Objectives

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

• Describe course objectives
• Describe course schedule
Course Objectives

Proactive Tuning:
• Describe the basic steps in processing SQL statements
• Describe the causes of performance problems
• Understand where SQL tuning fits in an overall tuning methodology
• Influence the physical data model so as to avoid performance problems
• Understand query optimizer behavior
• Influence the optimizer behavior
Course Objectives

Reactive Tuning:
- Use the diagnostic tools to gather information about SQL statement processing
- Describe Automatic SQL Tuning
- Describe alternative methods of accessing data
# Course Schedule

<table>
<thead>
<tr>
<th>Day</th>
<th>Lesson</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1. Oracle Database Architecture</td>
</tr>
<tr>
<td></td>
<td>2. Following a tuning methodology</td>
</tr>
<tr>
<td></td>
<td>3. Designing and developing for performance</td>
</tr>
<tr>
<td></td>
<td>4. Introducing the optimizer</td>
</tr>
<tr>
<td></td>
<td>5. Optimizer Operations</td>
</tr>
<tr>
<td></td>
<td>6. Execution Plans</td>
</tr>
<tr>
<td>2</td>
<td>7. Generating Execution Plans</td>
</tr>
<tr>
<td></td>
<td>8. Application Tracing</td>
</tr>
<tr>
<td></td>
<td><strong>Workshop 1</strong></td>
</tr>
</tbody>
</table>
# Course Schedule

<table>
<thead>
<tr>
<th>Day</th>
<th>Lesson</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9. Identifying High load SQL</td>
</tr>
<tr>
<td></td>
<td>10. Automatic SQL Tuning</td>
</tr>
<tr>
<td></td>
<td>Workshop 2</td>
</tr>
<tr>
<td>3</td>
<td>11. Using Indexes</td>
</tr>
<tr>
<td></td>
<td>12. Different Types of Indexes</td>
</tr>
<tr>
<td></td>
<td>Workshop 3 and 4</td>
</tr>
<tr>
<td></td>
<td>13. Using Hints</td>
</tr>
<tr>
<td></td>
<td>Workshop 5</td>
</tr>
<tr>
<td></td>
<td>14. Materialized Views</td>
</tr>
<tr>
<td></td>
<td>Workshop 6 and Optional Workshop 7</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned to:

• Describe course objectives
• Describe course schedule
Oracle Database Architecture: Overview
Objectives

After completing this lesson, you should be able to:

• Describe the Oracle Database architecture and components
• Make qualified decisions about your tuning actions
Oracle Database Architecture: Overview

• The Oracle Database consists of two main components:
  – The database: physical structures
  – The instance: memory structures
• The size and structure of these components impact performance.
Oracle Instance Management

SGA
- Shared pool
- Streams pool
- Large pool
- Java pool
- Database buffer cache
- Redo log buffer

System Monitor SMON
- Process Monitor PMON
- Database Writer DBW0
- Log Writer LGWR

Check point CKPT
- Control file
- Data files
- Redo log files

Archiver ARC0
- Archived redo log files
Database Physical Structure

Control files

Data files

Online redo log files

Parameter file

Password file

Archive log files
Oracle Memory Structures

- SGA
  - Shared pool
  - Streams pool
  - Large pool
  - Java Pool
  - Database buffer cache
  - Redo log buffer

- PGA
  - Server process 1
  - Server process 2
  - Server process 3
Automatic Shared Memory Management

SGA
- Shared pool
- Fixed SGA
- Large pool
- Java pool
- Database buffer cache
- Redo log buffer

Which size to choose?
Shared Pool

The shared pool consists of:

- Data dictionary cache containing information on objects, storage, and privileges
- Library cache containing information such as SQL statements, parsed or compiled PL/SQL blocks, and Java classes

Appropriate sizing of the shared pool affects performance by:

- Reducing disk reads
- Allowing shareable SQL code
- Reducing parsing, thereby saving CPU resources
- Reducing latching overhead, thereby improving scalability
Shared SQL Areas

- Cursor for SELECT statement 2
- Cursor for SELECT statement 1

SGA

Shared SQL

- SELECT statement 1
- SELECT statement 2
- SELECT statement 1

User A

User B

User C
Program Global Area (PGA)

- PGA is a memory area that contains:
  - Session information
  - Cursor information
  - SQL execution work areas
    - Sort area
    - Hash join area
    - Bitmap merge area
    - Bitmap create area
- Work area size influences SQL performance.
- Work areas can be automatically or manually managed.
Automated SQL Execution Memory (PGA) Management

• Allocation and tuning of PGA memory is simplified and improved.
  – Efficient memory allocation for varying workloads
  – Queries optimized for both throughput and response times

• DBAs can use parameters to specify the policy for PGA sizing.
Connecting to an Instance

User → Server

Oracle database

Client

Application server

Browser

Server
SQL Statement Processing Phases

- Open
- Parse
- Bind
- Execute
- Fetch
- Close
SQL Statement Processing Phases: Parse

- **Parse phase:**
  - Searches for the statement in the shared pool
  - Checks syntax
  - Checks semantics and privileges
  - Merges view definitions and subqueries
  - Determines execution plan

- **Minimize parsing as much as possible:**
  - Parse calls are expensive.
  - Avoid reparsing
  - Parse once, execute many times
SQL Statement Processing Phases: Bind

- **Bind phase:**
  - Checks the statement for bind variables
  - Assigns or reassigns a value to the bind variable

- **Bind variables impact performance when:**
  - They are not used, and your statement would benefit from a shared cursor
  - They are used, and your statement would benefit from a different execution plan
SQL Statement Processing Phases: Execute and Fetch

- **Execute phase:**
  - Executes the SQL statement
  - Performs necessary I/O and sorts for data manipulation language (DML) statements

- **Fetch phase:**
  - Retrieves rows for a query
  - Sorts for queries when needed
  - Uses an array fetch mechanism
Processing a DML Statement

1. Server process receives the DML statement: `UPDATE employees ...`
2. Database buffer cache
3. Redo log buffer
4. User process
Commit Processing

Instance

SGA
- Database buffer cache
- Redo log buffer

Shared pool

Server process

User process

Database
- Data files
- Control files
- Redo log files

User process

LGWR
Functions of the Oracle Query Optimizer

The Oracle query optimizer determines the most efficient execution plan and is the most important step in the processing of any SQL statement.

The optimizer:

• Evaluates expressions and conditions
• Uses object and system statistics
• Decides how to access the data
• Decides how to join tables
• Decides which path is most efficient
Top Database Performance Issues

- Bad connection management
- Poor use of cursors and the shared pool
- Bad SQL
- Nonstandard initialization parameters
- I/O issues
- Long full-table scans
- In-disk sorts
- High amounts of recursive SQL
- Schema errors and optimizer problems
Summary

In this lesson, you should have learned about the Oracle Database architecture and various components that require tuning.
Following a Tuning Methodology
Objectives

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

• Determine performance problems
• Manage performance
• Describe tuning methodologies
• Identify goals for tuning
• Describe automatic SQL tuning features
• List manual SQL tuning steps
Performance Problems

- Inadequate consumable resources
  - CPU
  - I/O
  - Memory (may be detected as an I/O problem)
  - Data communications resources
- High-load SQL
- Contention
Factors to Be Managed

- **Schema**
  - Data design
  - Indexes
- **Application**
  - SQL statements
  - Procedural code
- **Instance**
- **Database**
- **User expectations**
- **Hardware and network tuning**
Tuning Goals

- Reduce the response time
- Reduce resource usage
Overview of SQL Tuning

1. Identify causes of poor performance.
2. Identify problematic SQL.
   - Automatic: ADDM, Top SQL
   - Manual: V$ views, statspack
3. Apply a tuning method.
   - Manual tuning
   - Automatic SQL tuning
4. Implement changes to:
   - SQL statement constructs
   - Access structures such as indexes
Identifying High-Load SQL

Identify high-load or problematic SQL

- ADDM
- Top SQL report

- Dynamic performance views
- Statspack
Manual Tuning

1. Gather information about the referenced objects.
2. Gather optimizer statistics.
3. Review execution plans.
4. Restructure SQL statements.
5. Restructure indexes and create materialized views.
6. Maintain execution plans.
Gather Information About Referenced Objects

- SQL text
- Structure of tables and indexes
- Optimizer statistics
- Views
- Optimizer plan: current and prior
Gathering Optimizer Statistics

- Gather statistics for all tables.
- Gather new statistics when existing statistics become stale.
Reviewing the Execution Plan

- Driving table has the best filter.
- Fewest number of rows are being returned to the next step.
- The join method is appropriate for the number of rows being returned.
- Views are used efficiently.
- There are no unintentional Cartesian products.
- Each table is being accessed efficiently.
- Examine the predicates in the SQL statement and the number of rows in the table.
- A full table scan does not mean inefficiency.
Restructuring the SQL Statements

- Compose predicates by using `AND` and `=`.
- Avoid transformed columns in the `WHERE` clause.
- Avoid mixed-mode expressions and beware of implicit type conversions.
- Write separate SQL statements for specific tasks and use SQL constructs appropriately.
- Use `EXISTS` or `IN` for subqueries as required.
- Cautiously change the access path and join order with hints.
Restructuring the Indexes

• Remove unnecessary indexes to speed the DML.
• Index the performance-critical access paths.
• Reorder columns in existing concatenated indexes.
• Add columns to the index to improve selectivity.
• Create appropriate indexes based on usage type:
  – B*tree
  – Bitmap
  – Bitmap join
  – Concatenated
• Consider index-organized tables.
Maintaining Execution Plans over Time

- Stored outlines
- Stored statistics
- Locking statistics
Automatic SQL Tuning

• Automatic SQL tuning facilitates these steps:
  – Gather information on the referenced objects.
  – Verify optimizer statistics.
  – Review execution plans.
  – Restructure SQL statements
  – Restructure indexes and create materialized views.
  – Maintain execution plans.

• Four types of analysis are performed in automatic SQL tuning:
  – Statistics analysis
  – SQL profiling
  – Access path analysis
  – SQL structure analysis
Automatic Tuning Mechanisms

You can perform automatic SQL tuning using:

- SQL Tuning Advisor
- SQL Access advisor
SQL Tuning Advisor

The SQL Tuning Advisor does the following:

- **Accepts input from:**
  - Automatic Database Diagnostic Monitor (ADDM)
  - Automatic Workload Repository (AWR)
  - Cursor cache
  - Custom SQL as defined by the user

- **Provides:**
  - Recommendations
  - Rationale
  - Expected benefits
  - SQL commands for implementing the recommendations
The SQL Access Advisor does the following:

- Provides comprehensive advice on schema design by accepting input from:
  - Cursor cache
  - Automatic Workload Repository (AWR)
  - User-defined workload
  - Hypothetical workload if a schema contains dimensions or primary/foreign key relationships

- Analyzes the entire workload and recommends:
  - Creating new indexes as needed
  - Dropping any unused indexes
  - Creating new materialized views and materialized view logs
Summary

In this lesson, you should have learned how to:

• Manage performance
  – Start early; be proactive
  – Set measurable objectives
  – Monitor requirements compliance
  – Handle exceptions and changes

• Identify performance problems
  – Inadequate consumable resources
  – Inadequate design resources
  – Critical resources
  – Excessive demand
Summary

In this lesson, you should have learned how to:

• Tune SQL statements
  – Analyze the results at each step
  – Tune the physical schema
  – Choose when to use SQL
  – Reuse SQL statements when possible
  – Design and tune the SQL statement
  – Get maximum performance with the optimizer
Designing and Developing for Performance
Objectives

After completing this lesson, you should be able to describe the basic steps involved in designing and developing for performance.
Understanding Scalability

- Scalability is a system’s ability to process more workload, with a proportional increase in system resource use.
- Poor scalability leads to system resource exhaustion to the extent that additional throughput is impossible when the system’s workload is increased.
Scalability with Application Design, Implementation, and Configuration

Applications have a significant impact on scalability.

- Poor schema design can cause expensive SQL that does not scale.
- Poor transaction design can cause locking and serialization problems.
- Poor connection management can cause unsatisfactory response times and unreliable systems.
Configuring the Appropriate System Architecture for Your Requirements

- Interactive applications (OLTP)
- Process-driven applications (OLAP)
Proactive Tuning Methodology

- Simple design
- Data modeling
- Tables and indexes
- Using views
- Writing efficient SQL
- Cursor sharing
- Using bind variables
- SQL versus PL/SQL
- Dynamic SQL
Simplicity In Application Design

- Simple tables
- Well-written SQL
- Indexing only as required
- Retrieving only required information
Data Modeling

- Accurately represent business practices
- Focus on the most frequent and important business transactions
- Use modeling tools
- Normalize the data
Table Design

• Compromise between flexibility and performance
  – Principally normalize
  – Selectively denormalize

• Use Oracle performance features
  – Default values
  – Check constraints
  – Materialized views
  – Clusters

• Focus on business-critical tables
Index Design

• Index keys
  – Primary key
  – Unique key
  – Foreign keys

• Index data that is frequently queried

• Use SQL as a guide to index design
Using Views

- Simplifies application design
- Is transparent to the end user
- Can cause suboptimal execution plans
SQL Execution Efficiency

- Good database connectivity
- Using cursors
- Minimizing parsing
- Using bind variables
Importance of Sharing Cursors

- Reduces parsing
- Dynamically adjusts memory
- Improves memory usage
Writing SQL to Share Cursors

• Create generic code using the following:
  – Stored procedures and packages
  – Database triggers
  – Any other library routines and procedures

• Write to format standards:
  – Case
  – White space
  – Comments
  – Object references
  – Bind variables
Controlling Shared Cursors

The `CURSOR_SHARING` initialization parameter can be set to:

- **EXACT** (default)
- **SIMILAR** (not recommended)
- **FORCE**
Performance Checklist

- Set initialization parameters and storage options.
- Verify resource usage of SQL statements.
- Validate connections by middleware.
- Verify cursor sharing.
- Validate migration of all required objects.
- Verify validity and availability of optimizer statistics.
Summary

In this lesson, you should have learned the basic steps that are involved in designing and developing for performance.
Introduction to the Optimizer
Objectives

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

• Describe the functions of the Oracle optimizer
• Identify the factors influencing the optimizer
• Set the optimizer approach at the instance and session level
Oracle Optimizer

The optimizer creates an execution plan for every SQL statement by:

- Evaluating expressions and conditions
- Using object and system statistics
- Deciding how to access the data
- Deciding how to join tables
- Deciding which path is most efficient
- Comparing the cost for execution of different plans
- Determining the least-cost plan
Functions of the Query Optimizer

- Parsed query (from parser)
  - Query transformer
    - Transformed query
      - Estimator
        - Query + estimates
          - Plan generator
            - Query plan
              (to row-source generator)
    - Statistics
      - Dictionary
Selectivity

• Selectivity represents a fraction of rows from a row set.
• Selectivity lies in a value range from 0.0 to 1.0.
• When statistics are available, the estimator uses them to estimate selectivity.
• With histograms on columns that contain skewed data, the results are good selectivity estimates.
Cardinality and Cost

- Cardinality represents the number of rows in a row set.
- Cost represents the units of work or resource that are used.
Query Optimizer Statistics in the Data Dictionary

• The Oracle optimizer requires statistics to determine the best execution plan.

• Statistics
  – Stored in the data dictionary tables
  – Must be true representations of data
  – Gathered using:
    DBMS_STATS package
    Dynamic sampling
Enabling Query Optimizer Features

- The optimizer behavior can be set to prior releases of the database.
- The `OPTIMIZER_FEATURES_ENABLE` initialization parameter can be set to values of different database releases (such as 8.1.7 or 10.0.0).
- Example:

```
OPTIMIZER_FEATURES_ENABLE=9.2.0;
```
Controlling the Behavior of the Optimizer

Optimizer behavior can be controlled using the following initialization parameters:

- CURSOR_SHARING
- DB_FILE_MULTIBLOCK_READ_COUNT
- OPTIMIZER_INDEX_CACHING
- OPTIMIZER_INDEX_COST_ADJ
- OPTIMIZER_MODE
- PGA_AGGREGATE_TARGET
Choosing an Optimizer Approach

- `OPTIMIZER_MODE` initialization parameter
- `OPTIMIZER_MODE` parameter of `ALTER SESSION` statement
- Optimizer statistics in the data dictionary
- Optimizer SQL hints for influencing the optimizer decision
Setting the Optimizer Approach

• At the instance level, set the following parameter:

```
OPTIMIZER_MODE = {FIRST_ROWS(_n) | ALL_ROWS}
```

• For a session, use the following SQL command:

```
ALTER SESSION SET optimizer_mode = {first_rows(_n) | all_rows}
```
Optimizing for Fast Response

- OPTIMIZER_MODE is set to FIRST_ROWS or FIRST_ROWS_n, where n is 1, 10, 100, or 1000.
- This approach is suitable for online users.
- The optimizer generates a plan with the lowest cost to produce the first row or the first few rows.
- The value of n should be chosen based on the online user requirement (specifically, how the result is displayed to the user).
- The optimizer explores different plans and computes the cost to produce the first n rows for each plan.
Optimizing SQL Statements

Best throughput
- Time required to complete the request
- Suitable for:
  - Batch processing
  - Report applications

Fast response
- Time for retrieving the first rows
- Suitable for:
  - Interactive applications
  - Web-based or GUI applications
How the Query Optimizer Executes Statements

The factors considered by the optimizer are:

- Access path
- Join method
- Join order
Access Paths

- Full-table scans
- Row ID scans
- Index scans
- Cluster scans
- Hash scans
Join Orders

A join order is the order in which different join items (such as tables) are accessed and joined together.
Join Methods

The different join methods considered by the optimizer are:
- Nested-loop join
- Hash join
- Sort-merge join
- Cartesian join
Summary

In this lesson, you should have learned about:

- Functions of the optimizer
- Cost factors that are considered by the optimizer
- Setting the optimizer approach
Optimizer Operations
Objectives

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

• Describe different access paths
• Optimize sort performance
• Describe different join techniques
• Explain join optimization
• Find optimal join execution plans
Review: How the Query Optimizer Executes Statements

The factors considered by the optimizer are:

- Access path
- Join order
- Join method
Access Paths

- Full table scan
- Row ID scan
- Index scan
- Sample table scan
Choosing an Access Path

- Available access paths for the statement
- Estimated cost of executing the statement, using each access path or combination of paths
Full Table Scans

- Lack of index
- Large amount of data
- Small table
Row ID Scans

- The row ID specifies the data file and data block containing the row as well as the location of the row in that block.
- Using the row ID is the fastest way to retrieve a single row.
- Every index scan does not imply access by row ID.
Index Scans

Types of index scans:
- Index unique scan
- Index range scan
- Index range scan descending
- Index skip scan
Index Scans

Types of index scans:
- Full scan
- Fast-full index scan
- Index join
- Bitmap join
Joining Multiple Tables

You can join only two row sources at a time. Joins with more than two tables are executed as follows:

1. Two tables are joined, resulting in a row source.
2. The next table is joined with the row source that results from step 1.
3. Step 2 is repeated until all tables are joined.
Join Terminology

- Join statement
- Join predicate, nonjoin predicate
- Single-row predicate

```
SELECT c.cust_last_name, c.cust_first_name, co.country_id, co.country_name
FROM customers c JOIN countries co
ON (c.country_id = co.country_id)
AND (co.country_id = '52790'
OR c.cust_id = 205);
```
Join Terminology

• Natural join

```sql
SELECT c.cust_last_name, co.country_name
FROM customers c NATURAL JOIN countries co;
```

• Join with nonequal predicate

```sql
SELECT s.amount_sold, p.promo_name
ON( s.time_id
From sales s, promotions p
BETWEEN p.promo_begin_date
AND p.promo_end_date );
```

• Cross join

```sql
SELECT *
FROM customers c CROSS JOIN countries co;
```
SQL:1999 Outer Joins

- Plus (+) sign is not used.
- **Keyword** OUTER JOIN **is used instead.**

```sql
SELECT s.time_id, t.time_id
FROM   sales s
RIGHT OUTER JOIN times t
ON     (s.time_id = t.time_id);
```
Oracle Proprietary Outer Joins

- Join predicates with a plus (+) sign
- Nonjoin predicates with a plus (+) sign
- Predicates without a plus (+) sign disable outer joins

```
SELECT s.time_id, t.time_id
FROM sales s, times t
WHERE s.time_id (+) = t.time_id;
```
Full Outer Joins

- A full outer join acts like a combination of the left and right outer joins.
- In addition to the inner join, rows in both tables that have not been returned in the result of the inner join are preserved and extended with nulls.

```sql
SELECT c.cust_id, c.cust_last_name, co.country_name
FROM customers c
FULL OUTER JOIN countries co
ON (c.country_id = co.country_id);
```
Execution of Outer Joins

Indexes can be used for outer join predicates.

```
SELECT  c.cust_id,  co.country_name
FROM    customers c
LEFT OUTER JOIN countries co
ON     (c.country_id = co.country_id)
AND     co.country_id = 'IT';
```
Join Order Rules

Rule 1
A *single-row predicate* forces its row source to be placed first in the join order.

Rule 2
For *outer joins*, the table with the outer-joined table must come after the other table in the join order for processing the join.
Join Optimization

• As a first step, a list of possible join orders is generated.
• This potentially results in the following:

<table>
<thead>
<tr>
<th>Number of Tables</th>
<th>Join Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2! = 2</td>
</tr>
<tr>
<td>3</td>
<td>3! = 6</td>
</tr>
<tr>
<td>4</td>
<td>4! = 24</td>
</tr>
</tbody>
</table>

• Parse time grows factorially when adding tables to a join.
Join Methods

- A join operation combines the output from two row sources and returns one resulting row source.
- Join operation types include the following:
  - Nested loop join
  - Sort-merge join
  - Hash join
Nested Loop Joins

- One of the two tables is defined as the *outer* table (or the *driving* table).
- The other table is called the *inner* table.
- For each row in the outer table, all matching rows in the inner table are retrieved.
Nested Loop Join Plan

Nested loops

1

2  3

Table access (Outer/driving table)  Table access (Inner table)
When Are Nested Loop Joins Used?

Nested loop joins are used when:

- Joining a few rows that have a good driving condition
- Order of tables is important
- USE_NL(table1 table2) hint is used
Hash Joins

A hash join is executed as follows:

• Both tables are split into as many partitions as required, using a full table scan.
• For each partition pair, a hash table is built in memory on the smallest partition.
• The other partition is used to probe the hash table.
Hash Join Plan

Hash join

Table access

Table access
When Are Hash Joins Used?

- Hash joins are used if either of the following conditions is true:
  - A large amount of data needs to be joined.
  - A large fraction of the table needs to be joined.
- Use the `USE_HASH` hint.
Sort-Merge Joins

A sort-merge join is executed as follows:

1. The rows from each row source are sorted on the join predicate columns.
2. The two sorted row sources are then merged and returned as the resulting row source.
Sort-Merge Join Plan

1. Merge
2. Sort
3. Sort
4. Table access
5. Table access
When Are Sort-Merge Joins Used?

Sort-merge joins can be used if either of the following conditions is true:

- Join condition is not an equijoin.
- Sorts are required for other operations.
Star Joins

Dimension tables

PRODUCTS
CUSTOMERS

Facts table

CHANNELS
PROMOTIONS
TIMES
How the Query Optimizer Chooses Execution Plans for Joins

The query optimizer determines:

- Row sources
- Type of join
- Join method
- Cost of execution plans
- Other costs such as:
  - I/O
  - CPU time
  - `DB_FILE_MULTIBLOCK_READ_COUNT`
- Hints specified
Subqueries and Joins

- Subqueries (like joins) are statements that reference multiple tables
- Subquery types:
  - Noncorrelated subquery
  - Correlated subquery
  - \texttt{NOT IN} subquery (antijoin)
  - \texttt{EXISTS} subquery (semijoin)
Sort Operations

- SORT UNIQUE
- SORT AGGREGATE
- SORT GROUP BY
- SORT JOIN
- SORT ORDER BY
Tuning Sort Performance

- Because sorting large sets can be expensive, you should tune sort parameters.
- **Note that** `DISTINCT`, `GROUP BY`, and most set operators cause implicit sorts.
- Minimize sorting by one of the following:
  - Try to avoid `DISTINCT` and `GROUP BY`.
  - Use `UNION ALL` instead of `UNION`.
  - Enable index access to avoid sorting.
Top-N SQL

```
SELECT *
FROM (SELECT prod_id
      , prod_name
      , prod_list_price
      , prod_min_price
      FROM products
      ORDER BY prod_list_price DESC)
WHERE ROWNUM <= 5;
```
Memory and Optimizer Operations

- Memory-intensive operations use up work areas in the Program Global Area (PGA).
- Automatic PGA memory management simplifies and improves the way PGA memory is allocated.
- The size of a work area must be big enough to avoid multi-pass execution.
- A reasonable amount of PGA memory allows single-pass executions.
- The size of PGA is controlled with:
  - PGA_AGGREGATE_TARGET
  - WORKAREA_SIZE_POLICY
Summary

In this lesson, you should have learned how to:

• Describe available join operations
• Optimize join performance against different requirements
• Influence the join order
• Explain why tuning joins is more complicated than tuning single table statements
Execution Plans
Objectives

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

- Use the `EXPLAIN PLAN` command to show how a statement is processed
- Use the `DBMS_XPLANE` package
- Use the Automatic Workload Repository
- Query the `V$SQL_PLAN` performance view
- Use the SQL*Plus `AUTOTRACE` setting to show SQL statement execution plans and statistics
What Is an Execution Plan?

An execution plan is a set of steps that are performed by the optimizer in executing a SQL statement and performing an operation.
Methods for Viewing Execution Plans

- EXPLAIN PLAN
- SQL Trace
- Statspack
- Automatic Workload Repository
- V$SQL_PLAN
- SQL*Plus AUTOTRACE
Using Execution Plans

- Determining the current execution plan
- Identifying the effect of indexes
- Determining access paths
- Verifying the use of indexes
- Verifying which execution plan may be used
DBMS_XPLAN Package: Overview

• The DBMS_XPLAN package provides an easy way to display the output from:
  – EXPLAIN PLAN command
  – Automatic Workload Repository (AWR)
  – V$SQL_PLAN and V$SQL_PLAN_STATISTICS_ALL fixed views

• The DBMS_XPLAN package supplies three table functions that can be used to retrieve and display the execution plan:
  – DISPLAY
  – DISPLAY_CURSOR
  – DISPLAY_AWR
EXPLAIN PLAN Command

- Generates an optimizer execution plan
- Stores the plan in the PLAN table
- Does not execute the statement itself
EXPLAIN PLAN Command

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'text'
  INTO your plan table
  FOR statement
```
EXPLAIN PLAN Command: Example

```
EXPLAIN PLAN
SET STATEMENT_ID = 'demo01' FOR
SELECT e.last_name, d.department_name
FROM hr.employees e, hr.departments d
WHERE e.department_id = d.department_id;
```

Explained.

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

```sql
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

Plan hash value: 2933537672

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>106</td>
<td>2862</td>
<td>6 (17)</td>
</tr>
<tr>
<td>1</td>
<td>MERGE JOIN</td>
<td></td>
<td>106</td>
<td>2862</td>
<td>6 (17)</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPARTMENTS</td>
<td>27</td>
<td>432</td>
<td>2 (0)</td>
</tr>
<tr>
<td>3</td>
<td>INDEX FULL SCAN</td>
<td>DEPT_ID_PK</td>
<td>27</td>
<td></td>
<td>1 (0)</td>
</tr>
<tr>
<td>*4</td>
<td>SORT JOIN</td>
<td></td>
<td>107</td>
<td>1177</td>
<td>4 (25)</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS FULL</td>
<td>EMPLOYEES</td>
<td>107</td>
<td>1177</td>
<td>3 (0)</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
    filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")

18 rows selected.
Parse Tree

SELECT STATEMENT

MERGE JOIN

SORT JOIN

FULL TABLE SCAN of EMPLOYEES

INDEX FULL SCAN DEPT_ID_PK

TABLE ACCESS BY INDEX ROWID of DEPARTMENTS

Oracle
Using the `V$SQL_PLAN` View

- `V$SQL_PLAN` provides a way of examining the execution plan for cursors that were recently executed.
- Information in `V$SQL_PLAN` is very similar to the output of an `EXPLAIN PLAN` statement:
  - `EXPLAIN PLAN` shows a theoretical plan that can be used if this statement were to be executed.
  - `V$SQL_PLAN` contains the actual plan used.
### V$SQL_PLAN Columns

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASH_VALUE</td>
<td>Hash value of the parent statement in the library cache</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>Object number of the table or the index</td>
</tr>
<tr>
<td>CHILD_NUMBER</td>
<td>Child cursor number using this execution plan</td>
</tr>
<tr>
<td>POSITION</td>
<td>Order of processing for operations that all have the same PARENT_ID</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>ID of the next execution step that operates on the output of the current step</td>
</tr>
<tr>
<td>ID</td>
<td>Number assigned to each step in the execution plan</td>
</tr>
</tbody>
</table>

Note: This is only a partial listing of the columns.
Querying \texttt{V\\$SQL\\_PLAN}

```sql
SELECT PLAN_TABLE_OUTPUT FROM
TABLE(DBMS_XPLAN.DISPLAY_CURSOR('47ju6102uvq5q'));
```

SQL ID 47ju6102uvq5q, child number 0

```sql
SELECT e.last_name, d.department_name
FROM hr.employees e, hr.departments d
WHERE e.department_id = d.department_id
```

Plan hash value: 2933537672

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td>6 (100)</td>
</tr>
<tr>
<td>1</td>
<td>MERGE JOIN</td>
<td></td>
<td>106</td>
<td>2862</td>
<td>6 (17)</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPARTMENTS</td>
<td>27</td>
<td>432</td>
<td>2 (0)</td>
</tr>
<tr>
<td>3</td>
<td>INDEX FULL SCAN</td>
<td>DEPT_ID_PK</td>
<td>27</td>
<td></td>
<td>1 (0)</td>
</tr>
<tr>
<td>* 4</td>
<td>SORT JOIN</td>
<td></td>
<td>107</td>
<td>1177</td>
<td>4 (25)</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS FULL</td>
<td>EMPLOYEES</td>
<td>107</td>
<td>1177</td>
<td>3 (0)</td>
</tr>
</tbody>
</table>
```

Predicate Information (identified by operation id):

```
4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
  filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

24 rows selected.
V$SQL_PLAN_STATISTICS View

- V$SQL_PLAN_STATISTICS provides actual execution statistics.
- V$SQL_PLAN_STATISTICS_ALL enables side-by-side comparisons of the optimizer estimates.
Automatic Workload Repository

- Collects, processes, and maintains performance statistics for problem-detection and self-tuning purposes
- Statistics include:
  - Object statistics
  - Time-model statistics
  - Some system and session statistics
  - Active Session History (ASH) statistics
- Automatically generates snapshots of the performance data
Managing AWR with PL/SQL

- Creating snapshots
- Dropping snapshots
- Managing snapshot settings
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**
Querying the AWR

```sql
SELECT PLAN_TABLE_OUTPUT FROM TABLE
(DBMS_XPLAN.DISPLAY_AWR('454rug2yva18w'));
```

**PLAN_TABLE_OUTPUT**

---

**SQL_ID** 454rug2yva18w

---

`select /* example */ * from hr.employees natural join hr.departments`

**Plan hash value:** 4179021502

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td>6 (100)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>HASH JOIN</td>
<td></td>
<td>11</td>
<td>968</td>
<td>6 (17)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>DEPARTMENTS</td>
<td>11</td>
<td>220</td>
<td>2 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>DEPARTMENTS</td>
<td>11</td>
<td>220</td>
<td>2 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>EMPLOYEES</td>
<td>107</td>
<td>7276</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>
SQL*Plus AUTOTRACE

SET AUTOTRACE
  OFF
  ON
  TRACE [ONLY]

EXPLAIN
STATISTICS

SHOW AUTOTRACE
SQL*Plus AUTOTRACE: Examples

- To start tracing statements using AUTOTRACE
  `set autotrace on`

- To hide statement output
  `set autotrace traceonly`

- To display execution plans only
  `set autotrace traceonly explain`

- Control the layout with column settings
SQL*Plus AUTOTRACE: Statistics

```
set autotrace traceonly statistics
SELECT *
FROM products;
```

Statistics
--------
1 recursive calls
0 db block gets
9 consistent gets
3 physical reads
0 redo size
15028 bytes sent via SQL*Net to client
556 bytes received via SQL*Net from client
6 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
72 rows processed
Summary

In this lesson, you should have learned how to:

• Use `EXPLAIN PLAN` to view execution plans
• Query `V$SQL_PLAN` to see the execution plan for cursors that were recently executed
• Use the Automatic Workload Repository
• Use SQL*Plus `AUTOTRACE` to run statements and display execution plans and statistics
Practice 6: Overview

This practice covers the following topics:

- Using AUTOTRACE
- Using EXPLAIN PLAN
- Using AWR
- Retrieving the execution plan using DBMS_XPLAN
Gathering Statistics
Objectives

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

- Identify table, index, and column statistics
- Describe the Automatic Statistics Gathering mechanism
- Use the DBMS_STATS package to collect statistics manually
- Identify predicate selectivity calculations
What Are Optimizer Statistics?

- Collection of data that describes the database and the objects in the database
- Information used by query optimizer to estimate:
  - Selectivity of predicates
  - Cost of each execution plan
  - Access method and join method
  - CPU and I/O costs
Types of Optimizer Statistics

- **Object statistics**
  - Table statistics
  - Column statistics
  - Index statistics

- **System statistics**
  - I/O performance and utilization
  - CPU performance and utilization
How Statistics Are Gathered

- **Automatic statistics gathering**
  - GATHER_STATS_JOB
- **Manual statistics gathering**
  - DBMS_STATS package
- **Dynamic sampling**
Automatic Statistics Gathering

- **Oracle Database 10g** automates optimizer statistics collection:
  - Statistics are gathered automatically on all database objects.
  - `GATHER_STATS_JOB` is used for statistics collection and maintenance.
  - Scheduler interface is used for scheduling the maintenance job.

- **Automated statistics collection:**
  - Eliminates need for manual statistics collection
  - Significantly reduces the chances of getting poor execution plans
Manual Statistics Gathering

You can use the `DBMS_STATS` package to:

• Generate and manage statistics for use by the optimizer
• Gather, modify, view, export, import, and delete statistics
• Identify or name statistics that are gathered
• Gather statistics on:
  – Indexes, tables, columns, and partitions
  – All schema objects in a schema or database
• Gather statistics either serially or in parallel
Managing Automatic Statistics Collection

- Job configuration options
- Statistics-collection configuration options
Job Configuration Options

- **Setting status:** ENABLED or DISABLED
- **Maintaining schedule:** maintenance window
Managing the Job Scheduler

Verifying Automatic Statistics Gathering:

```
SELECT owner, job_name, enabled
FROM DBA_SCHEDULER_JOBS
WHERE JOB_NAME = 'GATHER_STATS_JOB';
```

Disabling and enabling Automatic Statistics Gathering:

```
BEGIN
DBMS_SCHEDULER.DISABLE('GATHER_STATS_JOB');
END;
/

BEGIN
DBMS_SCHEDULER.ENABLE('GATHER_STATS_JOB');
END;
/
```
Managing the Maintenance Window

- WEEKNIGHT_WINDOW
- WEEKEND_WINDOW

```sql
EXECUTE DBMS_SCHEDULER.SET_ATTRIBUTE(
    'WEEKNIGHT_WINDOW',
    'repeat_interval',
    'freq=daily; byday= MON, TUE, WED, THU, FRI;
    byhour=0; byminute=0; bysecond=0');
```
Changing the `GATHER_STATS_JOB` Schedule

Scheduler Windows

Following are the system windows that specify resource usage limits based on time-duration windows.

<table>
<thead>
<tr>
<th>Select Name</th>
<th>Resource Plan</th>
<th>Enabled</th>
<th>Next Open Date</th>
<th>End Date</th>
<th>Duration (min)</th>
<th>Active</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEEKNIGHT_WINDOW</td>
<td>TRUE</td>
<td>Dec 8, 2003 10:00:00 PM</td>
<td></td>
<td></td>
<td>480</td>
<td>FALSE</td>
<td>Weeknight window for maintenance task</td>
</tr>
<tr>
<td>WEEKEND_WINDOW</td>
<td>TRUE</td>
<td>Dec 13, 2003 12:00:00 AM</td>
<td></td>
<td></td>
<td>2560</td>
<td>FALSE</td>
<td>Weekend window for maintenance task</td>
</tr>
</tbody>
</table>
Statistics Collection Configuration

• DML monitoring
• Sampling
• Degree of parallelism
• Histograms
• Cascade
DML Monitoring

• The DML monitoring facility:
  – Tracks DML statements and truncation of tables
  – Is used by the Automatic Statistics Gathering mechanism for identifying segments with stale statistics
  – Is enabled by default when \texttt{STATISTICS\_LEVEL} is set to \texttt{TYPICAL} or \texttt{ALL}

• You can:
  – View the information on DML changes in the \texttt{USER\_TAB\_MODIFICATIONS} view
  – \texttt{Use DBMS\_STATS.FLUUSH\_DATABASE\_MONITORING\_INFO} to update the view with current information
  – \texttt{Use GATHER\_DATABASE\_STATS} or \texttt{GATHER\_SCHEMA\_STATS} for manual statistics gathering for tables with stale statistics when \texttt{OPTIONS} is set to \texttt{GATHER\_STALE} or \texttt{GATHER\_AUTO}
Sampling

- Statistics gathering relies on sampling to minimize resource usage.
- You can use the `ESTIMATE_PERCENT` argument of the `DBMS_STATS` procedures to change the sampling percentage to any value.
- Set to `DBMS_STATS.AUTO_SAMPLE_SIZE` (default) to maximize performance gains.
- `AUTO_SAMPLE_SIZE` enables the database to determine the appropriate sample size for each object automatically.

```sql
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS ('SH',DBMS_STATS.AUTO_SAMPLE_SIZE);
```
Degree of Parallelism

- Automatic Statistics Gathering operations can run either serially or in parallel.
- By default, the degree of parallelism is determined automatically.
- You can also manually specify the degree of parallelism using the `DEGREE` argument of the `DBMS_STATS` procedures.
- Setting the `DEGREE` parameter to `DBMS_STATS.AUTO_DEGREE` (default) enables the Oracle Database to choose an appropriate degree of parallelism even when collecting statistics manually.
Histories

- Influence optimizer decisions on selecting the optimal execution plan
- Provide improved selectivity estimates in the presence of data skew
- Enable optimal execution plans with nonuniform data distributions

<table>
<thead>
<tr>
<th>Column Value</th>
<th>Count of Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>1050</td>
</tr>
<tr>
<td>30</td>
<td>126</td>
</tr>
<tr>
<td>40</td>
<td>567</td>
</tr>
<tr>
<td>50</td>
<td>248</td>
</tr>
</tbody>
</table>

Number of buckets = 5

Bar chart showing the distribution of column values.
Creating Histograms

- The Automatic Statistics Gathering mechanism creates histograms as needed by default.
- You can use the `DBMS_STATS` package to change this default.
- You can use `DBMS_STATS` to create histograms manually.
- The following example shows how to create a histogram with 50 buckets on `PROD_LIST_PRICE`:

```sql
EXECUTE dbms_stats.gather_table_stats
('sh', 'products',
 method_opt => 'for columns size 50 prod_list_price');
```
Viewing Histogram Statistics

1. SELECT column_name, num_distinct, num_buckets, histogram
   FROM USER_TAB_COL_STATISTICS
   WHERE histogram <> 'NONE';

2. SELECT column_name, num_distinct, num_buckets, histogram
   FROM USER_TAB_COL_STATISTICS
   WHERE column_name = 'PROD_LIST_PRICE';
Histogram Tips

• The default option for `DBMS_STATS METHOD_OPTS` is `FOR ALL COLUMNS SIZE AUTO`, which enables automatic creation of histograms as needed.

• Alternatively, you can create histograms manually:
  – On skewed columns that are used frequently in `WHERE` clauses of queries
  – On columns that have a highly skewed data distribution
Histogram Tips

• Do not use histograms unless they substantially improve performance.
  – Histograms allocate additional storage.
  – Histograms, like all other optimizer statistics, are static.
  – Recompute the histogram when the data distribution of a column changes frequently.
  – For queries with bind variables
Bind Variable Peeking

- The optimizer peeks at the values of bind variables on the first invocation of a cursor.
- This is done to determine the selectivity of the predicate.
- Peeking does not occur for subsequent invocations of the cursor.
- Cursor is shared, based on the standard cursor-sharing criteria even for different bind values.
Cascading to Indexes

- The Automatic Statistics Gathering mechanism is configured by default to gather index statistics while gathering statistics on the parent tables.
- You can change the default behavior by modifying the `CASCADE` option of the `DBMS_STATS` package.
- Set the `CASCADE` option to:
  - `TRUE` to gather index statistics
  - `DBMS_STATS.AUTO_CASCADE` to have the Oracle Database determine whether index statistics are to be collected or not
Managing Statistics Collection: Example

dbms_stats.gather_table_stats
('sh' -- schema
,'customers' -- table
, null -- partition
, 20 -- sample size(%) 
, false -- block sample?
,'for all columns' -- column spec 
, 4 -- degree of parallelism 
,'default' -- granularity 
, true ); -- cascade to indexes

dbms_stats.set_param('CASCADE',
    'DBMS_STATS.AUTO.Cascade' );
dbms_stats.set_param('ESTIMATE_PERCENT','5');
dbms_stats.set_param('DEGREE','NULL');
When to Gather Manual Statistics

- Rely mostly on automatics statistics collection
- Change frequency of automatic statistics collection to meet your needs
- Gather statistics manually:
  - For objects that are volatile
  - For objects modified in batch operations
Statistics Gathering: Manual Approaches

- Dynamic sampling:

```sql
BEGIN
DBMS_STATS.DELETE_TABLE_STATS('OE', 'ORDERS');
DBMS_STATS.LOCK_TABLE_STATS('OE', 'ORDERS');
END;
```

- Manual statistics collection:

```sql
BEGIN
DBMS_STATS.GATHER_TABLE_STATS('OE', 'ORDERS');
DBMS_STATS.LOCK_TABLE_STATS('OE', 'ORDERS');
END;
```

- For objects modified in batch operations: gather statistics as part of the batch operation
- For new objects: gather statistics immediately after object creation
Dynamic Sampling

Dynamic sampling is used to automatically collect statistics when:

- The cost of collecting the statistics is minimal compared to the execution time
- The query is executed many times
Locking Statistics

• Prevents automatic gathering
• Is used primarily for volatile tables
  – Lock without statistics implies dynamic sampling.
  – Lock with statistics is for representative values.

EXECUTE DBMS_STATS.LOCK_TABLE_STATS
('owner name', 'table name');

EXECUTE DBMS_STATS.LOCK_SCHEMA_STATS
('owner name');

SELECT stattype_locked
FROM dba_tab_statistics;
Verifying Table Statistics

```sql
SELECT last_analyzed analyzed, sample_size, monitoring, table_name
FROM dba_tables
WHERE table_name = 'EMPLOYEES';
```

<table>
<thead>
<tr>
<th>ANALYZED</th>
<th>SAMPLE_SIZE</th>
<th>MON</th>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-FEB-04</td>
<td>2000</td>
<td>YES</td>
<td>EMPLOYEES</td>
</tr>
</tbody>
</table>
Verifying Column Statistics

```
SELECT column_name, num_distinct, histogram,
num_buckets, density, last_analyzed analyzed
FROM dba_tab_col_statistics
WHERE table_name  = 'SALES'
ORDER BY column_name;
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>NUM_DISTINCT</th>
<th>HISTOGRAM</th>
<th>NUM_BUCKETS</th>
<th>DENSITY</th>
<th>ANALYZED</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMOUNT_SOLD</td>
<td>3586</td>
<td>NONE</td>
<td>1</td>
<td>.000278862</td>
<td>09-FEB-04</td>
</tr>
<tr>
<td>CHANNEL_ID</td>
<td>4</td>
<td>NONE</td>
<td>1</td>
<td>.25</td>
<td>09-FEB-04</td>
</tr>
<tr>
<td>CUST_ID</td>
<td>7059</td>
<td>NONE</td>
<td>1</td>
<td>.000141663</td>
<td>09-FEB-04</td>
</tr>
<tr>
<td>PROD_ID</td>
<td>72</td>
<td>FREQUENCY</td>
<td>72</td>
<td>5.4416E-07</td>
<td>09-FEB-04</td>
</tr>
<tr>
<td>PROMO_ID</td>
<td>4</td>
<td>NONE</td>
<td>1</td>
<td>.25</td>
<td>09-FEB-04</td>
</tr>
<tr>
<td>QUANTITY_SOLD</td>
<td>1</td>
<td>NONE</td>
<td>1</td>
<td>1</td>
<td>09-FEB-04</td>
</tr>
<tr>
<td>TIME_ID</td>
<td>1460</td>
<td>NONE</td>
<td>1</td>
<td>.000684932</td>
<td>09-FEB-04</td>
</tr>
</tbody>
</table>

7 rows selected.
verifying index statistics

```sql
SELECT index_name name, num_rows n_r,
       last_analyzed l_a, distinct_keys
       d_k, leaf_blocks l_b,
       avg_leaf_blocks_per_key a_l,
       join_index j_I
FROM dba_indexes
WHERE table_name = 'EMPLOYEES'
ORDER BY index_name;
```
History of Optimizer Statistics

GATHER_STATS  IMPORT_STATS  SET_STATS

RESTORE_TABLE_STATS

New statistics

DBA_OPTSTATS_OPERATIONS

Operation date

Old statistics

31 days

DBA_TAB_STATS_HISTORY
Managing Historical Optimizer Statistics

- `RESTORE_*_STATS()`
- `PURGE_STATS()`
- `ALTER_STATS_HISTROY_RETENTION()`
Generating System Statistics

- I/O
- CPU

```
BEGIN
dbms_stats.gather_system_stats(
    gathering_mode => 'interval',
    interval => 720,
    stattab => 'mystats',
    statid => 'oltp');
END;
/
```
Statistics on Dictionary Objects

- **GATHER_FIXED_OBJECTS_STATS**
- **Dictionary tables**
- **GATHER_DATABASE_STATS**
- **GATHER_DICTIONARY_STATS**

Fixed tables
Dictionary Statistics: Best Practices

GATHER_SCHEMA_STATS('SYS')
GATHER_DICTIONARY_STATS

GATHER_FIXED_OBJECTS_STATS

GATHER_DATABASE_STATS(OPTIONS=>'GATHER AUTO')

CREATE
ALTER
DROP...

Workload

DDLs
Summary

In this lesson, you should have learned how to:

- Use the Automatic Statistics Gathering mechanism
- Use the `DBMS_STATS` package for manual statistics gathering
- Determine selectivity for predicates with and without bind variables
Practice 7: Overview

This practice covers the following topics:

• Using `DBMS_STATS` to gather manual statistics
• Verifying the existence of the `gather_stats_job`
• Understanding the use of histograms
• Understanding bind variable peeking
Application Tracing
Objectives

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

• Configure the SQL Trace facility to collect session statistics
• Enable SQL Trace and locate your trace files
• Format trace files using the TKPROF utility
• Interpret the output of the TKPROF command
Overview of Application Tracing

• End to End Application Tracing
  – Enterprise Manager
  – DBMS_MONITOR
• trcsess utility
• SQL Trace and TKPROF
End to End Application Tracing

• Simplifies the process of diagnosing performance problems in multitier environments

• Can be used to
  – Identify high-load SQL
  – Monitor what a user's session is doing at the database level

• Simplifies management of application workloads by tracking specific modules and actions in a service
End to End Application Tracing Using EM

**Top Consumers**

**Overview**
- Top Services
- Top Modules
- Top Actions
- Top Clients
- Top Sessions

**Top Services**
- 100% SYS$USERS (100%)

**Top Modules (by Service)**
- 50% OEM_DefaultPool (SYS$USERS)
- 17% OEM_SystemPool (SYS$USERS)
- 17% Unnamed (SYS$USERS)
- 17% Admin_Connection (SYS$USERS)

**Top Clients**
- 33% SYSTEM@148.87.19.50@Mozilla/4.0 (compatible: MSIE 6.0; Windows (66.7%)
- 67% Unnamed (33.3%)

**Top Actions (by Module) (by Service)**
- 50% /database/instance/sitemap (OEM_DefaultPool (SYS$USERS)
- 17% / (SYS$USERS)
- 17% Unnamed (OEM_SystemPool) (SYS$USERS)
- 17% Unnamed (Admin_Connection) (SYS$USERS)
Using **DBMS_MONITOR**

```sql
EXECUTE DBMS_MONITOR.CLIENT_ID_STAT_ENABLE(
    client_id => 'OE.OE',
    waits => TRUE, binds => FALSE);

EXECUTE DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE(
    service_name =>'ACCTG',
    module_name => 'PAYROLL');

EXECUTE DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE(
    service_name =>'ACCTG',
    module_name => 'GLEDGER',
    action_name => 'INSERT ITEM');
```
Viewing Gathered Statistics for End to End Application Tracing

- The accumulated statistics for a specified service can be displayed in the `V$SERVICE_STATS` view.
- The accumulated statistics for a combination of specified service, module, and action can be displayed in the `V$SERV_MOD_ACT_STATS` view.
- The accumulated statistics for elapsed time of database calls and for CPU use can be displayed in the `V$SVCMETRIC` view.
- All outstanding traces can be displayed in an Oracle Enterprise Manager report or with the `DBA_ENABLED_TRACES` view.
trcsess Utility

```sql
SQL> select sid||'.'||serial#, username
    2 from v$session
    3 where username in ('HR', 'SH');
```

| SID||'..'||SERIAL# | USERNAME |
|-------------------|--------------------------|
| 236.57             | HR                       |
| 245.49845          | SH                       |

$ trcsess session= 236.57 orcl_ora_11155.trc
  output=x.txt
SQL Trace Facility

- Usually enabled at the session level
- Gathers session statistics for SQL statements grouped by session
- Produces output that can be formatted by **TKPROF**
Information Captured by SQL Trace

- Parse, execute, and fetch counts
- CPU and elapsed times
- Physical reads and logical reads
- Number of rows processed
- Misses on the library cache
- Username under which each parse occurred
- Each commit and rollback
How to Use the SQL Trace Facility

1. Set the initialization parameters.
2. Enable tracing.
3. Run the application.
4. Disable Trace
5. Close the session.
6. Format the trace file.
7. Interpret the output.
Initialization Parameters

TIMED_STATISTICS = {false|true}
MAX_DUMP_FILE_SIZE = {n|unlimited}
USER_DUMP_DEST = directory_path
STATISTICS_LEVEL = {BASIC|TYPICAL|ALL}
Enabling SQL Trace

- For your current session:

  SQL> ALTER SESSION SET sql_trace = true;

- For any session:

  SQL> EXECUTE dbms_session.set_sql_trace(true);

  SQL> EXECUTE dbms_system.set_sql_trace_in_session 2 (session_id, serial_id, true);

- For an instance, set the following parameter:

  SQL_TRACE = TRUE
Formatting Your Trace Files

**OS> tkprof tracefile outputfile [options]**

**TKPROF command examples:**

OS> tkprof
OS> tkprof ora_902.trc run1.txt
OS> tkprof ora_902.trc run2.txt sys=no
    sort=execpu print=3
TKPROF Command Options

SORT = option
PRINT = n
EXPLAIN = username/password
INSERT = filename
SYS = NO
AGGREGATE = NO
RECORD = filename
TABLE = schema.tablename
Output of the **TKPROF** Command

- Text of the SQL statement
- Trace statistics (for statement and recursive calls) separated into three SQL processing steps:

<table>
<thead>
<tr>
<th>PROCESS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARSE</strong></td>
<td>Translates the SQL statement into an execution plan</td>
</tr>
</tbody>
</table>
| **EXECUTE** | Executes the statement  
(This step modifies the data for `INSERT`, `UPDATE`, and `DELETE` statements.) |
| **FETCH** | Retrieves the rows returned by a query  
(Fetches are performed only for `SELECT` statements.) |
Output of the TKPROF Command

There are seven categories of trace statistics:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>Number of times the procedure was executed</td>
</tr>
<tr>
<td>CPU</td>
<td>Number of seconds to process</td>
</tr>
<tr>
<td>Elapsed</td>
<td>Total number of seconds to execute</td>
</tr>
<tr>
<td>Disk</td>
<td>Number of physical blocks read</td>
</tr>
<tr>
<td>Query</td>
<td>Number of logical buffers read for consistent read</td>
</tr>
<tr>
<td>Current</td>
<td>Number of logical buffers read in current mode</td>
</tr>
<tr>
<td>Rows</td>
<td>Number of rows processed by the fetch or execute</td>
</tr>
</tbody>
</table>
Output of the **TKPROF** Command

The **TKPROF** output also includes the following:

- Recursive SQL statements
- Library cache misses
- Parsing user ID
- Execution plan
- Optimizer mode or hint
- Row source operation

... 

Misses in library cache during parse: 1  
Optimizer mode: ALL_ROWS  
Parsing user id: 61  

<table>
<thead>
<tr>
<th>Rows</th>
<th>Row Source Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>TABLE ACCESS BY INDEX ROWID EMPLOYEES (cr=9 pr=0 pw=0 time=129 us)</td>
</tr>
<tr>
<td>24</td>
<td>INDEX RANGE SCAN SAL_IDX (cr=3 pr=0 pw=0 time=1554 us) (object id ...</td>
</tr>
</tbody>
</table>
**TKPROF Output with No Index: Example**

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

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.02</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>0.10</td>
<td>0.09</td>
<td>1408</td>
<td>1459</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>0.12</td>
<td>0.11</td>
<td>1408</td>
<td>1459</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 61

Rows Row Source Operation
---------- ---------------------------------------------------
1     SORT AGGREGATE (cr=1459 pr=1408 pw=0 time=93463 us)  
77    TABLE ACCESS FULL CUSTOMERS (cr=1459 pr=1408 pw=0 time=31483 us)
TKPROF Output with Index: Example

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

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.01</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>0.01</td>
<td>0.00</td>
<td>0</td>
<td>77</td>
<td>0</td>
</tr>
</tbody>
</table>

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 61

Rows Row Source Operation
--------- ---------------------------------------------------
1  SORT AGGREGATE (cr=77 pr=0 pw=0 time=732 us)
77  TABLE ACCESS BY INDEX ROWID CUSTOMERS (cr=77 pr=0 pw=0 time=1760 us)
77  INDEX RANGE SCAN CUST_CUST_CITY_IDX (cr=2 pr=0 pw=0 time=100 us)(object id 55097)
Summary

In this lesson, you should have learned how to:

- Set SQL Trace initialization parameters
  - SQL_TRACE, TIMED_STATISTICS
  - USER_DUMP_DEST, MAX_DUMP_FILE_SIZE
- Enable SQL Trace for a session
  
  ```sql
  ALTER SESSION set sql_trace = true
  dbms_session.set_sql_trace(...)
  dbms_system.set_sql_trace_in_session(...)
  
  • Format trace files with TKPROF
  • Interpret the output
Practice 8: Overview

This practice covers the following topics:

- **Using** `TKPROF`
- **Using** `DBMS_MONITOR`
Identifying High-Load SQL
Objectives

After completing this lesson, you should understand the different methods of identifying high-load SQL:

- ADDM
- Top SQL
- Dynamic performance views
- Statspack
SQL Tuning Process: Overview

1. Identify high-load SQL
2. Analyze SQL
3. Take corrective action
Identifying High-Load SQL

- SQL workload
- Automatic (ADDM)
- Manual (Top SQL)
- High-load SQL
Automatic Database Diagnostic Monitor

SGA
In-memory statistics

MMON

60 minutes

Snapshots

EM
ADDM results

ADDM

ADDM results

AWR
ADDM Output

EM

Findings with impact

Recommendations

ADDM

Invoke advisors

ADDM results

AWR
Manual Identification: Top SQL
Spot SQL

Spot SQL shows all the SQL statements that have been active in a recent 5 minute interval.

**Spot Interval Selection**

Drag the shaded box to select the 5 minute interval for which you want to view details in the section below. Use the active sessions data to help with your selection.

**Detail for Selected 5 minute Interval**

Start Time: **Aug 12, 2004 5:59:03 AM**

**All SQL**

<table>
<thead>
<tr>
<th>Select SQL ID</th>
<th>SQL Type</th>
<th>Activity (%)</th>
<th>CPU (%)</th>
<th>Wait (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6yvch1xu9ca3g</td>
<td>PL/SQL EXECUTE</td>
<td>60.00</td>
<td>60.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2z064ybzkwflx</td>
<td>PL/SQL EXECUTE</td>
<td>60.00</td>
<td>60.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Top SQL (ordered by Activity)**

50%  50%
Manual Identification: Statspack

Statspack:

- Collects data about high-load SQL
- Precalculates useful data
  - Cache hit ratios
  - Transaction statistics
- Uses permanent tables owned by the `PERFSTAT` user to store performance statistics
- Separates data collection from report generation
- Uses snapshots to compare performance at different times
Using Dynamic Performance Views

- Select a slow performing period of time to identify high-load SQL.
- Gather operating system and Oracle statistics
- Identify the SQL statements that use the most resources.
### V$SQLAREA View

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_TEXT</td>
<td>First thousand characters of the SQL text</td>
</tr>
<tr>
<td>SORTS</td>
<td>Sum of the number of sorts that were done for all the child cursors</td>
</tr>
<tr>
<td>EXECUTIONS</td>
<td>Total number of executions, totaled over all the child cursors</td>
</tr>
<tr>
<td>DISK_READS</td>
<td>Sum of the number of disk reads over all child cursors</td>
</tr>
<tr>
<td>CPU_TIME</td>
<td>CPU time (in microseconds) used by this cursor for parsing/executing/fetching</td>
</tr>
<tr>
<td>ELAPSED_TIME</td>
<td>Elapsed time (in microseconds) used by this cursor for parsing, executing, and fetching</td>
</tr>
</tbody>
</table>
Querying the V$SQLAREA View

```
SELECT sql_text, disk_reads, sorts, cpu_time, elapsed_time
FROM v$sqlarea
WHERE upper(sql_text) like '%PROMOTIONS%'
ORDER BY sql_text;
```
### Investigating Full Table Scan Operations

```sql
SELECT name, value FROM v$sysstat
WHERE name LIKE '%table scan%';
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>table scans (short tables)</td>
<td>217842</td>
</tr>
<tr>
<td>table scans (long tables)</td>
<td>3040</td>
</tr>
<tr>
<td>table scans (rowid ranges)</td>
<td>254</td>
</tr>
<tr>
<td>table scans (cache partitions)</td>
<td>7</td>
</tr>
<tr>
<td>table scans (direct read)</td>
<td>213</td>
</tr>
<tr>
<td>table scan rows gotten</td>
<td>40068909</td>
</tr>
<tr>
<td>table scan blocks gotten</td>
<td>1128681</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned about the different methods of identifying high-load SQL:

• ADDM
• Top SQL
• Statspack
• Dynamic performance views
Automatic SQL Tuning
Objectives

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

• Describe automatic SQL tuning
• Describe the Automatic Workload Repository
• Use Automatic Database Diagnostic Monitor
• View the cursor cache
• Perform automatic SQL tuning
• Use the SQL Tuning Advisor
• Use the SQL Access Advisor
SQL Tuning Process: Overview

1. Identify high-load SQL
2. Analyze SQL
3. Take corrective action
Automatic SQL Tuning

How can I tune high-load SQL?

SQL workload

ADDM

High-load SQL

DBA
Automatic Tuning Optimizer

- Is the query optimizer running in tuning mode
- Performs verification steps
- Performs exploratory steps
SQL Tuning Advisor

SQL workload

ADDM

High-load SQL

SQL Tuning Advisor
SQL Tuning Advisor Analysis

Automatic Tuning Optimizer

Statistics analysis
SQL profiling
Access path analysis
SQL structure analysis

Comprehensive SQL tuning

Detect stale or missing statistics
Perform SQL profiling
Add missing index
Run Access Advisor
Restructure SQL

SQL Tuning Advisor
Statistics Analysis

Automatic Statistics Gathering disabled

Stale or missing statistics

SQL Tuning Advisor

DBMS_STATS.GATHER_TABLE_STATS(
    ownname=>'SH', tabname=>'CUSTOMERS',
    estimate_percent=>
    DBMS_STATS.AUTO_SAMPLE_SIZE);
SQL Profiling

SQL Tuning Advisor

Submit

Optimizer (Tuning mode)

Create

SQL Profile

Use

Database users

No application code change

Optimizer (Normal mode)

Output

Well-tuned plan
SQL Access Path Analysis

SQL Tuning Advisor

Indexes

Significant performance gain

SQL statements
How can I rewrite it?

- Poorly written SQL statement
- Type mismatch and indexes
- Design mistakes

SQL Structure Analysis

Restructured SQL statement

SQL constructs

SQL Tuning Advisor
SQL Tuning Set

Sources
- AWR
- Cursor cache
- Custom

Manual selection
- Filter/rank
- STS

DBA

SQL Tuning Advisor
SQL Tuning Views

• Advisor information views:
  – DBA_ADVISOR_TASKS
  – DBA_ADVISOR_FINDINGS
  – DBA_ADVISOR_RECOMMENDATIONS
  – DBA_ADVISOR_RATIONALE

• SQL tuning information views:
  – DBA_SQLTUNE_STATISTICS
  – DBA_SQLTUNE_BINDS
  – DBA_SQLTUNE_PLANS

• SQL Tuning Set views:
  – DBA_SQLSET, DBA_SQLSET_BINDS
  – DBA_SQLSET_STATEMENTS
  – DBA_SQLSET_REFERENCES

• SQL Profile view: DBA_SQL_PROFILES
Enterprise Manager: Usage Model

- Launch SQL Tuning Advisor
- ADDM report page
- Top SQL page
- SQL Tuning Set page
- View recommendations
- Implement recommendations
SQL Access Advisor

SQL workload

Select ...
Select ...
Select ...

...  

SQL Access Advisor

Recommendations

Create index ...
Drop index...
Create materialized view ...
SQL Access Advisor: Features

Using the SQL Access Advisor Wizard or API, you can do the following:

- Recommend creation or removal of materialized views and indexes
- Manage workloads
- Mark, update, and remove recommendations
SQL Access Advisor: Usage Model

Cursor cache
User-defined
Hypothetical
STS

Filter options
Workload
SQL Access Advisor
SQL Access Advisor: User Interface
## SQL Tuning Advisor and SQL Access Advisor

<table>
<thead>
<tr>
<th>Analysis Types</th>
<th>Advisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistics</td>
<td>SQL Tuning Advisor</td>
</tr>
<tr>
<td>SQL Profile</td>
<td>SQL Tuning Advisor</td>
</tr>
<tr>
<td>SQL Structure</td>
<td>SQL Tuning Advisor</td>
</tr>
<tr>
<td>Access Path: Indexes</td>
<td>SQL Tuning/Access Advisor</td>
</tr>
<tr>
<td>Access Path: Materialized Views</td>
<td>SQL Access Advisor</td>
</tr>
<tr>
<td>Access Path: Materialized View Logs</td>
<td>SQL Access Advisor</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned how to:

- Describe the Automatic Workload Repository
- Use Automatic Database Diagnostic Monitor
- View the cursor cache
- Perform automatic SQL tuning
- Use the SQL Tuning Advisor
- Use the SQL Access Advisor
Objectives

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

• Identify index types
• Identify basic access methods
• Monitor index usage
Indexing Guidelines

• You should create indexes only as needed.
• Creating an index to tune a specific statement could affect other statements.
• It is best to drop unused indexes.
• `EXPLAIN PLAN` can be used to determine if an index is being used by the optimizer.
Types of Indexes

- Unique and nonunique indexes
- Composite indexes
- Index storage techniques:
  - B*-tree
    - Normal
    - Reverse key
    - Descending
    - Function based
  - Bitmap
  - Domain indexes
  - Key compression
## When to Index

<table>
<thead>
<tr>
<th>Index</th>
<th>Do Not Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keys frequently used in search or query expressions</td>
<td>Keys and expressions with few distinct values except bitmap indexes in data warehousing</td>
</tr>
<tr>
<td>Keys used to join tables</td>
<td>Frequently updated columns</td>
</tr>
<tr>
<td>High-selectivity keys</td>
<td>Columns used only with functions or expressions unless creating function-based indexes</td>
</tr>
<tr>
<td>Foreign keys</td>
<td>Columns based only on query performance</td>
</tr>
</tbody>
</table>
Effect of DML Operations on Indexes

- Inserts result in the insertion of an index entry in the appropriate block. (Block splits might occur.)
- Delete rows result in a deletion of the index entry. (Empty blocks become available.)
- Updates to the key columns result in a logical delete and an insert to the index.
Indexes and Constraints

The Oracle Server implicitly creates or uses B*-tree indexes when you define the following:

- Primary key constraints
- Unique key constraints

```sql
CREATE TABLE new_channels
  ( channel_id CHAR(1)
    CONSTRAINT channels_channel_id_pk PRIMARY KEY,
    channel_desc VARCHAR2(20)
    CONSTRAINT channels_channel_desc_nn NOT NULL,
    channel_class VARCHAR2(20),
    channel_total VARCHAR2(13)
  );
```
Indexes and Foreign Keys

- Indexes are not created automatically.
- There are locking implications to DML activity on parent-child tables.
Basic Access Methods

- **Full table scans:**
  - Can use multiblock I/O
  - Can be parallelized

- **Index scans:**
  - Allow index access only
  - Are followed by access by **ROWID**

- **Fast-full index scans:**
  - Can use multiblock I/O
  - Can be parallelized
Identifying Unused Indexes

- The Oracle Database provides the capability to gather statistics about the usage of an index.
- Benefits include:
  - Space conservation
  - Improved performance by eliminating unnecessary overhead during DML operations
Enabling and Disabling the Monitoring of Index Usage

• To start monitoring the usage of an index:
  ```sql
  ALTER INDEX customers_pk MONITORING USAGE;
  ```

• To stop monitoring the usage of an index:
  ```sql
  ALTER INDEX customers_pk NOMONITORING USAGE;
  ```

• `V$OBJECT_USAGE` contains information about the usage of an index.
Index Tuning Using the SQL Access Advisor

The SQL Access Advisor:

- Determines which indexes are required
- Recommends a set of indexes
- Is invoked from
  - Advisor Central in Oracle Enterprise Manager
  - Run through the DBMS_ADVISOR package APIs
- Uses a workload such as:
  - Current contents of the SQL cache
  - A user-defined set of SQL statements
  - A SQL Tuning Set
  - Hypothetical workload
- Generates a set of recommendations
- Provides an implementation script
Summary

In this lesson, you should have learned about the following:

• **Indexes**
  – Index types
  – DML operations and indexes
  – Indexes and constraints

• **Monitoring indexes**
  – Index usage monitoring
Using Different Indexes
Objectives

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

- Use composite indexes
- Use bitmap indexes
- Use bitmap join indexes
- Identify bitmap index operations
- Create function-based indexes
- Use index-organized tables
Composite Indexes

Here are some features of the index displayed below.

- **Combinations of columns that are leading portions of the index:**
  - cust_last_name
  - cust_last_name cust_first_name
  - cust_last_name cust_first_name cust_gender

- **Combinations of columns that are *not* leading portions of the index:**
  - cust_first_name cust_gender
  - cust_first_name
  - cust_gender

```
CREATE INDEX cust_last_first_gender_idx
ON customers (cust_last_name,
              cust_first_name, cust_gender);
```
Composite Index Guidelines

- Create a composite index on keys that are used together frequently in `WHERE` clauses.
- Create the index so that the keys used in `WHERE` clauses make up a leading portion.
- Put the most frequently queried column in the leading part of the index.
- Put the most restrictive column in the leading part of the index.
Skip Scanning of Indexes

• Index skip scanning enables access through a composite index when the prefix column is not part of the predicate.

• Skip scanning is supported by:
  – Cluster indexes
  – Descending scans
  – CONNECT BY clauses

• Skip scanning is not supported by reverse key or bitmap indexes.
Bitmap Index

- Compared with regular B*-tree indexes, bitmap indexes are faster and use less space for low-cardinality columns.
- Each bitmap index comprises storage pieces called *bitmaps*.
- Each bitmap contains information about a particular value for each of the indexed columns.
- Bitmaps are compressed and stored in a B*-tree structure.
When to Use Bitmap Indexes

Use bitmap indexes for:

- Columns with low cardinality
- Columns that are frequently used in:
  - Complex WHERE clause conditions
  - Group functions (such as COUNT and SUM)
- Very large tables
- DSS systems with many ad hoc queries and few concurrent DML changes
Advantages of Bitmap Indexes

When used appropriately, bitmap indexes provide:

• Reduced response time for many ad hoc queries
• Substantial reduction of space usage compared to other indexing techniques
• Dramatic performance gains
Bitmap Index Guidelines

• Reduce bitmap storage by:
  – Declaring columns NOT NULL when possible
  – Using fixed-length data types when feasible
  – Using the command:
    ALTER TABLE ... MINIMIZE RECORDS_PER_BLOCK

• Improve bitmap performance by increasing the value of PGA_AGGREGATE_TARGET.
CREATE BITMAP INDEX cust_sales_bji
ON sales(c.cust_city)
FROM sales s, customers c
WHERE c.cust_id = s.cust_id;
Bitmap Join Index

- No join with the CUSTOMERS table is needed.
- Only the index and the SALES table are used to evaluate the following query:

```
SELECT SUM(s.amount_sold)
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
AND    c.cust_city = 'Sully';
```
Bitmap Join Index: Advantages and Disadvantages

• Advantages
  – Good performance for join queries; space-efficient
  – Especially useful for large-dimension tables in star schemas

• Disadvantages
  – More indexes are required: Up to one index per dimension-table column rather than one index per dimension table is required.
  – Maintenance costs are higher: Building or refreshing a bitmap join index requires a join.
Function-Based Index

CREATE INDEX FBI_UPPER_LASTNAME
  ON CUSTOMERS(upper(cust_last_name));

ALTER SESSION
  SET QUERY_REWRITE_ENABLED = TRUE;

SELECT *
FROM customers
WHERE UPPER(cust_last_name) = 'SMITH';
Function-Based Indexes: Usage

Function-based indexes:
• Materialize computational-intensive expressions
• Facilitate non-case-sensitive searches
• Provide a simple form of data compression
• Can be used for an NLS sort index
Indexed access on table

Accessing index-organized table

ROWID

Non-key columns

Key column

Row header
Index-Organized Tables: Characteristics

Index-organized tables:
- Must have a primary key
- Cannot contain LONG columns
- Can be rebuilt
- Can be accessed by either primary key or leading columns
Advantages and Disadvantages of IOTs

- **Advantages**
  - IOTs provide fast key-based access for queries involving exact match and range searches.
  - DML causes only updates to index structure.
  - Storage requirements are reduced.
  - IOTs are useful in:
    - Applications that retrieve data based on a primary key
    - Applications that involve content-based information

- **Disadvantages**
  - Not suitable for queries that do not use the primary key in a predicate
Summary

In this lesson, you should have learned about:

• Composite indexes
• Bitmap indexes
• Bitmap join indexes
• Function-based indexes
• Index-organized tables
Optimizer Hints
Objectives

After completing this lesson, you should be able to specify hints for:

• Optimizer mode
• Query transformation
• Access path
• Join orders
• Join methods
Optimizer Hints: Overview

Optimizer hints:
- Are used to alter execution plans
- Influence optimizer decisions
- Provide a mechanism to instruct the optimizer to choose a certain query execution plan
# Types of Hints

<table>
<thead>
<tr>
<th>Type of Hint</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-table hints</td>
<td>Specified on one table or view</td>
</tr>
<tr>
<td>Multitable hints</td>
<td>Specify more than one table or view</td>
</tr>
<tr>
<td>Query block hints</td>
<td>Operate on a single query block</td>
</tr>
<tr>
<td>Statement hints</td>
<td>Apply to the entire SQL statement</td>
</tr>
</tbody>
</table>
Specifying Hints

```
/*+ hint comment text */
```
Rules for Hints

- Place hints immediately after the first SQL keyword of a statement block.
- Each statement block can have only one hint comment, but it can contain multiple hints.
- Hints apply to only the statement block in which they appear.
- If a statement uses aliases, hints must reference aliases rather than table names.
Hint Recommendations

• Use hints carefully because they imply a high maintenance load.
• Be aware of the performance impact of hard-coded hints when they become less valid.
Optimizer Hint Syntax: Example

```
UPDATE /*+ INDEX(p PRODUCTS_PROD_CAT_IX)*/
products p
SET   p.prod_min_price =
     (SELECT
      (pr.prod_list_price*.95)
     FROM products pr
     WHERE p.prod_id = pr.prod_id)
WHERE p.prod_category = 'Men'
AND   p.prod_status = 'available, on stock'
/```

Hint Categories

There are hints for:
• Optimization approaches and goals
• Access paths
• Query transformations
• Join orders
• Join operation
• Parallel execution
### Optimization Goals and Approaches

<table>
<thead>
<tr>
<th><strong>ALL_ROWS</strong></th>
<th>Chooses cost-based approach with a goal of best throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIRST_ROWS(n)</strong></td>
<td>Instructs Oracle Server to optimize an individual SQL statement for fast response</td>
</tr>
</tbody>
</table>
## Hints for Access Paths

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL</td>
<td>Performs a full table scan</td>
</tr>
<tr>
<td>ROWID</td>
<td>Accesses a table by ROWID</td>
</tr>
<tr>
<td>INDEX</td>
<td>Scans an index in ascending order</td>
</tr>
<tr>
<td>INDEX_ASC</td>
<td>Scans an index in ascending order</td>
</tr>
<tr>
<td>INDEX_COMBINE</td>
<td>Explicitly chooses a bitmap access path</td>
</tr>
</tbody>
</table>
## Hints for Access Paths

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX_JOIN</td>
<td>Instructs the optimizer to use an index join as an access path</td>
</tr>
<tr>
<td>INDEX_DESC</td>
<td>Chooses an index scan for the specified table</td>
</tr>
<tr>
<td>INDEX_FFS</td>
<td>Performs a fast-full index scan</td>
</tr>
<tr>
<td>NO_INDEX</td>
<td>Disallows using a set of indexes</td>
</tr>
<tr>
<td>AND_EQUAL</td>
<td>Merges single-column indexes</td>
</tr>
</tbody>
</table>
INDEX_COMBINE Hint: Example

```
SELECT --+INDEX_COMBINE(CUSTOMERS)
    cust_last_name
FROM   SH.CUSTOMERS
WHERE  ( CUST_GENDER= 'F' AND
        CUST_MARITAL_STATUS = 'single')
OR     CUST_YEAR_OF_BIRTH BETWEEN '1917'
       AND '1920';
```
INDEX_COMBINE Hint: Example

Execution Plan

0  SELECT STATEMENT Optimizer=CHOOSE (Cost=491
   Card=10481
   Bytes =167696)
1  0   TABLE ACCESS (BY INDEX ROWID) OF 'CUSTOMERS'
     (Cost=491 ...)
2  1     BITMAP CONVERSION (TO ROWIDS)
3  2       BITMAP OR
4  3           BITMAP AND
5  4             BITMAP INDEX (SINGLE VALUE) OF
                 'CUST_MARITAL_BIX'
6  4             BITMAP INDEX (SINGLE VALUE) OF
                 'CUST_GENDER_BIX'
7  3     BITMAP MERGE
8  7     BITMAP INDEX (RANGE SCAN) OF
         'CUST_YOB_BIX'
### Hints for Query Transformation

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE_CONCAT</td>
<td>Rewrites <strong>OR</strong> into <strong>UNION ALL</strong> and disables <strong>INLIST</strong> processing</td>
</tr>
<tr>
<td>NO_EXPAND</td>
<td>Prevents <strong>OR</strong> expansions</td>
</tr>
</tbody>
</table>
## Hints for Query Transformation

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MERGE</td>
<td>Merges a view for each query</td>
</tr>
<tr>
<td>NO_MERGE</td>
<td>Prevents merging of mergeable views</td>
</tr>
<tr>
<td>STAR_TRANSFORMATION</td>
<td>Makes the optimizer use the best plan in which the transformation can be used</td>
</tr>
<tr>
<td>FACT</td>
<td>Indicates that the hinted table should be considered as a fact table</td>
</tr>
<tr>
<td>NO_FACT</td>
<td>Indicates that the hinted table should not be considered as a fact table</td>
</tr>
</tbody>
</table>
## Hints for Join Orders

<table>
<thead>
<tr>
<th>HINT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERED</td>
<td>Causes the Oracle Server to join tables in the order in which they appear in the <code>FROM</code> clause</td>
</tr>
<tr>
<td>LEADING</td>
<td>Uses the specified table as the first table in the join order</td>
</tr>
</tbody>
</table>

## Hints for Join Operations

<table>
<thead>
<tr>
<th>Hints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USE_NL</strong></td>
<td>Joins the specified table using a nested loop join</td>
</tr>
<tr>
<td><strong>NO_USE_NL</strong></td>
<td>Does not use nested loops to perform the join</td>
</tr>
<tr>
<td><strong>USE_MERGE</strong></td>
<td>Joins the specified table using a sort-merge join</td>
</tr>
<tr>
<td><strong>NO_USE_MERGE</strong></td>
<td>Does not perform sort-merge operations for the join</td>
</tr>
<tr>
<td><strong>USE_HASH</strong></td>
<td>Joins the specified table using a hash join</td>
</tr>
<tr>
<td><strong>NO_USE_HASH</strong></td>
<td>Does not use hash join</td>
</tr>
</tbody>
</table>
### Other Hints

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND</td>
<td>Enables direct-path INSERT</td>
</tr>
<tr>
<td>NOAPPEND</td>
<td>Enable regular INSERT</td>
</tr>
<tr>
<td>ORDERED_PREDICATES</td>
<td>Forces the optimizer to preserve the order of predicate evaluation</td>
</tr>
<tr>
<td>CURSOR_SHARING_EXACT</td>
<td>Prevents replacing literals with bind variables</td>
</tr>
<tr>
<td>DYNAMIC_SAMPLING</td>
<td>Controls dynamic sampling to improve server performance</td>
</tr>
</tbody>
</table>
# Hints for Suppressing Index Usage

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_INDEX</td>
<td>Disallows use of any indexes</td>
</tr>
<tr>
<td>FULL</td>
<td>Forces a full table scan</td>
</tr>
<tr>
<td>INDEX or INDEX_COMBINE</td>
<td>Forces the optimizer to use a specific index or a set of listed indexes</td>
</tr>
</tbody>
</table>
Hints and Views

- Do not use hints in views.
- Use view-optimization techniques:
  - Statement transformation
  - Results accessed like a table
- Hints can be used on mergeable views and nonmergeable views.
# Hints for View Processing

<table>
<thead>
<tr>
<th>HINT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MERGE</td>
<td>Merges complex views or subqueries with the surrounding query</td>
</tr>
<tr>
<td>NO_MERGE</td>
<td>Does not merge mergeable views</td>
</tr>
</tbody>
</table>
Global and Local Hints

- Extended hint syntax enables the specifying of (global) hints through views
- References a table name in the hint with a dot notation

```sql
CREATE view city_view AS
SELECT *
FROM customers c
WHERE cust_city like 'S%';

SELECT /*+ index(v.c cust_credit_limit_idx) */
  v.cust_last_name, v.cust_credit_limit
FROM city_view v
WHERE cust_credit_limit > 5000;
```
Specifying a Query Block in a Hint

1. Explain plan for
   
   ```sql
   SELECT employee_id, last_name
   FROM hr.employees e
   WHERE last_name = 'Smith';
   ```

2. ```sql
   SELECT /*+ QB_NAME(qb) FULL(@qb e) */
   employee_id, last_name
   FROM hr.employees e
   WHERE employee_id = 100;
   ```
Specifying a Full Set of Hints

```
SELECT /*+ LEADING(e2 e1) USE_NL(e1) INDEX(e1 emp_emp_id_pk) USE_MERGE(j) FULL(j) */
  e1.first_name, e1.last_name, j.job_id,
  sum(e2.salary) total_sal
FROM hr.employees e1, hr.employees e2,
 hr.job_history j
WHERE e1.employee_id = e2.manager_id
AND e1.employee_id = j.employee_id
AND e1.hire_date = j.start_date
GROUP BY e1.first_name, e1.last_name, j.job_id
ORDER BY total_sal;
```
Summary

In this lesson, you should have learned how to:

- Set the optimizer mode
- Use optimizer hint syntax
- Determine access-path hints
- Analyze hints and their impact on views
Materialized Views
Objectives

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

- Identify the characteristics and benefits of materialized views
- Use materialized views to enable query rewrites
- Verify the properties of materialized views
- Perform refreshes on materialized views
Materialized Views

A materialized view:

• Is a precomputed set of results
• Has its own data segment and offers:
  – Space management options
  – Use of its own indexes
• Is useful for:
  – Expensive and complex joins
  – Summary and aggregate data
If Materialized Views Are Not Used

```sql
SELECT c.cust_id, SUM(amount_sold) 
FROM   sales s, customers c 
WHERE  s.cust_id = c.cust_id 
GROUP BY c.cust_id;
```

```sql
CREATE TABLE cust_sales_sum AS 
SELECT c.cust_id, SUM(amount_sold) AS amount 
FROM   sales s, customers c 
WHERE  s.cust_id = c.cust_id 
GROUP BY c.cust_id;
```

```sql
SELECT * FROM cust_sales_sum;
```
Benefits of Using Materialized Views

```
CREATE MATERIALIZED VIEW cust_sales_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_id, SUM(amount_sold) AS amount
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
SELECT c.cust_id, SUM(amount_sold)
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

Execution Plan
```
0  SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 ...)
1  0  MAT_VIEW REWRITE ACCESS (FULL) OF 'CUST_SALES_MV' (MAT_VIEW REWRITE) (Cost=6 ...)
```
How Many Materialized Views?

• One materialized view for multiple queries:
  – One materialized view can be used to satisfy multiple queries
  – Less disk space is needed
  – Less time is needed for maintenance

• Query rewrite chooses the materialized view to use.

• One materialized view per query:
  – Is not recommended because it consumes too much disk space
  – Improves one query's performance
Creating Materialized Views: Syntax Options

CREATE MATERIALIZED VIEW mview_name
   [TABLESPACE ts_name]
   [PARALLEL (DEGREE n)]
   [BUILD {IMMEDIATE|DEFERRED}]
   [{ REFRESH {FAST|COMPLETE|FORCE}
     [{ON COMMIT|ON DEMAND}]
     | NEVER REFRESH } ]
   [{ENABLE|DISABLE} QUERY REWRITE]

AS SELECT ... FROM ...
Creating Materialized Views: Example

```sql
CREATE MATERIALIZED VIEW cost_per_year_mv
ENABLE QUERY REWRITE
AS
SELECT        t.week_ending_day,
             t.calendar_year,
             p.prod_subcategory,
             sum(c.unit_cost) AS dollars
FROM          costs c,
             times t,
             products p
WHERE         c.time_id = t.time_id
AND           c.prod_id = p.prod_id
GROUP BY      t.week_ending_day,
             t.calendar_year,
             p.prod_subcategory;

Materialized view created.
```
Types of Materialized Views

- **Materialized views with aggregates**

```sql
CREATE MATERIALIZED VIEW cust_sales_mv AS
SELECT c.cust_id, s.channel_id,
       SUM(amount_sold) AS amount
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id, s.channel_id;
```

- **Materialized views containing only joins**

```sql
CREATE MATERIALIZED VIEW sales_products_mv AS
SELECT s.time_id, p.prod_name
FROM sales s, products p
WHERE s.prod_id = p.prod_id;
```
Refresh Methods

- You can specify how you want your materialized views to be refreshed from the detail tables by selecting one of four options:
  - COMPLETE
  - FAST
  - FORCE
  - NEVER

- You can view the `REFRESH_METHOD` in the `ALL_MVIEWS` data dictionary view.
Refresh Modes

- **Manual refresh**
  - Specify **ON DEMAND** option
  - By using the **DBMS_MVIEW** package

- **Automatic refresh Synchronous**
  - Specify **ON COMMIT** option
  - Upon commit of changes to the underlying tables but independent of the committing transaction

- **Automatic refresh Asynchronous**
  - Specify using **START WITH** and **NEXT** clauses
  - Defines a refresh interval for the materialized view

- **REFRESH_MODE in ALL_MVIEWS**
Manual Refresh with **DBMS_MVIEW**

- For **ON DEMAND** refresh only
- Three procedures with the **DBMS_MVIEW** package:
  - REFRESH
  - REFRESH_ALL_MVIEWS
  - REFRESH_DEPENDENT
Materialized Views: Manual Refresh

Specific materialized views:

```sql
Exec DBMS_MVIEW.REFRESH('cust_sales_mv');
```

Materialized views based on one or more tables:

```sql
VARIABLE fail NUMBER;
exec DBMS_MVIEW.REFRESH_DEPENDENT(-:fail,'CUSTOMERS,SALES');
```

All materialized views due for refresh:

```sql
VARIABLE fail NUMBER;
exec DBMS_MVIEW.REFRESH_ALL_MVIEWS(:fail);
```
Query Rewrites

- If you want to use a materialized view instead of the base tables, a query must be rewritten.
- Query rewrites are transparent to applications.
- Query rewrites do not require special privileges on the materialized view.
- A materialized view can be enabled or disabled for query rewrites.
Query Rewrites

- **Use** `EXPLAIN PLAN` or `AUTOTRACE` **to verify** that query rewrites occur.
- **Check** the query response:
  - Fewer blocks are accessed.
  - Response time should be significantly better.
Enabling and Controlling Query Rewrites

- Query rewrites are available with cost-based optimization only.

```sql
QUERY_REWRITE_ENABLED = {true | false | force}
QUERY_REWRITE_INTEGRITY = {enforced | trusted | stale_tolerated}
```

- The following optimizer hints influence query rewrites:
  - `REWRITE`
  - `NOREWRITE`
  - `REWRITE_OR_ERROR`
Query Rewrite: Example

```
EXPLAIN PLAN FOR
SELECT t.week_ending_day,
  t.calendar_year,
  p.prod_subcategory,
  sum(c.unit_cost) AS dollars
FROM costs c,
  times t,
  products p
WHERE c.time_id = t.time_id
...
```

Execution Plan
```
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost...)
1    0   MAT_VIEW REWRITE ACCESS (FULL) OF 'costs_per_year_mv' (MAT_VIEW REWRITE) (Cost...)
```
Query Rewrite: Example

```sql
SELECT   t.week_ending_day
,        t.calendar_year
,        p.prod_subcategory
,        sum(c.unit_cost) AS dollars
FROM     costs c, times t, products p
WHERE    c.time_id = t.time_id
AND      c.prod_id = p.prod_id
AND      t.calendar_year = '1999'
GROUP BY t.week_ending_day, t.calendar_year
,        p.prod_subcategory
HAVING   sum(c.unit_cost) > 10000;

SELECT   week_ending_day
,        prod_subcategory
,        dollars
FROM     cost_per_year_mv
WHERE    calendar_year = '1999'
AND      dollars > 10000;
```
Verifying Query Rewrite

CREATE MATERIALIZED VIEW cust_orders_mv
ENABLE QUERY REWRITE AS
SELECT c.customer_id, SUM(order_total) AS amt
FROM   oe.orders s, oe.customers c
WHERE  s.customer_id = c.customer_id
GROUP BY c.customer_id;

SELECT /*+ REWRITE_OR_ERROR */ c.customer_id, SUM(order_total) AS amt
FROM   oe.orders s, oe.customers c
WHERE  s.customer_id = c.customer_id
GROUP BY c.customer_id;

ORA-30393: a query block in the statement did not rewrite
SQL Access Advisor

For a given workload, the SQL Access Advisor:

• Recommends creating the appropriate:
  – Materialized views
  – Materialized view logs
  – Indexes

• Provides recommendations to optimize for:
  – Fast refresh
  – Query rewrite

• Can be run:
  – From Oracle Enterprise Manager by using the SQL Access Advisor Wizard
  – By invoking the DBMS_ADVISOR package
Using the **DBMS_MVIEW** Package

**DBMS_MVIEW methods**

- **EXPLAIN_MVIEW**
- **EXPLAIN_REWRITE**
- **TUNE_MVIEW**
Tuning Materialized Views for Fast Refresh and Query Rewrite

```sql
DBMS_ADVISOR.TUNE_MVIEW (  
    task_name IN OUT VARCHAR2,
    mv_create_stmt IN [CLOB | VARCHAR2]
);
```
Results of Tune_MVIEW

• IMPLEMENTATION recommendations
  – CREATE MATERIALIZED VIEW LOG statements
  – ALTER MATERIALIZED VIEW LOG FORCE statements
  – One or more CREATE MATERIALIZED VIEW statements

• UNDO recommendations
  – DROP MATERIALIZED VIEW statements
DBMS_MVIEW.EXPLAIN_MVIEW Procedure

- Accepts:
  - Materialized view name
  - SQL statement
- Advises what is and what is not possible:
  - For an existing materialized view
  - For a potential materialized view before you create it
- Stores results in MV_CAPABILITIES_TABLE (relational table) or in a VARRAY
- utlxmv.sql must be executed as the current user to create MV_CAPABILITIES_TABLE.
**Explain Materialized View: Example**

```sql
EXEC dbms_mview.explain_mview('cust_sales_mv', '123');

SELECT capability_name, possible, related_text, msgtxt
FROM mv_capabilities_table
WHERE statement_id = '123' ORDER BY seq;

---

<table>
<thead>
<tr>
<th>CAPABILITY_NAME</th>
<th>P</th>
<th>RELATED_TE</th>
<th>MSGTXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFRESH_COMPLETE</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFRESH_FAST</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REWRITE</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCT_TABLE</td>
<td>N</td>
<td>SALES</td>
<td>no partition key or PMARKER in select list</td>
</tr>
<tr>
<td>PCT_TABLE</td>
<td>N</td>
<td>CUSTOMERS</td>
<td>relation is not a partitioned table</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Designing for Query Rewrite

Query rewrite considerations:

• Constraints
• Outer joins
• Text match
• Aggregates
• Grouping conditions
• Expression matching
• Date folding
• Statistics
# Materialized View Hints

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REWRITE</td>
<td>Rewrites a query in terms of materialized views</td>
</tr>
<tr>
<td>REWRITE OR ERROR</td>
<td>Forces an error if a query rewrite is not possible</td>
</tr>
<tr>
<td>NO_REWRITE</td>
<td>Disables query rewrite for the query block</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned how to:

• Create materialized views
• Enable query rewrites using materialized views
Data Warehouse Tuning Considerations
Objectives

After completing this lesson, you should understand the following:

- Star transformations
- Basics of parallel execution
- Types of parallelism
- Parallel query
- Parallelizing SQL statements
- Viewing parallel queries with EXPLAIN PLAN
Star Transformation

With the star transformation, you can:

• Execute star queries efficiently, especially in the following cases:
  – Number of dimension tables is large.
  – Fact table is sparse.
  – Not all dimensions have constraining predicates.

• Set the `STAR_TRANSFORMATION_ENABLED` initialization parameter

• Use the `STAR_TRANSFORMATION` hint
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 ('I','P','S')
AND c.cust_city = 'Asten'
AND p.prod_id > 40000;
Steps in Execution

The Oracle Server processes the query by carrying out the following steps:

1. Use a bitmap index to identify row sets for sales in channels I, P, or S. Combine these with a bitmap OR operation.
2. Use a bitmap for rows corresponding to sales in the city of Asten.
3. Use a bitmap for rows with product ID greater than 40,000.
4. Combine these three bitmaps into a single bitmap with the bitmap AND operation.
5. Use this final bitmap to access rows that satisfy all the conditions from the fact table.
6. Join these rows from the fact table to the dimension tables.
Introduction to Parallel Execution

Parallel execution improves processing for:

- Queries requiring large table scans, joins, or partitioned index scans
- Creation of large indexes
- Creation of large tables
- Bulk inserts, updates, merges, and deletes
- Large sorts
When to Implement Parallel Execution

- DSS and data warehousing environments
- OLTP systems
  - During batch processing
  - During schema maintenance operations
Operations That Can Be Parallelized

- Access methods
- Join methods
- DDL
- DML
- Miscellaneous SQL operations
- Query
- SQL*Loader
How Parallel Execution Works

The query coordinator:

- Parses the query and determines the degree of parallelism
- Allocates one or two sets of slaves
- Controls the query and sends instructions to the PQ slaves
- Determines which tables or indexes need to be scanned by the PQ slaves
- Produces the final output to the user
Degree of Parallelism

User process

SELECT /*+ PARALLEL(ORDERS 2) */ ...
Parallelization Rules for SQL Statements

- A parallel query looks at every table and index in the statement.
- The basic rule is to pick the table or index with the largest DOP.
- For parallel DML, the reference object that determines the DOP is the table being modified by a DML operation.
- If the parallel DML statement includes a subquery, the subquery’s DOP is the same as the DML operation.
- For parallel DDL, the reference object that determines the DOP is the table, index, or partition that is being created, rebuilt, split, or moved.
- If the parallel DDL statement includes a subquery, the subquery’s DOP is the same as the DDL operation.
When to Parallelize a `SELECT` Statement

- **A parallel hint**
  - The query includes a parallel hint specification.
  - The schema objects have a `PARALLEL` declaration.
- **One or more tables specified in the query require one of the following:**
  - A full table scan
  - An index range scan
  - Absence of scalar subqueries are in the `SELECT` list.
Parallel DML

UPDATE /*+ PARALLEL(SALES,4) */ SALES
SET PROD_MIN_PRICE = PROD_MIN_PRICE * 1.10

ALTER SESSION FORCE PARALLEL DML

INSERT /*+ PARALLEL(new_emp,2) */ INTO new_emp
SELECT /*+ PARALLEL(employees,4) */ * FROM employees;

The DOP used is 2, as specified in the INSERT hint
Parallel DDL

Use default DOP

```
ALTER TABLE employees PARALLEL;
```

Use DOP of 4

```
ALTER TABLE employees PARALLEL 4;
```

Session override

```
ALTER SESSION FORCE PARALLEL DDL
```
Parallelization Rules

- Priority 1: \texttt{PARALLEL} hint
- Priority 2: \texttt{PARALLEL} clause or \texttt{ALTER SESSION FORCE PARALLEL} ...
- Priority 3: \texttt{PARALLEL} declaration while creating objects
## Displaying Parallel Explain Plans

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
<th>TQ</th>
<th>IN-OUT</th>
<th>PQ Distrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>41</td>
<td>1066</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PX COORDINATOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>PX SEND QC (RANDOM)</td>
<td>:TQ10001</td>
<td>41</td>
<td>1066</td>
<td>4</td>
<td>Q1,01</td>
<td>P-&gt;S</td>
<td>QC (RAND)</td>
</tr>
<tr>
<td>3</td>
<td>SORT GROUP BY</td>
<td></td>
<td>41</td>
<td>1066</td>
<td>4</td>
<td>Q1,01</td>
<td>PCWP</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>PX RECEIVE</td>
<td></td>
<td>41</td>
<td>1066</td>
<td>4</td>
<td>Q1,01</td>
<td>PCWP</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PX SEND HASH</td>
<td>:TQ10000</td>
<td>41</td>
<td>1066</td>
<td>4</td>
<td>Q1,00</td>
<td>P-&gt;P</td>
<td>HASH</td>
</tr>
<tr>
<td>6</td>
<td>SORT GROUP BY</td>
<td></td>
<td>41</td>
<td>1066</td>
<td>4</td>
<td>Q1,00</td>
<td>PCWP</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PX BLOCK ITERATOR</td>
<td></td>
<td>41</td>
<td>1066</td>
<td>1</td>
<td>Q1,00</td>
<td>PCWC</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>TABLE ACCESS FULL EMP2</td>
<td></td>
<td>41</td>
<td>1066</td>
<td>1</td>
<td>Q1,00</td>
<td>PCWP</td>
<td></td>
</tr>
</tbody>
</table>
Disabling Parallel Execution

ALTER SESSION DISABLE PARALLEL DML;

ALTER TABLE employees NOPARALLEL;
Hints for Parallel Execution

- PARALLEL
- NO_PARALLEL
- PQ_DISTRIBUTED
- PARALLEL_INDEX
- NO_PARALLEL_INDEX
Summary

In this lesson, you should have learned how to do the following:

- Describe parallel execution
- Describe the types of parallelism
- Use parallel query
- Parallelize SQL statements
- View parallel queries with `EXPLAIN PLAN`
Objectives

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

• Identify the purpose and benefits of optimizer plan stability
• Create stored outlines
• Use stored outlines
• Edit stored outlines
• Maintain stored outlines
Optimizer Plan Stability

- Enables well-tuned applications to force the use of the desired SQL access path
- Maintains consistent execution plans through database changes
- Is implemented using stored outlines consisting of hints
- Groups stored outlines in categories
Plan Equivalence

- Plans are maintained through:
  - New Oracle Database versions
  - New statistics on objects
  - Initialization parameter changes
  - Database reorganizations
  - Schema changes

- Plan equivalence can control execution plans for third-party applications.
Creating Stored Outlines

• For all statements during a session:

```
SQL> ALTER SESSION
  2  SET create_stored_outlines = OTLN1;
SQL> SELECT ... ;
SQL> SELECT ... ;
```

• For a specific statement:

```
SQL> CREATE OR REPLACE OUTLINE CU_CO_JOIN
  2  FOR CATEGORY OTLN1 ON
  3    SELECT co.country_name,
  4    cu.cust_city, cu.cust_last_name
  5    FROM   countries co
  6    JOIN customers cu ON
...```
Using Stored Outlines

• **Set** `USE_STORED_OUTLINES` **to** TRUE **or** to a category name:
  
  ```sql
  SQL> ALTER SESSION
  2  SET use_stored_outlines = OTLN1;
  SQL> SELECT ... 
  ```

• **You can set** `CREATE_STORED_OUTLINES` **and** `USE_STORED_OUTLINES` **at two levels:**
  
  – ALTER SYSTEM
  – ALTER SESSION
Data Dictionary Information

SQL> SELECT name, category, used, sql_text
    2 FROM user_outlines;

SQL> SELECT node, hint
    2 FROM user_outline_hints
    3 WHERE name = ...;

SQL> SELECT sql_text, outline_category
    2 FROM v$sql
    3 WHERE ...;
Execution Plan Logic

- **In shared pool?**
  - **y:** Same outline category?
    - **y:** Execute outline plan
    - **n:** Query DD for matching outline
      - **n:** Regular execute
      - **y:** Integrate outline and generate plan
  - **n:**

**Flowchart Diagram:**

1. **In shared pool?**
   - **y:** Same outline category?
     - **y:** Execute outline plan
     - **n:** Query DD for matching outline
       - **n:** Regular execute
       - **y:** Integrate outline and generate plan
   - **n:**

---

Copyright © 2004, Oracle. All rights reserved.
Maintaining Stored Outlines

• **Use DBMS_OUTLN to:**
  – Drop unused outlines
  – Drop categories of outlines
  – Rename a category

• **Use ALTER OUTLINE to:**
  – Rename an outline
  – Rebuild an outline
  – Change the category of an outline

• Outlines are stored in the OUTLN schema.
Maintaining Stored Outlines

```sql
SQL> BEGIN
  2     dbms_outln.drop_unused;
  3     dbms_outln.update_by_cat
  4      ('default','otln1');
  5     dbms_outln.drop_by_cat('otln1');
  6     END;
```
Public Versus Private Outlines

- **Public outlines**
  - Default setting when creating outlines
  - Stored in the `OUTLN` schema
  - Used when `USE_STORED_OUTLINES` is set to `TRUE` or a category
- **Private outlines**
  - Stored in the user’s schema for the duration of the session
  - Can be edited
  - Used when `USE_PRIVATE_OUTLINES` is set to `TRUE` or a category
  - Changes can be saved as public outlines.
Outline Editing: Overview

- Stored outlines can be edited.
- Users can tune execution plans without having to change the application.
- This is possible by editing the content of the saved plan.
Outline Editing: Overview

- Outline is cloned in a staging area.
- Outline is edited in the user’s session.
- When satisfied with the result, the editor can publicize the result to the user community.
Editable Attributes

- Join order
- Join methods
- Access methods
- Distributed execution plans
- Distribution methods for parallel query execution
- Query rewrite
- View and subquery merging
Editing Stored Outlines

To edit and use private outlines:
1. Create the outline tables in the current schema.
2. Copy the selected outline to a private outline.
3. Edit the outline that is stored as a private outline.
4. To use the private outline, set the USE_PRIVATE_OUTLINE parameter.
5. To allow public access to the new stored outline, overwrite the stored outline.
6. Reset USE_PRIVATE_OUTLINE to FALSE.
Outlines: Administration and Security

• Privileges required for cloning outlines
  – `SELECT_CATALOG_ROLE`
  – `CREATE ANY OUTLINE`
  – `EXECUTE privilege on DBMS_OUTLN_EDIT`
• `DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES`
  – Creates required temporary tables in user’s schema for cloning and editing outlines
Outlines: Administration and Security

- The OUTLINE_SID is available in the V$SQL fixed view.
- OUTLINE_SID identifies the session ID from which the outline was retrieved.
**Configuration Parameters**

**USE_PRIVATE_OUTLINES** is a session parameter that controls the use of private outlines instead of public outlines.

```
ALTER SESSION SET use_private_outlines = [TRUE | FALSE | category_name ];
```

- **TRUE** enables the use of private outlines in the **DEFAULT** category.
- **FALSE** disables use of private outlines.
- **category_name** enables use of private outlines in the named category.
Cloning Outlines

The CREATE OUTLINE command can be used to clone outlines:

```
CREATE [OR REPLACE]
  [PUBLIC | PRIVATE] OUTLINE [outline_name]
  [FROM [PUBLIC | PRIVATE] source_outline_name]
  [FOR CATEGORY category_name] [ON statement]
```

Example

```
CREATE OR REPLACE OUTLINE public_outline2
FROM public_outline1 FOR CATEGORY cat2;
```
SQL Profiles

- SQL Profiles
  - Are an alternative to using hints
  - Consist of auxiliary stored statistics that are specific to a statement
  - Contain execution history information about the SQL statement that the Automatic Tuning Optimizer uses to set optimizer parameter settings
- A SQL Profile, after being accepted, is stored persistently in the data dictionary.
- Information about SQL Profiles can be obtained from the `DBA_SQL_PROFILES` view.
Summary

In this lesson, you should have learned how to:

• Use stored outlines to ensure execution-plan consistency
• Create outlines for a session or a single statement
• Organize outlines in categories
• Enable or disable using outlines or categories of outlines
• Maintain outlines with the DBMS_OUTLN package or the ALTER OUTLINE command
Objectives

After completing this lesson, you should be able to do the following Use Statspack.
Overview of Statspack

Statspack

- collects data about high-resource SQL.
- precalculates many useful data
  - cache hit ratios
  - rates
  - transaction statistics
- Uses permanent tables owned by the PERFSTAT user to store performance statistics.
- Separates data collection from report generation
- Can be automated
Statspack Mechanism

• The `PERFSTAT` user is created automatically at installation.
• `PERFSTAT` owns all objects needed by the Statspack package and has query privileges on the `V$` views.
• A snapshot is a single collection of performance data, identified by a snapshot ID, which is generated at the time the snapshot is taken.
• The performance report uses start and end snapshot IDs and then calculates activity on the instance between the two snapshots.
Taking a Statistics Snapshot

```sql
SQL> variable snap number;
SQL> begin
    2 :snap := statspack.snap;
    3   end;
  4 /
PL/SQL procedure successfully completed.
```
Automatic Statistics Gathering

- You need to take multiple snapshots over a period of time for comparison
- To automate the collection at regular intervals use the Oracle `DBMS_JOB` procedure to schedule snapshots.
- The script `SPAUTO.SQL` schedules a snapshot every hour, on the hour.
Generating a Performance Report

The Statspack package includes two reports.

- **SPREPORT.SQL**
  - Covers all aspects of instance performance
  - Calculates and prints ratios and differences for all statistics between the two snapshots
  - Prompts for:
    - The beginning snapshot ID
    - The ending snapshot ID
    - The name of the report text file to be created

- **SPREPSQL.SQL**
  - Displays statistics, the complete SQL text, and information on any SQL plans associated with that statement.
## Snapshot Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>General performance statistics</td>
</tr>
<tr>
<td>&gt;= 5</td>
<td>Additional data: SQL statements</td>
</tr>
<tr>
<td>&gt;= 6</td>
<td>Additional Data: SQL Plans and SQL Plan Usage</td>
</tr>
<tr>
<td>&gt;= 7</td>
<td>Additional data: Segment Level Statistics</td>
</tr>
<tr>
<td>&gt;= 10</td>
<td>Additional Statistics: Parent and Child Latches</td>
</tr>
</tbody>
</table>
## Snapshot Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>General performance statistics</td>
</tr>
<tr>
<td>&gt;= 5</td>
<td>Additional data: SQL statements</td>
</tr>
<tr>
<td>&gt;= 6</td>
<td>Additional Data: SQL Plans and SQL Plan Usage</td>
</tr>
<tr>
<td>&gt;= 7</td>
<td>Additional data: Segment Level Statistics</td>
</tr>
<tr>
<td>&gt;= 10</td>
<td>Additional Statistics: Parent and Child Latches</td>
</tr>
</tbody>
</table>
Altering Snapshot Defaults

SQL> EXECUTE STATSPACK.MODIFY_STATSPACK_PARAMETER
       (i_snap_level=>10, i_buffer_gets_th=>10000, i_disk_reads_th=>1000);

SQL> EXECUTE STATSPACK.SNAP(i_snap_level=>10, i_modify_parameter=>'true');

SQL> EXECUTE STATSPACK.MODIFY_STATSPACK_PARAMETER
       (i_snap_level=>10, i_buffer_gets_th=>10000, i_disk_reads_th=>1000);

SQL> EXECUTE STATSPACK.SNAP(i_snap_level=>6);
Removing Statspack Data

- Use the `SPPURGE.SQL` script
- Deletes snapshots that fall between the begin and end snapshot IDs you specify
Summary

In this lesson, you should have learned about the use of Statspack in statistics gathering.