

When the subquery is not unnested, a similar result could be achieved by using a `FILTER` operation, scanning a row source until a match is found, and then returning it.

Note: A semijoin can always use a merge join. The optimizer may choose the nested loops or hash joins methods to perform semijoins as well.

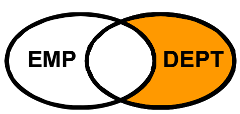
Antijoins

Reverse of what would have been returned by a join




```
SELECT deptno, dname
FROM   dept
WHERE  deptno not in
       (SELECT deptno FROM emp);
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		ANTI NA
SORT		JOIN
TABLE ACCESS	DEPT	BY INDEX RO...
INDEX	PK_DEPT	FULL SCAN
SORT		UNIQUE
Access Predicates		
DEPTNO=DEPTNO		
Filter Predicates		
DEPTNO=DEPTNO		
TABLE ACCESS	EMP	FULL




```
SELECT deptno, dname FROM dept
WHERE deptno IS NOT NULL AND
deptno NOT IN
 (SELECT /*+ HASH_AJ */ deptno FROM
emp WHERE deptno IS NOT NULL);
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH JOIN		ANTI
Access Predicates		
DEPTNO=DEPTNO		
TABLE ACCESS	DEPT	FULL
TABLE ACCESS	EMP	FULL
Filter Predicates		
DEPTNO IS NOT NULL		

Antijoins return rows that fail to match (`NOT IN`) the subquery on the right side. For example, an antijoin can select a list of departments that do not have any employees.

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.

Quiz

The optimizer uses nested loops joins when joining a small number of rows, with a good driving condition between two tables.

a. True

b. False

Answer: a

Quiz

The ________ join is used when one or more of the tables do not have any join conditions to any other tables in the statement.

a. Hash
b. Cartesian
c. Nonequijoin
d. Outer

Answer: b

Quiz

The ________ join looks only for the first match.

a. Hash
b. Cartesian
c. Semi
d. Outer

Answer: c

Quiz

In a hash join, the _______ row source is used to build a hash table.

a. Biggest
b. Smallest
c. Sorted
d. Unsorted

Answer: b

Summary

In this lesson, you should have learned to:

- Describe the SQL operators for joins
- List the possible access paths

Practice 9: Overview

This practice covers using different join paths for better optimization.

10

Other Optimizer Operators

Objectives

After completing this lesson, you should be able to:

- Describe result cache operator
- Describe SQL operators for:
  - Clusters
  - In-List
  - Sorts
  - Filters
  - Set Operations

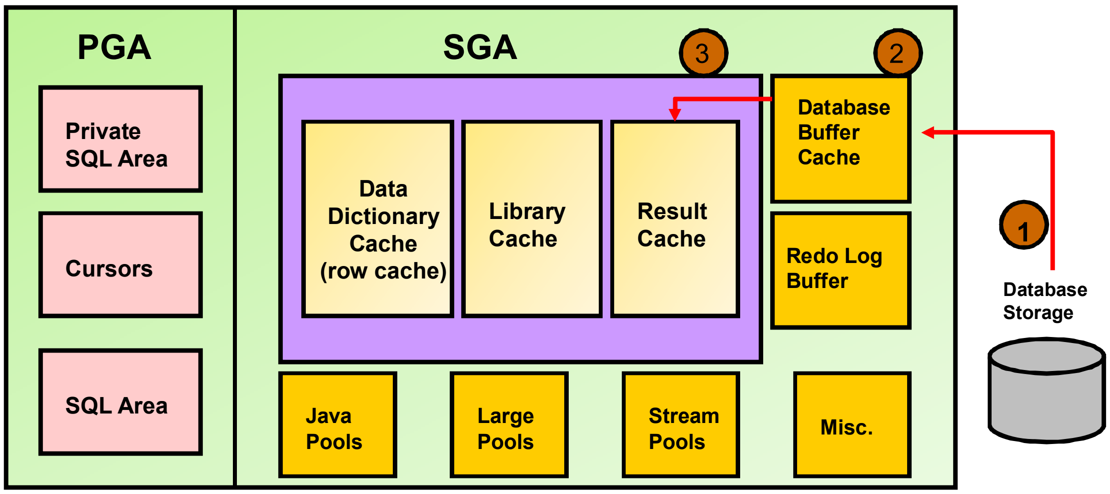
This lesson helps you to understand the execution plans that use the common operators of other access methods.

Lesson Agenda

- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

Result Cache Operator

A result cache is an area of memory either in SGA or client application memory that stores the results of a database query or query block for reuse.



The SQL query result cache enables explicit caching of query result sets and query fragments in database memory. A dedicated memory buffer stored in the shared pool can be used for storing and retrieving the cached results. The query results stored in this cache become invalid when data in the database objects that are accessed by the query is modified. Although the SQL query cache can be used for any query, good candidate statements are the ones that need to access a very high number of rows to return only a fraction of them. This is mostly the case for data-warehousing applications.

When a query is executed for the very first time, the user's process searches for the data in the database buffer cache. If data exists (because someone else had retrieved this data before), it uses it; otherwise, it performs an I/O operation to retrieve data from the data file on disk into the buffer cache, and from this data, the final result set is built.

The Result Cache can be managed either on the client side or the server side. A client side Result Cache implementation would require the application to use the Oracle Call Interface (OCI) calls. Comparatively, server side implementation is much simpler. In the course, the focus is on the server side implementation of the Result Cache.

If you want to use the query result cache and the `RESULT_CACHE_MODE` initialization parameter is set to `MANUAL`, you must explicitly specify the `RESULT_CACHE` hint in your query. For a server side implementation, the same query could be executed with `/*+ RESULT_CACHE */` hint or the `result_cache_mode` parameter could be set to `AUTO`.

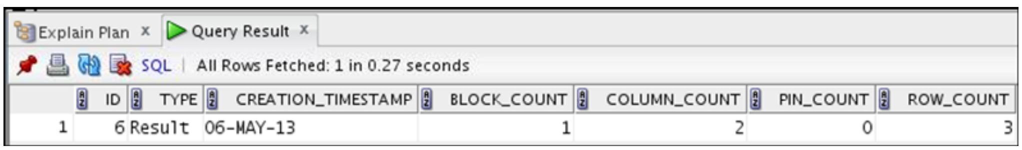
This hint introduces the ResultCache operator in the execution plan for the query. When you execute the query, the ResultCache operator looks up the result cache memory to check whether the result for the query already exists in the cache. If it exists, the result is retrieved directly out of the cache. If it does not yet exist in the cache, the query is executed, the result is returned as output, and it is also stored in the result cache memory. If the RESULT_CACHE_MODE initialization parameter is set to FORCE, and you do not want to store the result of a query in the result cache, you must use the NO_RESULT_CACHE hint in your query.

The DBMS_RESULT_CACHE package provides statistics, information, and operators that enable you to manage memory allocation for the server result cache. Use the DBMS_RESULT_CACHE package to perform operations such as retrieving statistics on cache memory usage and flushing the cache.

Using RESULT_CACHE

```
SELECT /*+ RESULT_CACHE */ deptno, AVG(sal)
FROM emp
GROUP BY deptno;
```


```
SELECT id, type, creation_timestamp, block_count,
column_count, pin_count, row_count FROM
V$RESULT_CACHE_OBJECTS
WHERE cache_id = '96xqvkkh2wguzbthnjkxnhgz55';
```



ORACLE

The parameters `client_result_cache_lag` and `client_result_cache_size` are used to configure the Result Cache at the client side. The other parameters are used for configuring the Result Cache at the server side.

The size of the Result Cache on the server is determined by two parameters: `result_cache_max_result` and `result_cache_max_size`.

The example in the slide shows a query of employees that uses the `RESULT_CACHE` hint to retrieve rows from the server result cache. Here, the results are retrieved directly from the cache, as indicated in step 1 of the execution plan. The value in the `Name` column is the cache ID of the result. The `CACHE_ID` for a query does not match the `SQL_ID` used to identify the query in the library cache and contained in `V$SQL`. Unlike the `SQL_ID`, which is generated for every SQL query that is executed against an Oracle database, the `CACHE_ID` is for an area or bucket in the Result Cache section of the shared pool that stores the end result of the query.

There are several views to monitor information related to the Result Cache. The objects that are related to the Result Cache can be obtained from the `V$RESULT_CACHE_OBJECTS` view. The following query helps verify the result set contained in the Result Cache for the `CACHE_ID` named `96xqvkkh2wguzbthnjkxnhgz55`.

Using Result Cache Table Annotations

- You can also use table annotations to control result caching.
- You can use table annotations to avoid the necessity of adding result cache hints to queries at the application level.

```
CREATE TABLE sales (...) RESULT_CACHE (MODE DEFAULT);
SELECT prod_id, SUM(amount_sold)
FROM sales
GROUP BY prod_id ORDER BY prod_id;
```

```
ALTER TABLE sales RESULT_CACHE (MODE FORCE);
SELECT prod_id, SUM(amount_sold)
FROM sales
GROUP BY prod_id HAVING prod_id=136;

SELECT /*+ NO_RESULT_CACHE */ *
FROM sales ORDER BY time_id DESC;
```


Table annotations affect the entire query, not query segments. Because a table annotation has a lower precedence than a SQL result cache hint, you can override table and session settings by using hints at the query level.

The `DEFAULT` table annotation prevents the database from caching results at the table level.

- `DEFAULT`: If at least one table in a query is set to `DEFAULT`, result caching is *not* enabled at the table level for this query.
- `FORCE`: If all the tables of a query are marked as `FORCE`, the query result is considered for caching. The `FORCE` table annotation forces the database to cache results at the table level.

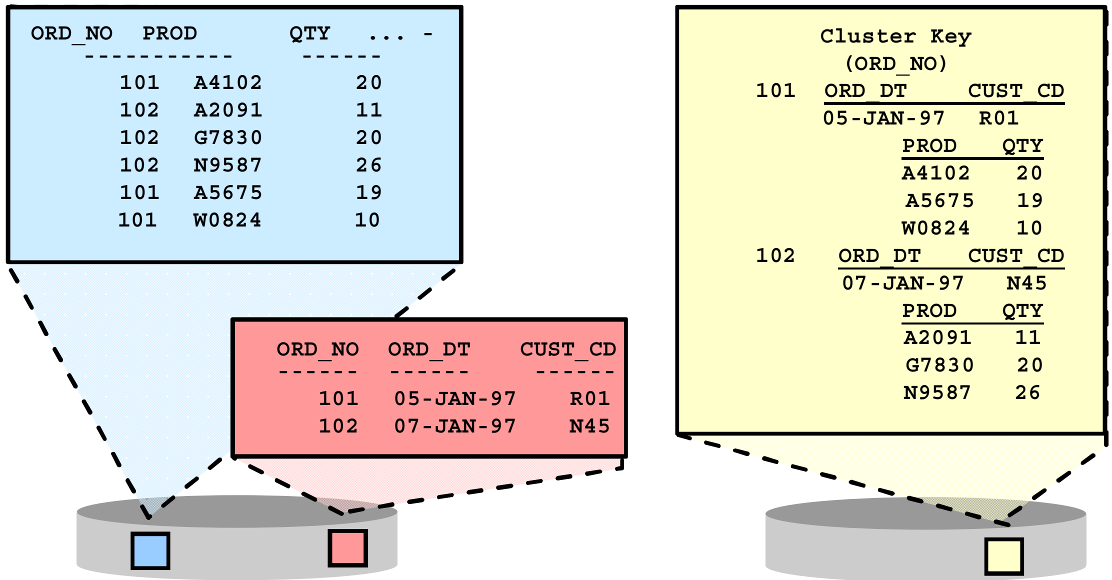
The example in the slide shows a `CREATE TABLE` statement that uses the `DEFAULT` table annotation to create a table called `sales` and a `SELECT` query on this table. In this example, the `sales` table is created with a table annotation that disables result caching. The example also shows a query of the `sales` table, whose results are not considered for caching because of the table annotation.

The second example includes two queries of the `sales` table. The first query, which is frequently used and returns a few rows, is eligible for caching because of the table annotation. The second query, which is a one-time query that returns many rows, uses a hint to prevent result caching.

Lesson Agenda

- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

Clusters



Unclustered ORDERS and ORDER_ITEMS tables

Clustered ORDERS and ORDER_ITEMS tables

Clusters are an optional method for 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 ORDERS and ORDER_ITEMS tables share the ORDER_ID column. When you cluster the ORDERS and ORDER_ITEMS tables, the system physically stores all rows for each order from both the ORDERS and ORDER_ITEMS tables in the same data blocks.

Cluster index: A cluster index is an index defined specifically for a cluster. Such an index contains an entry for each cluster key value. To locate a row in a cluster, the cluster index is used to find the cluster key value, which points to the data block associated with that cluster key value. Therefore, the system accesses a given row with a minimum of two I/Os.

Hash clusters: Hashing is an optional way of storing table data to improve the performance of data retrieval. To use hashing, you create a hash cluster and load tables into the cluster. The system physically stores the rows of a table in a hash cluster and retrieves them according to the results of a hash function. The key of a hash cluster (just as the key of an index cluster) can be a single column or composite key. To find or store a row in a hash cluster, the system applies the hash function to the row’s cluster key value; the resulting hash value corresponds to a data block in the cluster, which the system then reads or writes on behalf of the issued statement.

Note: Hash clusters are a better choice than an indexed table or index cluster when a table is queried frequently with equality queries.

When Are Clusters Useful?

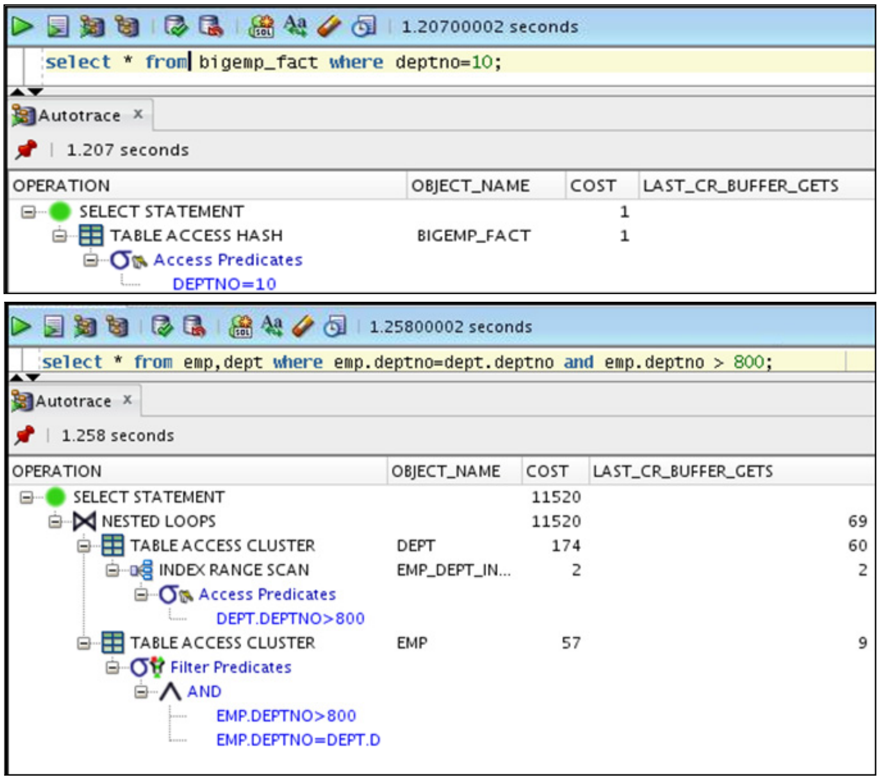
- Index cluster:
  - Tables are always joined on the same keys.
  - The size of the table is not known.
  - It can be used in any type of search.
- Hash cluster:
  - Tables are always joined on the same keys.
  - Storage for all cluster keys is allocated initially.
  - It can be used in either equality (=) or nonequality (<>) searches.
- Single-table hash cluster:
  - This is fastest way to access a large table with an equality search.
- Sorted hash cluster:
  - This is used only for equality search.

- Index clusters allow row data from one or more tables that share a cluster key value to be stored in same block. You can locate these rows by using a cluster index, which has one entry per cluster key value and *not* for each row. Therefore, the index is smaller and less costly to access for finding multiple rows. The rows with the same key are in a small group of blocks. This means that in an index cluster, the clustering factor is very good and provides clustering for data from multiple tables that share the same join key. The smaller index and smaller group of blocks reduce the cost of access by reducing block visits to the buffer cache. Index clusters are useful when the size of the tables is not known in advance (for example, creating a new table rather than converting an existing one whose size is stable), because a cluster bucket is created only after a cluster key value is used. They are also useful for all filter operations or searches. Note that full table scans do not perform well on a table in a multiple-table cluster because it has more blocks than the table would have if it were created as a heap table.
- Hash clusters allow row data from one or more tables that share a cluster key value to be stored in the same block. You can locate these rows by using a system- or user-provided hashing function or by using the cluster key value. The cluster key value method assumes that its value is evenly distributed so that row access is faster than with index clusters. Table rows with the same cluster key values hash into the same cluster buckets and can be stored in the same block or small group of blocks.

This means that in a hash cluster, the clustering factor is very good and a row may be accessed by its key with only one block visit and without needing an index. Hash clusters allocate all the storage for all the hash buckets when the cluster is created, so they may waste space. They also do not perform well other than on equality searches or non-equality searches. Like index clusters, if they contain multiple tables, full scans are more expensive for the same reason.

- Single-table hash clusters are similar to a hash cluster, but are optimized in the block structures for access to a single table, thereby providing the fastest possible access to a row other than by using a row ID filter. Because they have only one table, full scans, if they happen, cost as much as they would in a heap table.
- Sorted hash clusters are designed to reduce the costs of accessing ordered data by using a hashing algorithm on the hash key. Accessing the first row that matches the hash key may be less costly than using an index-organized table (IOT) for a large table because it saves the cost of a B*-tree probe. All the rows that match on a particular hash key (for example, account number) are stored in the cluster in the order of the sort key or keys (for example, phone calls), thereby eliminating the need for a sort to process the `ORDER BY` clause. These clusters are very good for batch reporting, billing, and so on.

Cluster Access Path: Examples



ORACLE

The example in the slide shows you two different cluster access paths.

In the access path example at the top, a hash scan is used to locate rows in a hash cluster, based on a hash value. In a hash cluster, all rows with the same hash value are stored in the same data block. To perform a hash scan, the system first obtains the hash value by applying a hash function to a cluster key value specified by the statement. The system then scans the data blocks containing rows with that hash value.

The second access path example assumes that a cluster index was used to cluster both the `EMP` and `DEPT` tables. In this case, a cluster scan is used to retrieve, from a table stored in an indexed cluster, all rows that have the same cluster key value. In an indexed cluster, all rows with the same cluster key value are stored in the same data block. To perform a cluster scan, the system first obtains the `ROWID` of one of the selected rows by scanning the cluster index. The system then locates the rows based on this `ROWID`.

Note: You see examples of how to create clusters in the practices for this lesson.

Lesson Agenda

- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

Sorting Operators

- SORT operator:
  - AGGREGATE: Retrieves a single row using a group function
  - UNIQUE: Removes duplicates
  - JOIN: Precedes a merge join
  - GROUP BY, ORDER BY: For these operators
- HASH operator:
  - GROUP BY: For this operator
  - UNIQUE: Equivalent to SORT UNIQUE
- If you want ordered results, *always* use ORDER BY.

Sort operations result when users specify an operation that requires a sort. Commonly encountered operations include the following:

- SORT AGGREGATE does not involve a sort. It retrieves a single row that is the result of applying a group function to a group of selected rows. Operations such as COUNT and MIN are shown as SORT AGGREGATE.
- SORT UNIQUE sorts output rows to remove duplicates. It occurs if a user specifies a DISTINCT clause or if an operation requires unique values for the next step.
- SORT JOIN happens during a sort-merge join, if the rows need to be sorted by the join key.
- SORT GROUP BY is used when aggregates are computed for different groups in the data. The sort is required to separate the rows into different groups.
- SORT ORDER BY is required when the statement specifies an ORDER BY that cannot be satisfied by one of the indexes.

- HASH GROUP BY hashes a set of rows into groups for a query with a GROUP BY clause.
- HASH UNIQUE hashes a set of rows to remove duplicates. It occurs if a user specifies a DISTINCT clause or if an operation requires unique values for the next step. This is similar to SORT UNIQUE.

Note: Several SQL operators cause implicit sorts (or hashes since Oracle Database 10*g*, Release 2), such as DISTINCT, GROUP BY, UNION, MINUS, and INTERSECT. However, do not rely on these SQL operators to return ordered rows. If you want to have rows ordered, use the ORDER BY clause.

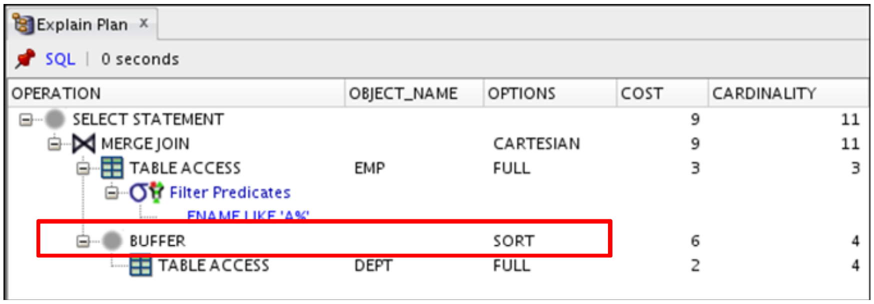
Review MOS 316467.1. MIN/MAX index is not used if the query has multiple MIN/MAX functions.

Lesson Agenda

- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

Buffer Sort Operator

```
SELECT ename, emp.deptno, dept.deptno, dname
FROM emp, dept
WHERE ename like 'A%';
```



The BUFFER SORT operator uses a temporary table or a sort area in memory to store intermediate data. However, the data is not necessarily sorted.

The BUFFER SORT operator is needed if there is an operation that needs all the input data before it can start. (See “Cartesian Join.”)

So BUFFER SORT uses the buffering mechanism of a traditional sort, but it does not do the sort itself. The system simply buffers the data, in the User Global Area (UGA) or Program Global Area (PGA), to avoid multiple table scans against real data blocks.

The whole sort mechanism is reused, including the swap to disk when not enough sort area memory is available, but without sorting the data.

The difference between a temporary table and a buffer sort is as follows:

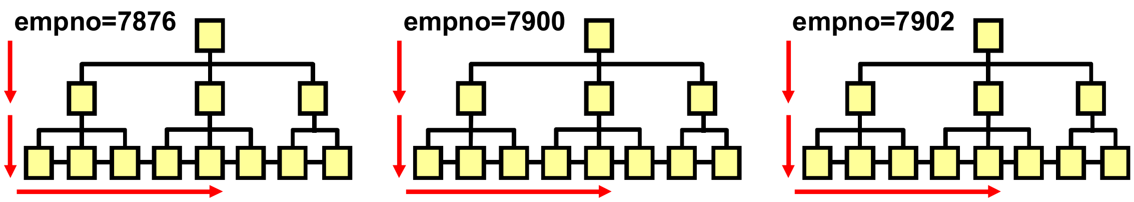
- A temporary table uses System Global Area (SGA).
- A buffer sort uses UGA.

Lesson Agenda

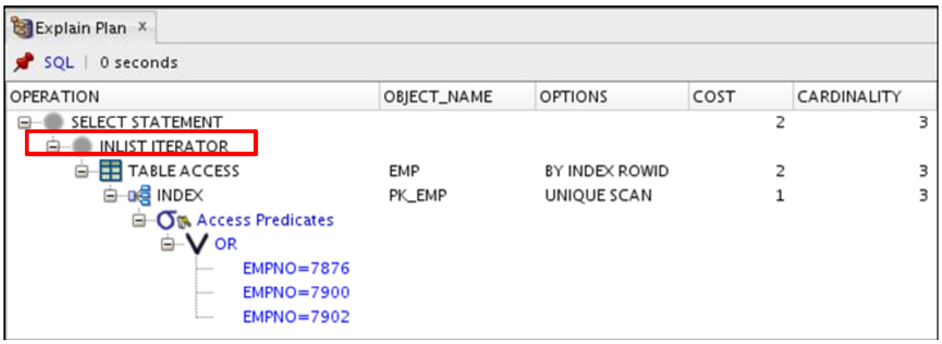
- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

Inlist Iterator

Every value executed separately




```
SELECT * FROM emp WHERE empno IN (7876, 7900, 7902);
```



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	3
INLIST ITERATOR				
TABLE ACCESS	EMP	BY INDEX ROWID	2	3
INDEX	PK_EMP	UNIQUE SCAN	1	3
Access Predicates				
OR				
EMPNO=7876				
EMPNO=7900				
EMPNO=7902				

It is used when a query contains an `IN` clause with values or multiple equality predicates on the same column linked with `OR`s.

The `INLIST ITERATOR` operator iterates over the enumerated value list, and every value is executed separately.

The execution plan is identical to the result of a statement with an equality clause instead of `IN`, except for one additional step. The extra step occurs when `INLIST ITERATOR` feeds the equality clause with unique values from the list.

You can view this operator as a `FOR LOOP` statement in PL/SQL. In the example in the slide, you iterate the index probe over three values: 7867, 7900, and 7902.

Also, it is a function that uses an index, which is scanned for each value in the list. An alternative handling is `UNION ALL` of each value or a `FILTER` of the values against all the rows; this is significantly more efficient.

The optimizer uses an `INLIST ITERATOR` when an `IN` clause is specified with values, and the optimizer finds a selective index for that column. If there are multiple `OR` clauses using the same index, the optimizer selects this operation rather than `CONCATENATION` or `UNION ALL`, because it is more efficient.

Lesson Agenda

- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

View Operator

```
create view V as select /*+ NO_MERGE */ DEPTNO, sal from emp ;
select * from V;
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			3	14
VIEW	V		3	14
TABLE ACCESS	EMP	FULL	3	14

```
select v.*,d.dname from (select DEPTNO, sum(sal) SUM_SAL
from emp group by deptno) v, dept d where v.deptno=d.deptno;
```

Script Output × | Query Result × | Explain Plan ×

SQL | 0 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			5	3
MERGE JOIN			5	3
TABLE ACCESS	DEPT	BY INDEX ROWID	2	4
INDEX	PK_DEPT	FULL SCAN	1	4
SORT		JOIN	3	3
Access Predicates				
V.DEPTNO=D.DEPTNO				
Filter Predicates				
V.DEPTNO=D.DEPTNO				
VIEW			3	3
HASH		GROUP BY	3	3
TABLE ACCESS	EMP	FULL	3	14

ORACLE

Each query produces a variable set of data in the form of a table. A view simply gives a name to this set of data.

When views are referenced in a query, the system can handle them in two ways. If a number of conditions are met, they can be merged into the main query. This means that the view text is rewritten as a join with the other tables in the query. Views can also be left as stand-alone views and selected from directly as in the case of a table. Predicates can also be pushed into or pulled out of the views as long as certain conditions are met.

When a view is not merged, you can see the `VIEW` operator. The view operation is executed separately. All rows from the view are returned, and the next operation can be performed.

Sometimes a view cannot be merged and must be executed independently in a separate query block. In this case, you can also see the `VIEW` operator in the explain plan. The `VIEW` keyword indicates that the view is executed as a separate query block. For example, views containing `GROUP BY` functions cannot be merged.

The second example in the slide shows a non-mergeable inline view. An inline view is basically a query within the `FROM` clause of your statement.

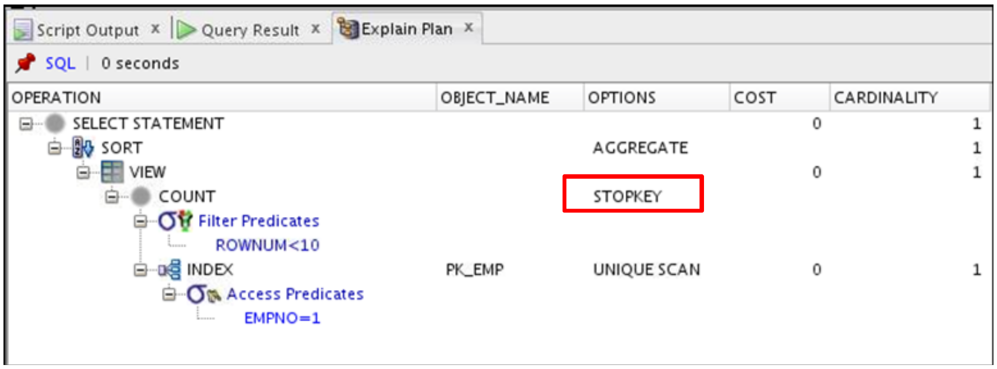
Basically, this operator collects all rows from a query block before they can be processed by higher operations in the plan.

Lesson Agenda

- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

Count Stop Key Operator

```
SELECT count(*)
FROM (SELECT /*+ NO_MERGE */ *
      FROM emp WHERE empno ='1' and rownum < 10);
```



COUNT STOPKEY limits the number of rows returned. The limitation is expressed by the ROWNUM expression in the WHERE clause. It terminates the current operation when the count is reached.

Note: The cost of this operator depends on the number of occurrences of the values you try to retrieve. If the value appears very frequently in the table, the count is reached quickly. If the value is very infrequent, and there are no indexes, the system has to read most of the table's blocks before reaching the count.

Lesson Agenda

- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

Min/Max and First Row Operators

```
SELECT MIN(quantity_on_hand)
FROM INVENTORIES
WHERE quantity_on_hand < 500;
```

Explain Plan ×

SQL | 0 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	1
SORT		AGGREGATE		1
FIRST ROW			2	1
INDEX	INV_QTY_INDEX	RANGE SCAN (MIN/MAX)	2	1
Access Predicates				
QUANTITY_ON_HAND<500				

ORACLE

`FIRST ROW` retrieves only the first row selected by a query. It stops accessing the data after the first value is returned. This optimization was introduced in Oracle 8*i*, and it works with the index range scan and the index full scan.

In the example in the slide, it is assumed that there is an index on the `quantity_on_hand` column.

Note: Review MOS 316467.1. `MIN/MAX` index is not used if the query has multiple `MIN/MAX` functions.

Lesson Agenda

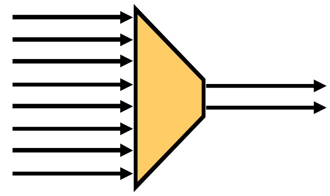
- Result Cache Operator
- Clusters
  - When Are Clusters Useful?
  - Cluster Access Path: Examples
- Sorting Operators
- Buffer Sort Operator
- Inlist Iterator
- View Operator
- Count Stop Key Operator
- Min/Max and First Row Operators
- Other N-Array Operations

Other N-Array Operations

- `FILTER`
- `CONCATENATION`
- `UNION ALL/UNION`
- `INTERSECT`
- `MINUS`

`FILTER` Operations

- Accepts a set of rows
- Eliminates some of them
- Returns the rest



1

```
SELECT  deptno, sum(sal) SUM_SAL FROM emp
GROUP BY deptno HAVING sum(sal) > 9000;
```

Explain Plan

SQL | 0 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			3	1
FILTER				
Filter Predicates				
SUM(SAL)>9000				
HASH		GROUP BY	3	1
TABLE ACCESS	EMP	FULL	3	14

2

```
SELECT deptno, dname FROM dept d WHERE NOT EXISTS
(select 1 from emp e where e.deptno=d.deptno);
```

Explain Plan

SQL | 0 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			5	1
MERGE JOIN		ANTI	5	1
TABLE ACCESS	DEPT	BY INDEX ROWID	2	4
INDEX	PK_DEPT	FULL SCAN	1	4
SORT		UNIQUE	3	14
Access Predicates				
E.DEPTNO=D.DEPTNO				
Filter Predicates				
E.DEPTNO=D.DEPTNO				
TABLE ACCESS	EMP	FULL	3	14

ORACLE

A `FILTER` operation is any operation that discards rows returned by another step, but is not involved in retrieving the rows itself. All sorts of operations can be filters, including subqueries and single table predicates.

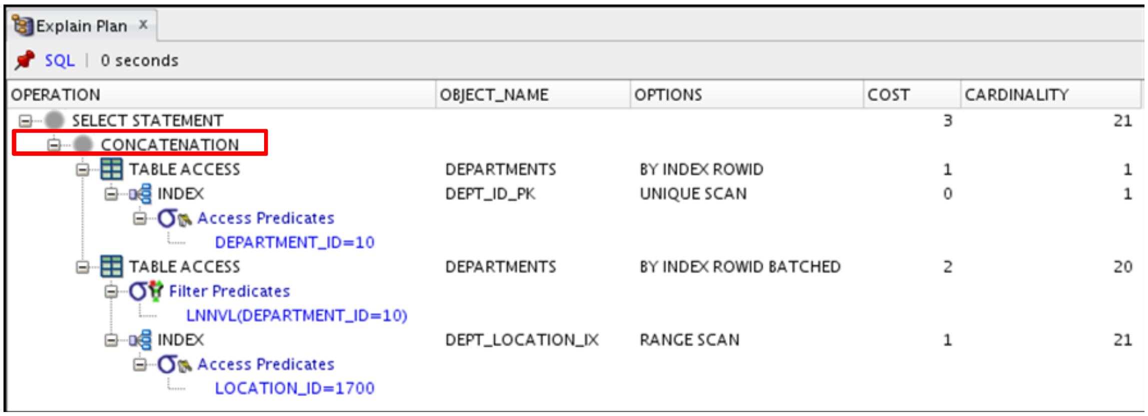
In Example 1 in the slide, `FILTER` applies to the groups that are created by the `GROUP BY` operation.

In Example 2 in the slide, `FILTER` is used almost in the same way as `NESTED LOOPS`. `DEPT` is accessed once, and for each row from `DEPT`, `EMP` is accessed by its index on `DEPTNO`. This operation is done as many times as the number of rows in `DEPT`.

The `FILTER` operation is applied, for each row, after the `DEPT` rows are fetched. `FILTER` discards rows for the inner query `(select 1 from emp e where e.deptno=d.deptno)` and returns at least one row that is `TRUE`.

Concatenation Operation

```
SELECT * FROM departments WHERE department_id=10
or location_id=1700;
```



Explain Plan

SQL | 0 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			3	21
CONCATENATION				
TABLE ACCESS	DEPARTMENTS	BY INDEX ROWID	1	1
INDEX	DEPT_ID_PK	UNIQUE SCAN	0	1
Access Predicates				
DEPARTMENT_ID=10				
TABLE ACCESS	DEPARTMENTS	BY INDEX ROWID BATCHED	2	20
Filter Predicates				
LNNVL(DEPARTMENT_ID=10)				
INDEX	DEPT_LOCATION_IX	RANGE SCAN	1	21
Access Predicates				
LOCATION_ID=1700				

`CONCATENATION` concatenates the rows returned by two or more row sets. This works like `UNION ALL` and does not remove duplicate rows.

It is used with `OR` expansions. However, `OR` does not return duplicate rows, so for each component after the first, it appends a negation of the previous components (`LNNVL`):

```
CONCATENATION
 - BRANCH 1 - SAL=2
 - BRANCH 2 - DEPTNO = 1 AND NOT row in Branch 1
```

The `LNNVL` function is generated by the `OR` clause to process this negation.

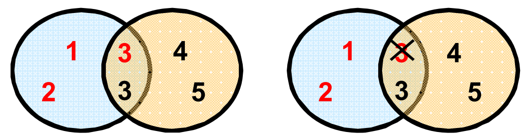
The `LNNVL()` function returns `TRUE` if the predicate is `NULL` or `FALSE`.

So `filter` (`LNNVL(SAL=2)`) returns all rows for which `SAL != 2` or `SAL is NULL`.

UNION [ALL], INTERSECT, MINUS

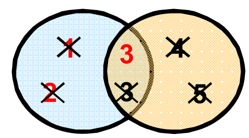
UNION
UNION ALL

```
SORT UNIQUE
UNION-ALL
INDEX FULL SCAN
INDEX FAST FULL SCAN
```



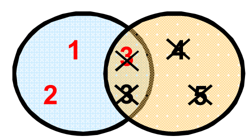
INTERSECT

```
INTERSECTION
SORT UNIQUE NOSORT
INDEX FULL SCAN
SORT UNIQUE
INDEX FAST FULL SCAN
```



MINUS

```
MINUS
SORT UNIQUE NOSORT
INDEX FULL SCAN
SORT UNIQUE
INDEX FAST FULL SCAN
```



ORACLE

SQL handles duplicate rows with an `ALL` or `DISTINCT` modifier in different places in the language. `ALL` preserves duplicates and `DISTINCT` removes them. Here is a quick description of the possible SQL set operations:

- **`INTERSECTION`:** Operation accepting two sets of rows and returning the intersection of the sets, eliminating duplicates. Subrow sources are executed or optimized individually. This is very similar to sort-merge-join processing: Full rows are sorted and matched.
- **`MINUS`:** Operation accepting two sets of rows and returning rows appearing in the first set, but not in the second, eliminating duplicates. Subrow sources are executed or optimized individually. This is similar to `INTERSECT` processing. However, instead of match-and-return, it is match-and-exclude.
- **`UNION`:** Operation accepting two sets of rows and returning the union of the sets, eliminating duplicates. Subrow sources are executed or optimized individually. Rows retrieved are concatenated and sorted to eliminate duplicate rows.
- **`UNION ALL`:** Operation accepting two sets of rows and returning the union of the sets, and not eliminating duplicates. The expensive sort operation is not necessary. Use `UNION ALL` if you know you do not have to deal with duplicates.

Quiz

Hash clusters are a better choice than indexed tables or index clusters when a table is queried frequently with equality queries.

a. True
b. False

Answer: a

Quiz

The ________ operator uses a temporary table to store intermediate data.

a. Buffer Sort operator
b. Inlist
c. Min/Max
d. N-Array

Answer: a

Quiz

The following query uses the ________ operator:

```
SELECT * FROM emp WHERE empno IN (7876, 7900,
7902);
```

a. Buffer Sort operator
b. Inlist
c. Min/Max
d. N-Array

Answer: b

Quiz

A `FILTER` operation retrieves the rows returned by another statement.

a. True

b. False

Answer: b

Summary

In this lesson, you should have learned to:

- Describe SQL operators for:
  - Clusters
  - In-List
  - Sorts
  - Filters
  - Set Operations
- Describe result cache operators

Practice 10: Overview

This practice covers the following topics:

- Using the result cache
- Using different access paths for better optimization (case 13 to case 15)(Optional)

11

Introduction to Optimizer Statistics Concepts

Objectives

After completing this lesson, you should be able to:

- Describe optimizer statistics
  - Table statistics
  - Index statistics
  - Column statistics (histogram)
  - Column statistics (extended statistics)
  - Session-specific statistics for global temporary tables
  - System statistics
- Gather and manage optimizer statistics

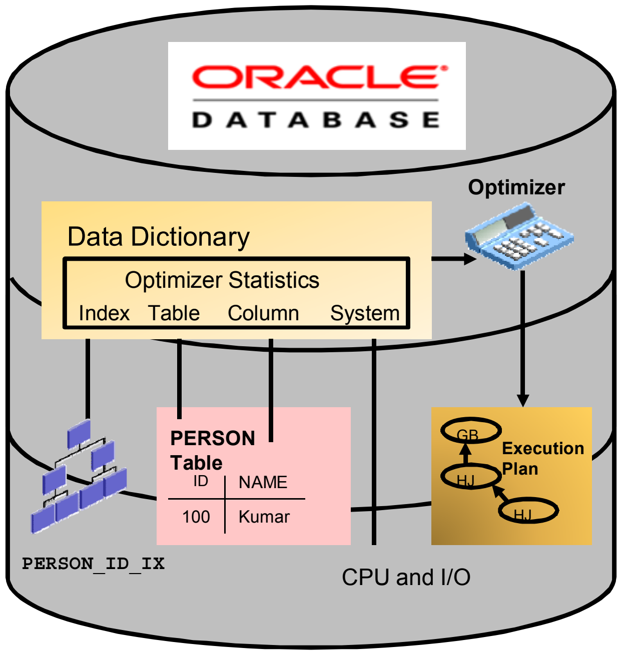
This lesson explains why statistics are important for the query optimizer and how to gather and use optimizer statistics.

Lesson Agenda

- Optimizer Statistics
  - Table Statistics
  - Index Statistics
    - Index Clustering Factor
  - Column Statistics
- Column Statistics: Histograms
- Column Statistics: Extended Statistics
- Session-Specific Statistics for Global Temporary Tables
- System Statistics
- Gathering and Managing Optimizer Statistics

Optimizer Statistics

Describe the database and the objects in the database



Optimizer statistics provide details about a 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. The information used by the query optimizer to estimate is as follows:

- Selectivity of predicates
- Cost of each execution plan
- Access method, join order, and join method
- CPU and I/O costs

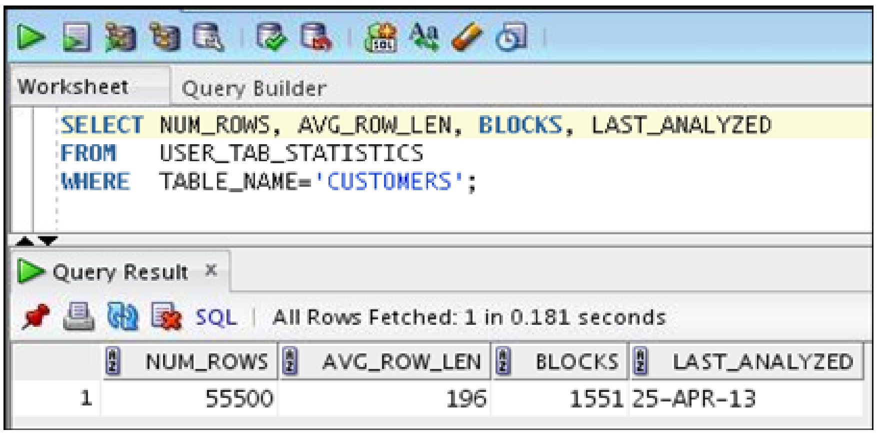
Because the objects in a database change constantly, statistics must be regularly updated so that they accurately describe these database objects. Statistics are maintained automatically by Oracle Database, or you can maintain the optimizer statistics manually by using the `DBMS_STATS` package.

- Table statistics
- Index statistics
- Column statistics
- System statistics

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 that are visible through `V$` views.

Table Statistics (`USER_TAB_STATISTICS`)

- Used to determine:
  - Table access cost
  - Join cardinality
  - Join order
- Example:



In Oracle Database, table statistics include information about rows and blocks. The optimizer uses these statistics to determine the cost of table scans and table joins. `DBMS_STATS` can gather statistics for both permanent and temporary tables. Some of the table statistics gathered are:

`NUM_ROWS`

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

`BLOCKS`

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

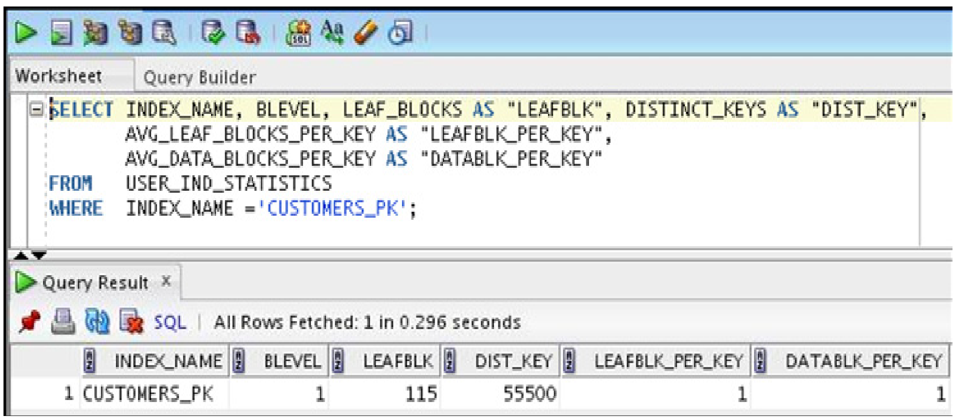
`STALE_STATS`

This statistic 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 are not gathered by the `DBMS_STATS` procedures. If they are required, the `ANALYZE` command must be used.

Index Statistics (`USER_IND_STATISTICS`)

- The index statistics include information about the number of index levels, the number of index blocks, and the relationship between the index and the data blocks.
- They are used to decide between:
  - Full table scan and index scan



Worksheet | Query Builder

```
SELECT INDEX_NAME, BLEVEL, LEAF_BLOCKS AS "LEAFBLK", DISTINCT_KEYS AS "DIST_KEY",
       AVG_LEAF_BLOCKS_PER_KEY AS "LEAFBLK_PER_KEY",
       AVG_DATA_BLOCKS_PER_KEY AS "DATABLK_PER_KEY"
FROM   USER_IND_STATISTICS
WHERE  INDEX_NAME ='CUSTOMERS_PK';
```

Query Result × | SQL | All Rows Fetched: 1 in 0.296 seconds

	INDEX_NAME	BLEVEL	LEAFBLK	DIST_KEY	LEAFBLK_PER_KEY	DATABLK_PER_KEY
1	CUSTOMERS_PK	1	115	55500	1	1

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. Index statistics stored in the `USER_IND_STATISTICS` view track the following:

`BLEVEL`

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

`CLUSTERING_FACTOR`

This statistic 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.

DISTINCT_KEYS

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

AVG_LEAF_BLOCKS_PER_KEY

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

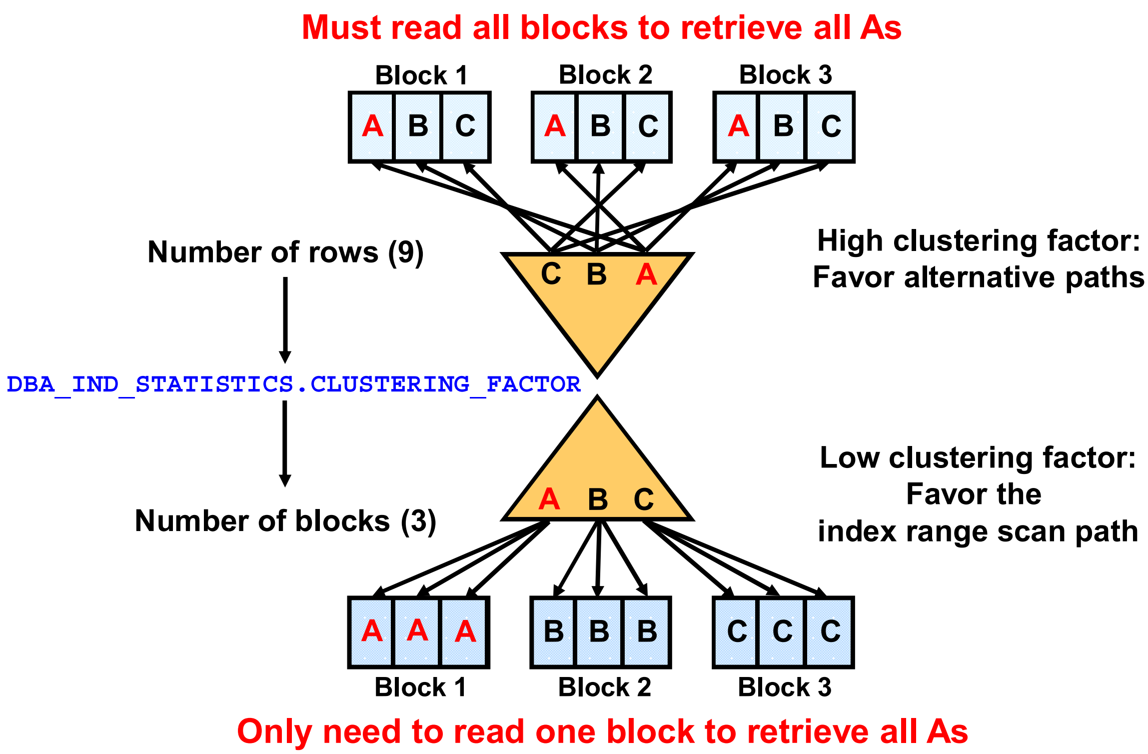
AVG_DATA_BLOCKS_PER_KEY

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

NUM_ROWS

This statistic is the number of rows in the index.

Index Clustering Factor



The system performs 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 selected index entry points to a block in the table. If each entry points to a different block, the accessed rows and accessed blocks 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 the table's selectivity.

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

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

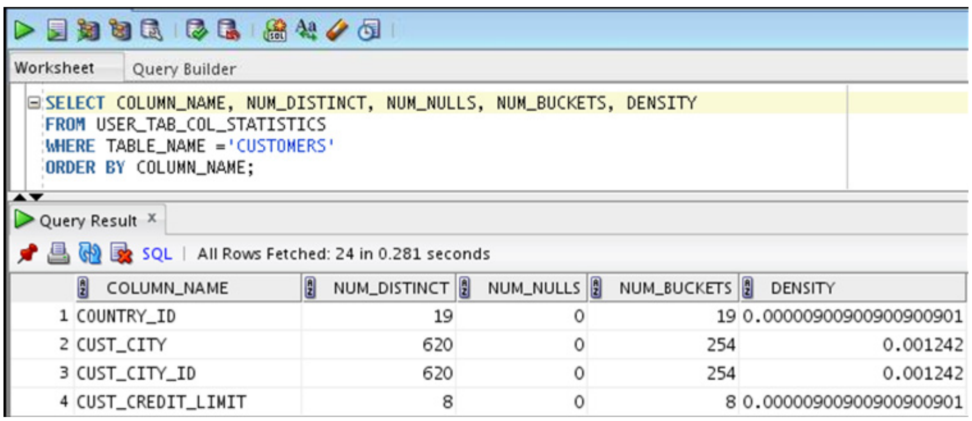
The example in the slide shows how the clustering factor can affect cost. Assume the following situation: There is a table with nine rows; there is a non-unique index on col1 for the table; the `c1` column currently stores the values A, B, and C; the table has only three data blocks.

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

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

Column Statistics (`USER_TAB_COL_STATISTICS`)

- Column statistics track information about column values and data distribution.
- The optimizer uses these statistics to generate accurate cardinality estimates and make better decisions about index usage, join orders, join methods, and so on.




```sql
SELECT COLUMN_NAME, NUM_DISTINCT, NUM_NULLS, NUM_BUCKETS, DENSITY
FROM USER_TAB_COL_STATISTICS
WHERE TABLE_NAME ='CUSTOMERS'
ORDER BY COLUMN_NAME;
```

	COLUMN_NAME	NUM_DISTINCT	NUM_NULLS	NUM_BUCKETS	DENSITY
1	COUNTRY_ID	19	0	19	0.0000090090090090090901
2	CUST_CITY	620	0	254	0.001242
3	CUST_CITY_ID	620	0	254	0.001242
4	CUST_CREDIT_LIMIT	8	0	8	0.0000090090090090090901

Column statistics track information about column values and data distribution.

The optimizer uses these statistics to generate accurate cardinality estimates and make better decisions about index usage, join orders, join methods, and so on.

Index statistics in `USER_TAB_COL_STATISTICS` track the following:

- `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 relevant only 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 the histogram for the column.
- `HISTOGRAM` indicates the existence or type of the histogram: `NONE`, `FREQUENCY`, `HEIGHT` and `BALANCED`.

Lesson Agenda

- Optimizer Statistics
  - Table Statistics
  - Index Statistics
    - Index Clustering Factor
  - Column Statistics
- **Column Statistics: Histograms**
- Column Statistics: Extended Statistics
- Session-Specific Statistics for Global Temporary Tables
- System Statistics
- Gathering and Managing Optimizer Statistics

Column Statistics: Histograms

- The optimizer assumes uniform distributions; this may lead to suboptimal access plans in the case of data skew.
- Histograms:
  - Store additional column distribution information
  - Give better selectivity estimates in the case of non-uniform distributions
- Types of histograms:
  - Frequency
  - Top-Frequency
  - Height-Balanced (legacy)
  - Hybrid
- They are stored in `DBA_TAB_HISTOGRAMS`.

ORACLE

A histogram is a special type of column statistic that provides more detailed information about the data distribution in a table column. A histogram sorts values into “buckets,” as you might sort coins into buckets. Because a histogram captures the distribution of different values in a column, it yields better selectivity estimates. Having histograms on columns that contain skewed data or values with large variations in the number of duplicates helps the query optimizer generate good selectivity estimates and make better decisions about index usage, join orders, and join methods.

Without histograms, uniform distribution is assumed. If a histogram is available on a column, the estimator uses it instead of the number of distinct values. Creation of histograms is controlled by the `METHOD_OPT` parameter.

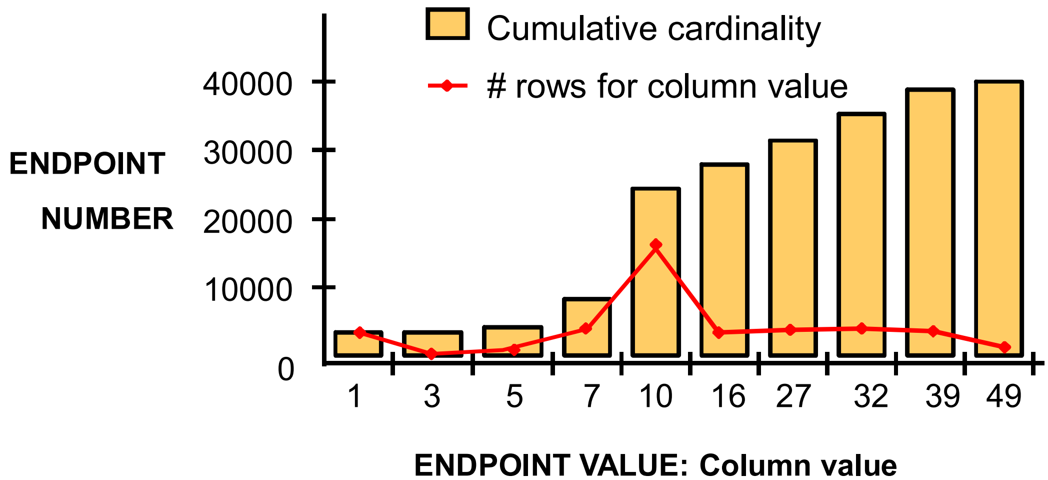
When creating histograms, Oracle Database uses different types of histogram representations, depending on the number of distinct values found in the corresponding column. When you have a data set with less than 254 distinct values, and the number of histogram buckets is not specified, the system creates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

10 buckets, 10 distinct values



Distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, 49

Number of rows: 40001

For the example in the slide, assume that you have a column that is populated with 40,001 numbers. You have only 10 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 the 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

```
SELECT country_subregion_id, count(*)
FROM sh.countries GROUP BY country_subregion_id
ORDER BY 1;
```

```
BEGIN DBMS_STATS.GATHER_TABLE_STATS ( ownname => 'SH' , tabname =>
'COUNTRIES' , method_opt => 'FOR COLUMNS COUNTRY_SUBREGION_ID' );
END;
```

```
SELECT TABLE_NAME, COLUMN_NAME, NUM_DISTINCT, NUM_BUCKETS, HISTOGRAM
FROM USER_TAB_COL_STATISTICS WHERE TABLE_NAME='COUNTRIES'
AND COLUMN_NAME='COUNTRY_SUBREGION_ID';

 TABLE_NAME  COLUMN_NAME          NUM_DISTINCT   NUM_BUCKETS HISTOGRAM
------------ ------------         -----------      --------- ---------
COUNTRIES    COUNTRY_SUBREGION_ID          8               8     FREQUENCY
```

```
SELECT ENDPOINT_NUMBER, ENDPOINT_VALUE
FROM USER_HISTOGRAMS
WHERE TABLE_NAME='COUNTRIES'
AND COLUMN_NAME='COUNTRY_SUBREGION_ID';

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
1                        52792
6                        52793
8                        52794
…
```


The example in the slide shows you how to view a frequency histogram. Here, you want to generate a frequency histogram on the `sh.countries.country_subregion_id` column. This table has 23 rows.

The first query shows that the `country_subregion_id` column contains eight distinct values that are unevenly distributed.

Because the number of distinct values in the `country_subregion_id` column of the `COUNTRIES` table is eight, and the number of requested buckets defaults to 254, the system automatically creates a frequency histogram with eight 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 the `ENDPOINT_NUMBER` column that corresponds to the cumulative frequency of the corresponding `ENDPOINT_VALUE` column, which represents, in this case, the actual value of the column data.

For 52793, the endpoint number 6 indicates that the value appears 5 times (6 - 1).

For 52794, the endpoint number 8 indicates that the value appears 2 times (8 - 6).

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

Top Frequency Histogram

- Traditionally a frequency histogram is created only if NDV < 254.
- If a column contains more than 254 distinct values, and if a small number of distinct values occupy more than 99% of the data, the database creates a Top Frequency histogram by using the small number of extremely popular distinct values.
- Creating a frequency histogram on that small set of values is very useful even though NDV is greater than 254.
- The unpopular values are ignored to create a better quality histogram for popular values.
- This is created only with `AUTO_SAMPLE_SIZE`.

Before Oracle Database 12*c*, a frequency histogram was created only if the number of distinct values (NDV) was less than 254. Starting with Oracle Database 12*c*, if a column contains more than 254 distinct values, and if a small number of distinct values occupy more than 99 percent of the data, the database creates a Top Frequency histogram by using the small number of extremely popular distinct values. It is built using the same technique that is used for frequency histograms.

A Top Frequency histogram can produce a better histogram for highly popular values by ignoring the statistically insignificant unpopular values.

A Top Frequency histogram is indicated by the `TOP-FREQUENCY` value in the `USER_TAB_COL_STATISTICS.HISTOGRAM` column.

In Oracle Database 12*c*, the database creates Top Frequency histograms when the following conditions are true:

- The sampling percentage is `AUTO`. In this release, the database constructs frequency histograms from a full table scan when the sampling size is `AUTO` (which is the default). For all other sampling percentage specifications, the database derives frequency histograms from a sample.
- The number of distinct values in the data set is *n*.
- The percentage of rows occupied by the top *n* frequent values is equal to or greater than threshold *p*, where *p* is (1-(1/n))*100.

Viewing Top Frequency Histograms

```
SELECT country_subregion_id, count(*)
FROM sh.countries GROUP BY country_subregion_id
ORDER BY 1;
```

```
BEGIN DBMS_STATS.GATHER_TABLE_STATS ( ownname => 'SH' , tabname =>
'COUNTRIES' , method_opt => 'FOR COLUMNS COUNTRY_SUBREGION_ID SIZE 7' );
END;
```

```
SELECT TABLE_NAME, COLUMN_NAME, NUM_DISTINCT, NUM_BUCKETS, HISTOGRAM
FROM USER_TAB_COL_STATISTICS WHERE TABLE_NAME='COUNTRIES'
AND COLUMN_NAME='COUNTRY_SUBREGION_ID';

TABLE_NAME  COLUMN_NAME            NUM_DISTINCT  NUM_BUCKETS  HISTOGRAM
---------- ---------------         -----------   ----------   ---------
COUNTRIES    COUNTRY_SUBREGION_ID          8            7    TOP-FREQUENCY
```

```
SELECT ENDPOINT_NUMBER, ENDPOINT_VALUE
FROM USER_HISTOGRAMS
WHERE TABLE_NAME='COUNTRIES'
AND COLUMN_NAME='COUNTRY_SUBREGION_ID';

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
1                        52792
6                        52793
8                        52794
…
```

ORACLE

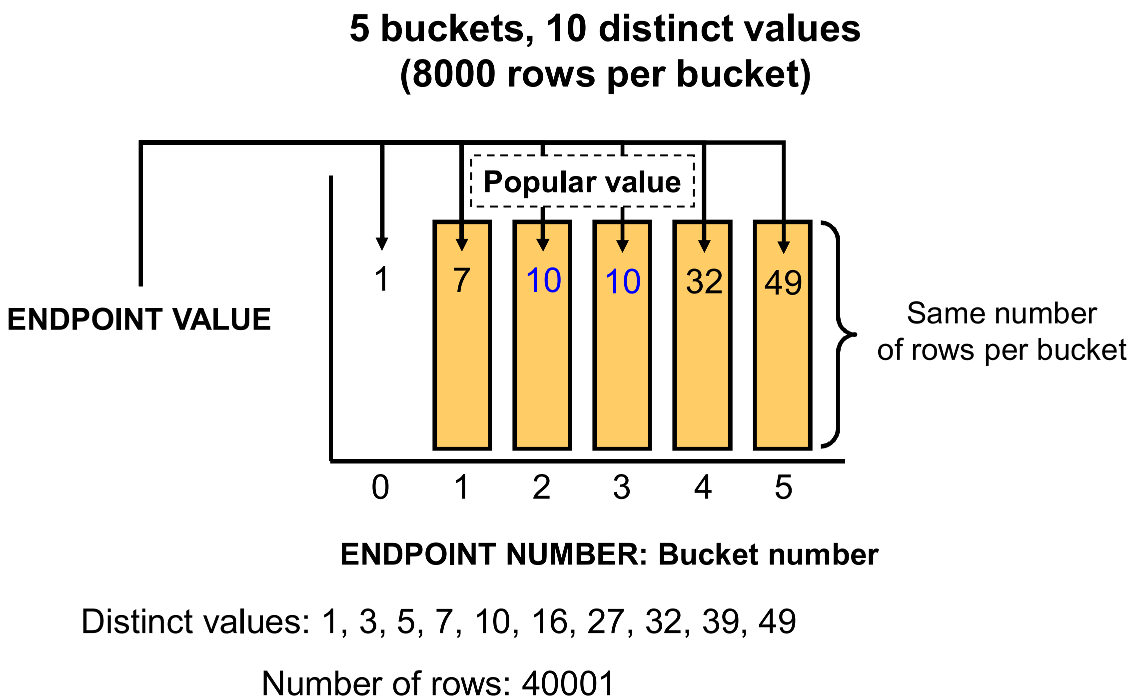
The example in the slide shows you how to view a Top Frequency histogram. Here, you want to generate a frequency histogram on the `sh.countries.country_subregion_id` column. This table has 23 rows.

The first query shows that the `country_subregion_id` column contains eight distinct values that are unevenly distributed.

Because the number of distinct values in the `country_subregion_id` column of the `COUNTRIES` table is eight and the number of requested buckets seven, the top seven most frequent values occupy 95.6% of the rows, which exceeds the threshold of 85.7%, thus generating a Top Frequency histogram.

When you query the endpoint number and endpoint value for the column, you see that each distinct value has its own bucket, except 52795, which is excluded from the histogram because it is nonpopular and statistically insignificant.

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. For 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 have only 10 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 8,000. Based on this assumption, value 10 appears between 8,000 and 24,000 times. So you are sure that value 10 is a popular value.

Before Oracle Database 12*c* Release 1 (12.1), the database created a height-balanced histogram when the NDV was greater than *n*.

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

The number of rows per bucket is not recorded because it 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. For 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

```
BEGIN
  DBMS_STATS.gather_table_STATS(OWNNAME =>'OE', TABNAME=>'INVENTORIES',
  METHOD_OPT => 'FOR COLUMNS SIZE 10 QUANTITY_ON_HAND',
ESTIMATE_PERCENT=>100);
END;
```

```
SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';

COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
QUANTITY_ON_HAND                        237          10 HEIGHT BALANCED
```

```
SELECT endpoint_number, endpoint_value
FROM USER_HISTOGRAMS
WHERE table_name = 'INVENTORIES' and column_name = 'QUANTITY_ON_HAND'
ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              0
              1             27
              2             42
              3             57
…
```

ORACLE

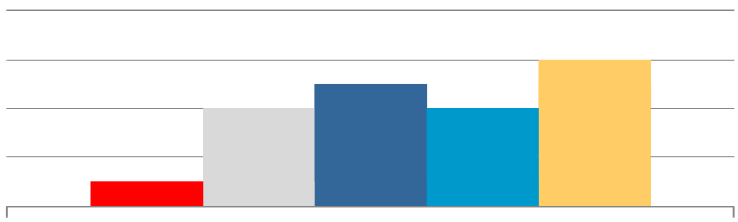
The example in the slide shows you how to view a height-balanced histogram. Because the number of distinct values in the `QUANTITY_ON_HAND` column of the `INVENTORIES` table is 237; the number of requested buckets is 10; and the `estimate_percent` is set to a nondefault value, the system creates a height-balanced histogram with 10 buckets.

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

Note: To simulate Oracle Database 11*g* behavior, which is necessary to create height-based histograms, set `estimate_percent` to a nondefault value. If you specify a nondefault percentage, the database creates frequency or height-balanced histograms.

Hybrid Histograms

- These are similar to height-balanced histogram and are created if the NDV >254.
- They store the actual frequencies of bucket endpoints in histograms.
- No values are allowed to spill over multiple buckets.
- More endpoint values can be squeezed in a histogram.
- The same effect is achieved as increasing the # of buckets.
- They are created only with `AUTO_SAMPLE_SIZE`.



A hybrid histogram combines the characteristics of both height-based histograms and frequency histograms. It is created if the number of distinct values in a column (NDV) is greater than 254 values.

The height-based histogram sometimes produces inaccurate estimates for values that are almost popular. For example, a value that occurs as an endpoint value of only one bucket but almost occupies two buckets is not considered as popular.

To solve this problem, a hybrid histogram stores the endpoint repeat count value, which is the number of times the endpoint value is repeated, for each endpoint in the histogram. By using the repeat count, the optimizer can obtain accurate estimates for almost popular values.

In Oracle Database 12*c*, the database creates hybrid histograms when the following conditions are true:

- The sampling percentage is `AUTO`.
  - If you specify your own percentage, the database creates frequency or height-balanced histograms.
- The criteria for Top Frequency histograms do not apply.
- *n* is less than `NDV`, where *n* is the user-specified number of buckets. If no number is specified, *n* defaults to 254.

Viewing Hybrid Histograms

```
BEGIN
  DBMS_STATS.gather_table_STATS(OWNNAME =>'OE', TABNAME=>'INVENTORIES',
  METHOD_OPT => 'FOR COLUMNS SIZE 10 QUANTITY_ON_HAND');
END;
```

```
SELECT column_name, num_distinct, num_buckets, histogram
FROM USER_TAB_COL_STATISTICS
WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';

COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
QUANTITY_ON_HAND                        237          10 HYBRID
```

```
SELECT endpoint_number, endpoint_value
FROM USER_HISTOGRAMS
WHERE table_name = 'INVENTORIES' and column_name = 'QUANTITY_ON_HAND'
ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              0
              1             27
              2             42
              3             57
…
```


The example in the slide shows you how to view a hybrid 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, which is less than 237, the optimizer cannot create a frequency histogram. The optimizer considers both hybrid and top frequency histograms.

To qualify for a top frequency histogram, the percentage of rows occupied by the top 10 most frequent values must be equal to or greater than threshold *p*, where *p* is (1-(1/10))*100, or 90%. However, in this case, the top 10 most frequent values is less than the threshold *p.* Therefore, the optimizer chooses a hybrid histogram because the criteria for a top frequency histogram do not apply.

Note: If no sampling percentage is specified, Oracle Database 12*c* Release 1 (12.1) no longer creates height-balanced histograms. If you upgrade the database from Oracle Database 11*g* to Oracle Database 12*c*, any height-based histograms created before the upgrade remain in use. If Oracle Database 12*c* Release 1 (12.1) creates new histograms, and if the sampling percentage is `AUTO_SAMPLE_SIZE`, the histograms are either top frequency or hybrid, but not height-balanced.

Best Practices: Histogram

- Histograms are useful when you have a high degree of skew in column distribution.
- Histograms are *not* useful for:
  - Columns that do not appear in the `WHERE` or `JOIN` clause
  - Columns with uniform distributions
  - Equality predicates with unique columns
- The maximum number of buckets is the least (254, # distinct values). If possible, frequency histograms are preferred.
- Do not use histograms unless they substantially improve performance.

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

In addition, for columns that are 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. If possible, frequency histograms are preferred. 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 because histogram data is stored only for the first 32 bytes of any string.

Best Practices: Histogram

- Set `METHOD_OPT` to `FOR ALL COLUMNS AUTO`.
- Use `TRUNCATE` instead of dropping and re-creating the same table if you need to remove all the rows from a table.
- If incorrect cardinality or selectivity is observed in an execution plan, check to see if a histogram can resolve the problem.
- Make sure that statistics for objects are collected at the highest sample size you can afford and see if the plan improves.
- In earlier releases, if a query uses binds or binds are not representative of future executions, you should not consider histograms to avoid bind peeking. Starting from Oracle Database 11*g*, adaptive cursor sharing resolves bind/histogram issues.

When gathering statistics, histograms are specified by using the `METHOD_OPT` argument of the `DBMS_STATS` gathering procedures. Oracle recommends setting `METHOD_OPT` to `FOR ALL COLUMNS SIZE AUTO`. With this setting, Oracle Database automatically determines which columns require histograms and the number of buckets (size) for each histogram. Column usage history is collected and monitored at `sys.col_usage$`. This information is needed by `dbms_stats` to identify candidate columns on which to build histograms when `METHOD_OPT` is set to `FOR ALL COLUMNS SIZE AUTO`.

Note: When upgrading, in some cases, the effect of a histogram is adverse to the generation of a better plan (especially in the presence of bind variables combined with small or `AUTO` sample sizes). Again, you may want to initially set this parameter to its release value before the upgrade, and later adjust to your release default value after the upgrade.

If you need to remove all the rows from a table when using `DBMS_STATS`, use `TRUNCATE` instead of dropping and re-creating the same table. When you drop a table, workload information used by the auto-histogram gathering feature and saved statistics history used by the `RESTORE_*_STATS` procedures are lost. Without this data, these features do not function properly.

Note: Statistics gathering is covered in Appendix F titled “Gathering and Managing Optimizer Statistics.”

If incorrect cardinality or selectivity is observed in an execution plan, check to see if a histogram can resolve the problem.

Make sure that statistics for objects are collected at the highest sample size you can afford and see if the plan improves. The default sample size is 100 percent.

Lesson Agenda

- Optimizer Statistics
  - Table Statistics
  - Index Statistics
    - Index Clustering Factor
  - Column Statistics
- Column Statistics: Histograms
- Column Statistics: Extended Statistics
- Session-Specific Statistics for Global Temporary Tables
- System Statistics
- Gathering and Managing Optimizer Statistics

Column Statistics: Extended Statistics

- The optimizer poorly estimates selectivity on *Highly Correlated Column Predicates:*
  - Columns have values that are highly correlated.
  - Actual selectivity is often much lower or higher than the optimizer estimates. For example,
    ```
    WHERE cust_state_province = 'CA'
    AND country_id=52775;
    ```
- The optimizer poorly estimates *Expression on Columns:*
  - `WHERE upper(model)='MODEL'`
  - When a function is applied to a column in the `WHERE` clause, the optimizer has no way of knowing how that function affects the selectivity of the column.

Extended statistics enable the gathering of statistics on a group of columns within a table as a whole, providing the optimizer with more information about any correlation that may exist between the columns. `DBMS_STATS` enables you to collect extended statistics, which are statistics that can improve cardinality estimates when multiple predicates exist on different columns of a table.

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, the optimizer uses the number of distinct keys (NDK) in the index for estimating selectivity, as 1/NDK.
- When `DYNAMIC_STATISTICS` is set to level `4`, the query optimizer uses dynamic statistics to estimate the selectivity of complex predicates that involve 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.

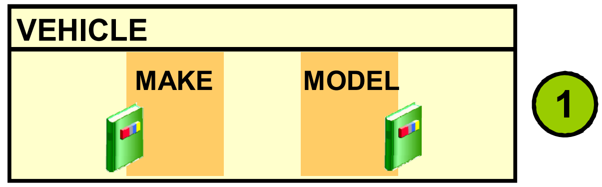
In all other cases, the optimizer assumes that the values of columns used in a complex predicate are independent of each other. It estimates the selectivity of a conjunctive predicate by multiplying the selectivity of individual predicates. This approach results in underestimation of the selectivity if there is a correlation between the columns.

To circumvent this issue, Oracle Database 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.

When a function is applied to a column in the `WHERE` clause of a query (`function(col1)=constant`), the optimizer has no way of knowing how that function affects the selectivity of the column. 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. By gathering expression statistics on the expression `function(col1)`, the optimizer obtains a more accurate selectivity value.

Oracle Database gathers statistics on a group of columns within a table or an expression on a column to obtain a more accurate selectivity value.

Column Group Statistics



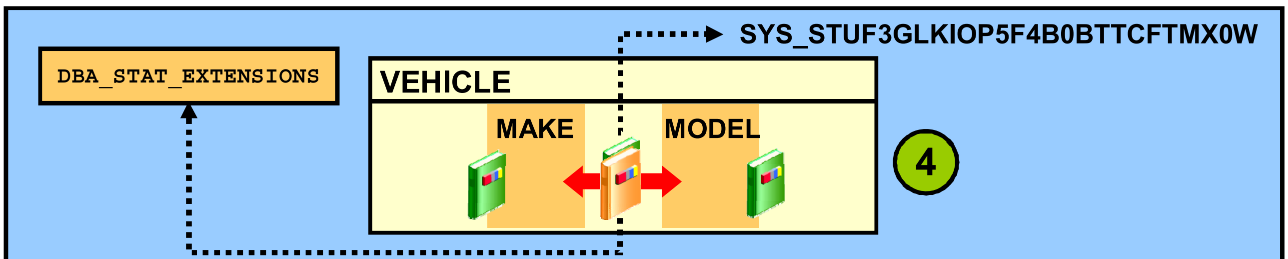
S(MAKE Λ MODEL)=S(MAKE)xS(MODEL)

2

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

3

```
exec dbms_stats.gather_table_stats('jfv','vehicle',-
method_opt=>'for all columns size 1 for columns (make,model) size 3');
```



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

Starting in this release, Oracle Database automatically determines the column groups that are required in a given workload or SQL tuning set (STS), and then creates the column groups. Thus, for any given workload, you no longer need to know which columns from each table will be used together as a workload. By monitoring a workload, Oracle Database 12*c* records the necessary column groups. You can then create the column groups by executing `DBMS_STATS.CREATE_EXTENDED_STATS`.

You can use `DBMS_STATS.SEED_COL_USAGE` and `REPORT_COL_USAGE` to determine the column groups that are required for a table based on a specified workload. This technique is useful when you do not know what extended statistics to create. This technique does not work for expression statistics.

The optimizer uses column group statistics for equality predicates, inlist predicates, and for estimating the group by cardinality.

Note

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

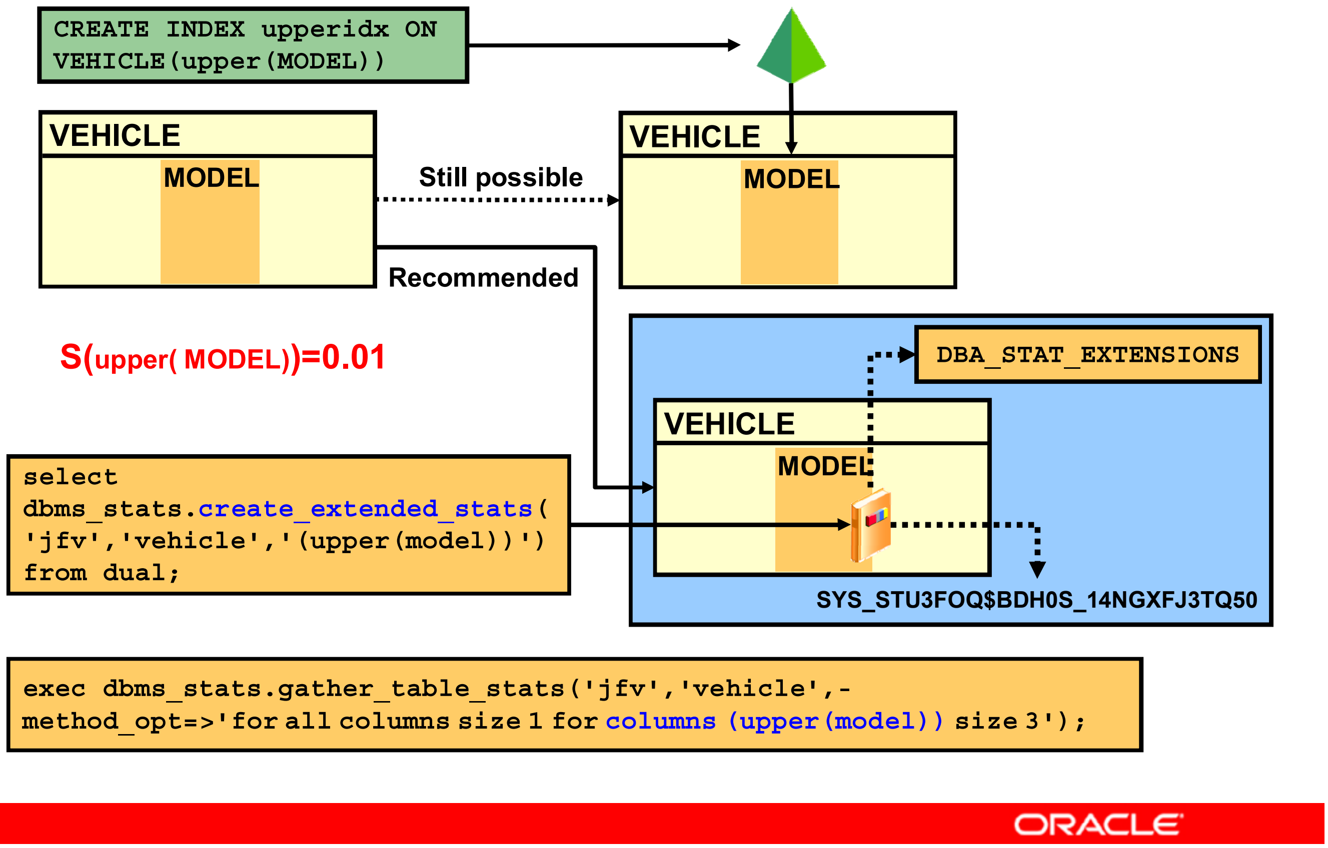
```
select
dbms_stats.show_extended_stats_name('jfv','vehicle','(make,model)')
from dual
```

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

You can use the `DBMS_STATS.DROP_EXTENDED_STATS` function to delete a column group from a table.

You learn about detecting useful column groups for a specific workload, creating column groups manually, and gathering column group statistics in the practices.

Expression Statistics



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 a column and thus allow the optimizer to use 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 by using dynamic statistics. Finally, the optimizer collects statistics on the virtual columns that are created to support function-based indexes.

However, these solutions are either limited to a certain class of functions or they work only for the expressions that are used to create function-based indexes. With expression statistics in Oracle Database, you can use a more general solution that includes arbitrary user-defined functions and does not depend on the presence of function-based indexes.

Expression statistics improve optimizer estimates when a `WHERE` clause has predicates that use expressions. As shown in the example in the slide, this feature relies on the virtual column infrastructure to create statistics on expressions of columns.

You can use DBMS_STATS to create statistics for a user-specified expression. You have the option of using either of the following program units:

- GATHER_TABLE_STATS procedure
- CREATE_EXTENDED_STATISTICS function followed by the GATHER_TABLE_STATS procedure

You can use the database view DBA_STAT_EXTENSIONS and the DBMS_STATS.SHOW_EXTENDED_STATS_NAME function to obtain information about expression statistics. You can also use views to obtain information such as the number of distinct values and whether the column group has a histogram. Use the DBMS_STATS.DROP_EXTENDED_STATS function to delete a column group from a table.

Note

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

Lesson Agenda

- Optimizer Statistics
  - Table Statistics
  - Index Statistics
    - Index Clustering Factor
  - Column Statistics
- Column Statistics: Histograms
- Column Statistics: Extended Statistics
- Session-Specific Statistics for Global Temporary Tables
- System Statistics
- Gathering and Managing Optimizer Statistics

Session-Specific Statistics for Global Temporary Tables

- Traditionally, statistics gathered on global temporary tables (GTT) were shared by all sessions, even though data in different sessions could differ.
- Now each session can have its own version of statistics for GTT.
- This is controlled by the new preference `GLOBAL_TEMP_TABLE_STATS`.
- The default value is `SESSION` (non-shared).
- To force sharing, set table preference to `SHARED`.

A global temporary table is a special table that stores intermediate session-private data for a specific duration. Before Oracle Database 12*c*, it was difficult to gather statistics on global temporary tables because only one set of statistics was available for all occurrences of global temporary tables. When you create a global temporary table, you create a definition that is visible to all sessions. No physical storage is allocated. When a session first puts data into the table, the database allocates storage space. The data in a temporary table is visible only to the current session.

In previous releases, the database did not maintain statistics for global temporary tables and non-global temporary tables differently. The database maintained one version of the statistics that was shared by all sessions, even though data in different sessions could differ.

Starting in Oracle Database 12*c*, you can maintain session-private statistics. You can choose to gather statistics for a global temporary table in one session and use the statistics only for this session. You can set the table-level preference `GLOBAL_TEMP_TABLE_STATS` to make statistics on a global temporary table shared or session-specific. Meanwhile, you can continue to maintain a shared version of the statistics that is usable by all other sessions, which have not yet gathered session-private statistics. Session level DML monitoring is also provided to the global temporary tables. Session-specific statistics for global temporary tables allow queries to be optimized more accurately based on your own data in a global temporary table.

Session-specific statistics have the following characteristics:

- Dictionary views that track statistics show both the shared statistics and the session-specific statistics in the current session.
- The views are DBA_TAB_STATISTICS, DBA_IND_STATISTICS, DBA_TAB_HISTOGRAMS, and DBA_TAB_COL_STATISTICS (each view has a corresponding USER_ and ALL_ version). The SCOPE column shows whether statistics are session-specific or shared.
- Other sessions do not share the cursor that is using the session-specific statistics.
- Different sessions can share the cursor that is using shared statistics, as in previous releases. The same session can share the cursor that is using session-specific statistics.
- Pending statistics are not supported for session-specific statistics.
- When the GLOBAL_TEMP_TABLE_STATS preference is set to SESSION, by default, GATHER_TABLE_STATS immediately invalidates the previous cursors that were compiled in the same session. However, this procedure does not invalidate cursors compiled in other sessions.

DBMS_STATS commits changes to session-specific global temporary tables, but not to transaction-specific global temporary tables. In previous releases, running DBMS_STATS.GATHER_TABLE_STATS on a transaction-specific temporary table (ON COMMIT DELETE ROWS) would delete all the rows in the table, making the statistics show the table as empty. Starting in Oracle Database 12*c* Release 1 (12.1), the following procedures do not commit for transaction-specific temporary tables, so that data in these tables is not lost:

- GATHER_TABLE_STATS
- DELETE_TABLE_STATS
- DELETE_COLUMN_STATS
- DELETE_INDEX_STATS
- SET_TABLE_STATS
- SET_COLUMN_STATS
- SET_INDEX_STATS
- GET_TABLE_STATS
- GET_COLUMN_STATS
- GET_INDEX_STATS

The preceding program units observe the GLOBAL_TEMP_TABLE_STATS preference.

Session-Specific Statistics for Global Temporary Tables: Example

- Statistics gathered on a GTT are no longer shared by all sessions.
- To restore shared statistics, change the table preference `GLOBAL_TEMP_TABLE_STATS` to `SHARED`.

1

```
SQL> CREATE GLOBAL TEMPORARY TABLE TG(col1 NUMBER);

Table created.

SQL> SELECT DBMS_STATS.GET_PREFS('GLOBAL_TEMP_TABLE_STATS','SH','TG')from dual;

DBMS_STATS.GET_PREFS('GLOBAL_TEMP_TABLE_STATS','SH','TG')
--------------------------------------------------------------------------------
SESSION
```

By default, statistics on GTT are session-private.

2

```
SQL> BEGIN
dbms_stats.set_table_prefs('SH','TG','GLOBAL_TEMP_TABLE_STATS','SHARED');
END;
/  2    3    4

PL/SQL procedure successfully completed.

SQL> SELECT DBMS_STATS.GET_PREFS('GLOBAL_TEMP_TABLE_STATS','SH','TG')from dual;

DBMS_STATS.GET_PREFS('GLOBAL_TEMP_TABLE_STATS','SH','TG')
--------------------------------------------------------------------------------
SHARED
```

ORACLE

Lesson Agenda

- Optimizer Statistics
  - Table Statistics
  - Index Statistics
    - Index Clustering Factor
  - Column Statistics
- Column Statistics: Histograms
- Column Statistics: Extended Statistics
- Session-Specific Statistics for Global Temporary Tables
- System Statistics
- Gathering and Managing Optimizer Statistics

System Statistics

- System statistics are used to estimate:
  - I/O performance and utilization
  - CPU performance and utilization
- System statistics enable the query optimizer to estimate I/O and CPU costs more accurately, enabling the query optimizer to choose a better execution plan.
- Procedures:
  - `DBMS_STATS.GATHER_SYSTEM_STATS`
  - `DBMS_STATS.SET_SYSTEM_STATS`
  - `DBMS_STATS.GET_SYSTEM_STATS`

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 system characteristics to select the most efficient plan with optimal proportion between I/O and CPU costs. System CPU and I/O characteristics depend on many factors and do not stay constant all the time.

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 by using `DBMS_STATS.SET_SYSTEM_STATS`. Use `DBMS_STATS.GET_SYSTEM_STATS` to verify system statistics.

System Statistics: Example

Viewing System Statistics:

```
SELECT * FROM sys.aux_stats$;

SNAME                PNAME              PVAL1      PVAL2
-------------------- ------------------ ---------- -------------------
SYSSTATS_INFO       STATUS                         COMPLETED
SYSSTATS_INFO       DSTART                         08-09-2001 16:40
SYSSTATS_INFO       DSTOP                          08-09-2001 16:42
SYSSTATS_INFO       FLAGS                        0
SYSSTATS_MAIN       SREADTIM                 7.581
SYSSTATS_MAIN       MREADTIM                56.842
SYSSTATS_MAIN       CPUSPEED                   117
SYSSTATS_MAIN       MBRC                         9
```


The slide shows an example of the optimizer system statistics.

- **sreadtim:** Single block read time is the average time to read a single block randomly.
- **mreadtim:** Multiblock read is the average time to read a multiblock sequentially.
- **cpuspeed:** This represents the workload CPU speed. CPU speed is the average number of CPU cycles in each second.
- **mbrc:** Multiblock count is the average multiblock read count sequentially.

For more information about these statistics, see the following link:

http://st-doc.us.oracle.com/12/121/server.121/e15858/tgsql_stats.htm#CHDFCHJF

(Table 13-7 Optimizer System Statistics in the `DBMS_STAT` Package)

Best Practices: System Statistics

- System statistics must be gathered on a regular basis; this does not invalidate cached plans.
- Gathering system statistics equals analyzing system activity for a specified period of time.
- When gathering optimizer system statistics:
  - It is highly recommended that you gather system statistics during normal workload for several hours
  - If no real workload is available, you can also gather `NORWORKLOAD` statistics

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 online analytical processing (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 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 by using `DBMS_STATS.SET_SYSTEM_STATS`. Use `DBMS_STATS.GET_SYSTEM_STATS` to verify system statistics.

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

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

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

Lesson Agenda

- Optimizer Statistics
  - Table Statistics
  - Index Statistics
    - Index Clustering Factor
  - Column Statistics
- Column Statistics: Histograms
- Column Statistics: Extended Statistics
- Session-Specific Statistics for Global Temporary Tables
- System Statistics
- Gathering and Managing Optimizer Statistics

Gathering Statistics: Overview

- Optimizer statistics collection is the gathering of optimizer statistics for database objects.
- You can gather statistics using the following:
  - Automatic optimizer statistics gathering
  - Manual statistics gathering
    - `DBMS_STATS` package
  - Adaptive statistics
    - Dynamic statistics
    - Automatic re-optimization
    - SQL plan directives
  - Online statistics gathering for bulk loads

Selectivity:	
Equality	**1%**
Inequality	**5%**
Other predicates	**5%**
Table row length	**20**
# of index leaf blocks	**25**
# of distinct values	**100**
Table cardinality	**100**
Remote table cardinality	**2000**

In Oracle Database, optimizer statistics collection is the gathering of optimizer statistics for database objects. The database can collect these statistics automatically, or you can collect them manually by using the system-supplied `DBMS_STATS` package. These are discussed in more detail in the subsequent slides. It is recommended that you use automatic statistics gathering for objects.

The contents of tables and associated indexes change frequently, which can lead the optimizer to choose suboptimal execution plans for queries. Thus, statistics must be kept current to avoid any potential performance issues because of suboptimal plans.

To minimize DBA involvement, Oracle Database automatically gathers optimizer statistics at various times. Some automatic options are configurable, such as enabling AutoTask to run `DBMS_STATS`. You can manage optimizer statistics either through Oracle Enterprise Manager Cloud Control (Cloud Control) or using PL/SQL on the command line.

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 (including remote tables and external tables), it does not perform dynamic sampling. In those cases and also when dynamic sampling has been disabled, the optimizer uses default values for its statistics.

Note: For more information about Managing Statistics, refer to Appendix F titled “Gathering and Managing Optimizer Statistics.”

Manual Statistics Gathering

You can use Enterprise Manager and the `DBMS_STATS` package to:

- Generate and manage statistics for use by the optimizer:
  - Gather or modify
  - View or name
  - Export or import
  - Delete or lock
- Gather statistics on:
  - Indexes, tables, columns, and partitions
  - Object, schema, or database
- Gather statistics either serially or in parallel
- Gather or set system statistics (currently not possible in EM)

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

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

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

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

When to Gather Statistics Manually

- Rely mostly on automatic statistics collection:
  - Change the frequency of automatic statistics collection to meet your needs.
  - Remember that `STATISTICS_LEVEL` should be set to `TYPICAL` or `ALL` for automatic statistics collection to work properly.
- Gather statistics manually for:
  - Objects that are volatile
  - Objects modified in batch operations (Gather statistics as part of the batch operation.)
  - External tables, system statistics, and fixed objects
  - New objects (Gather statistics right after object creation.)

The automatic statistics gathering mechanism gathers 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 percent or more to the object's total size between statistics-gathering intervals

For external tables, statistics are collected manually only by using `GATHER_TABLE_STATS`. Because sampling on external tables is not supported, 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 system statistics and fixed objects, such as the dynamic performance tables. These statistics are not automatically gathered.

Managing Statistics: Overview (Export / Import / Lock / Restore / Publish)

- Purpose:
  - To revert to pre-analyzed statistics if gathering statistics cause critical statements to perform badly
  - To test the new statistics before publishing
- Importing previously exported statistics (9*i*)
- Locking and unlocking statistics on a specific table (10*g*)
- Restoring statistics archived before gathering (10*g*)
- Keeping statistics pending before publishing (11*g*R2)

In the next few slides, the following topics are covered to discuss how to manage optimizer statistics:

- Importing previously exported statistics (9*i*)
- Locking and unlocking statistics on a specific table (10*g*)
- Restoring statistics archived before gathering (10*g*)
- Keeping statistics pending before publishing (11*g*R2)

Note: For more information about managing statistics, refer to Appendix F titled “Gathering and Managing Optimizer Statistics.”

Quiz

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

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

Answer: a, c, d, e

Summary

In this lesson, you should have learned how to:

- Describe optimizer statistics
  – Table statistics
  – Index statistics
  – Column statistics (histogram)
  – Column statistics (extended statistics)
  – Session-specific statistics for global temporary tables
  – System statistics
- Gather and manage optimizer statistics

Practice 11: Overview

This practice covers the following topics:

- Using the index clustering factor
- Creating expression statistics
- Enabling automatic statistics gathering (Optional)
- Using system statistics (Optional)