

# Oracle Database 10g: SQL Tuning Workshop

Student Guide • Volume 1

D17265GC11

Edition 1.1

March 2007

D49768

**ORACLE®**

**Author**

Priya Vennapusa

**Technical Contributors  
and Reviewers**

Andrew Brannigan  
Cecelia Gervasio  
Chika Izumi  
Connie Green  
Dairy Chan  
Donna Keesling  
Graham Wood  
Harald Van Breederode  
Helen Robertson  
Janet Stern  
Jean Francois Verrier  
Joel Goodman  
Lata Shivaprasad  
Lawrence Hopper  
Lillian Hobbs  
Marcelo Manzano  
Martin Jensen  
Mughees Minhas  
Ric Van Dyke  
Robert Bungenstock  
Russell Bolton  
Dr. Sabine Teuber  
Stefan Lindblad

**Editor**

Raj Kumar

**Graphic Designer**

Rajiv Chandrabhanu

**Publisher**

Jobi Varghese

Copyright © 2007, Oracle. All rights reserved.

**Disclaimer**

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

**Restricted Rights Notice**

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

**U.S. GOVERNMENT RIGHTS**

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

**Trademark Notice**

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

# Contents

## 1 Oracle Database Architecture: Overview

- Objectives 1-2
- Oracle Database Architecture: Overview 1-3
- Oracle Instance Management 1-4
- Database Physical Structure 1-5
- Oracle Memory Structures 1-6
- Automatic Shared Memory Management 1-8
- Shared Pool 1-9
- Shared SQL Areas 1-10
- Program Global Area (PGA) 1-11
- Automated SQL Execution Memory (PGA) Management 1-13
- Connecting to an Instance 1-14
- SQL Statement Processing Phases 1-16
- SQL Statement Processing Phases: Parse 1-17
- SQL Statement Processing Phases: Bind 1-19
- SQL Statement Processing Phases: Execute and Fetch 1-20
- Processing a DML Statement 1-21
- COMMIT Processing 1-23
- Functions of the Oracle Query Optimizer 1-25
- Top Database Performance Issues 1-26
- Summary 1-28

## 2 Following a Tuning Methodology

- Objectives 2-2
- Performance Problems 2-3
- Factors to Be Managed 2-4
- Tuning Goals 2-6
- Overview of SQL Tuning 2-8
- Identifying High-Load SQL 2-9
- Manual Tuning 2-10
- Gather Information About Referenced Objects 2-11
- Gathering Optimizer Statistics 2-12
- Reviewing the Execution Plan 2-14
- Restructuring the SQL Statements 2-16
- Restructuring the Indexes 2-18

Maintaining Execution Plans over Time 2-19  
Automatic SQL Tuning 2-20  
Automatic Tuning Mechanisms 2-22  
SQL Tuning Advisor 2-23  
SQL Access Advisor 2-24  
Summary 2-25

### **3 Designing and Developing for Performance**

Objectives 3-2  
Understanding Scalability 3-3  
Scalability with Application Design, Implementation, and Configuration 3-4  
Configuring the Appropriate System Architecture for Your Requirements 3-5  
Proactive Tuning Methodology 3-6  
Simplicity In Application Design 3-7  
Data Modeling 3-8  
Table Design 3-9  
Index Design 3-10  
Using Views 3-12  
SQL Execution Efficiency 3-13  
Importance of Sharing Cursors 3-14  
Writing SQL to Share Cursors 3-15  
Controlling Shared Cursors 3-16  
Performance Checklist 3-17  
Summary 3-18  
Practice Lesson 3 3-19

### **4 Introduction to the Optimizer**

Objectives 4-2  
Oracle Optimizer 4-3  
Functions of the Query Optimizer 4-5  
Selectivity 4-7  
Cardinality and Cost 4-8  
Query Optimizer Statistics in the Data Dictionary 4-9  
Enabling Query Optimizer Features 4-10  
Controlling the Behavior of the Optimizer 4-11  
Choosing an Optimizer Approach 4-13  
Setting the Optimizer Approach 4-14  
Optimizing for Fast Response 4-15  
Optimizing SQL Statements 4-17  
How the Query Optimizer Executes Statements 4-18  
Access Paths 4-19

Join Orders 4-20  
Join Methods 4-21  
Summary 4-22

## **5 Optimizer Operations**

Objectives 5-2  
Review: How the Query Optimizer Executes Statements 5-3  
Access Paths 5-4  
Choosing an Access Path 5-5  
Full Table Scans 5-6  
Row ID scans 5-8  
Index Scans 5-9  
Joining Multiple Tables 5-13  
Join Terminology 5-14  
SQL:1999 Outer Joins 5-16  
Oracle Proprietary Outer Joins 5-17  
Full Outer Joins 5-18  
Execution of Outer Joins 5-19  
Join Order Rules 5-20  
Join Optimization 5-21  
Join Methods 5-22  
Nested Loop Joins 5-23  
Nested Loop Join Plan 5-24  
When Are Nested Loop Joins Used? 5-25  
Hash Joins 5-26  
Hash Join Plan 5-27  
When Are Hash Joins Used? 5-28  
Sort-Merge Joins 5-29  
Sort-Merge Join Plan 5-30  
When Are Sort-Merge Joins Used? 5-31  
Star Joins 5-32  
How the Query Optimizer Chooses Execution Plans for Joins 5-33  
Subqueries and Joins 5-35  
Sort Operations 5-38  
Tuning Sort Performance 5-39  
Top-N SQL 5-40  
Memory and Optimizer Operations 5-41  
Summary 5-42

## **6 Execution Plans**

Objectives 6-2

What Is an Execution Plan? 6-3

Methods for Viewing Execution Plans 6-4

Using Execution Plans 6-6

DBMS\_XPLAN Package: Overview 6-7

EXPLAIN PLAN Command 6-9

EXPLAIN PLAN Command: Example 6-11

EXPLAIN PLAN Command: Output 6-12

Parse Tree 6-13

Using the V\$SQL\_PLAN View 6-14

V\$SQL\_PLAN Columns 6-15

Querying V\$SQL\_PLAN 6-16

V\$SQL\_PLAN\_STATISTICS View 6-17

Automatic Workload Repository 6-18

Managing AWR with PL/SQL 6-20

AWR Views 6-22

Querying the AWR 6-23

SQL\*Plus AUTOTRACE 6-25

SQL\*Plus AUTOTRACE: Examples 6-26

SQL\*Plus AUTOTRACE: Statistics 6-27

Summary 6-28

Practice 6: Overview 6-29

## **7 Gathering Statistics**

Objectives 7-2

What Are Optimizer Statistics? 7-3

Types of Optimizer Statistics 7-4

How Statistics Are Gathered 7-6

Automatic Statistics Gathering 7-7

Manual Statistics Gathering 7-8

Managing Automatic Statistics Collection 7-9

Job Configuration Options 7-10

Managing the Job Scheduler 7-11

Managing the Maintenance Window 7-12

Changing the GATHER\_STATS\_JOB Schedule 7-13

Statistics Collection Configuration 7-14

DML Monitoring 7-15

Sampling 7-17

Degree of Parallelism 7-19

- Histograms 7-20
- Creating Histograms 7-21
- Viewing Histogram Statistics 7-23
- Histogram Tips 7-24
- Bind Variable Peeking 7-26
- Cascading to Indexes 7-27
- Managing Statistics Collection: Example 7-28
- When to Gather Manual Statistics 7-29
- Statistics Gathering: Manual Approaches 7-30
- Dynamic Sampling 7-31
- Locking Statistics 7-32
- Verifying Table Statistics 7-33
- Verifying Column Statistics 7-34
- Verifying Index Statistics 7-35
- History of Optimizer Statistics 7-37
- Managing Historical Optimizer Statistics 7-38
- Generating System Statistics 7-40
- Statistics on Dictionary Objects 7-42
- Dictionary Statistics: Best Practices 7-43
- Summary 7-44
- Practice 7: Overview 7-45

## **8 Application Tracing**

- Objectives 8-2
- Overview of Application Tracing 8-3
- End to End Application Tracing 8-4
- End to End Application Tracing Using EM 8-5
- Using DBMS\_MONITOR 8-6
- Viewing Gathered Statistics for End to End Application Tracing 8-8
- trcsess Utility 8-9
- SQL Trace Facility 8-12
- Information Captured by SQL Trace 8-13
- How to Use the SQL Trace Facility 8-14
- Initialization Parameters 8-15
- Enabling SQL Trace 8-17
- Formatting Your Trace Files 8-19
- TKPROF Command Options 8-20
- Output of the TKPROF Command 8-22
- TKPROF Output with No Index: Example 8-27
- TKPROF Output with Index: Example 8-28

Summary 8-29  
Practice 8: Overview 8-30

## **9 Identifying High-Load SQL**

Objectives 9-2  
SQL Tuning Process: Overview 9-3  
Identifying High-Load SQL 9-4  
Automatic Database Diagnostic Monitor 9-6  
ADDM Output 9-7  
Manual Identification: Top SQL 9-8  
Spot SQL 9-9  
Period SQL 9-10  
Manual Identification: Statspack 9-11  
Using Dynamic Performance Views 9-12  
V\$SQLAREA View 9-13  
Querying the V\$SQLAREA View 9-14  
Investigating Full Table Scan Operations 9-15  
Summary 9-16

## **10 Automatic SQL Tuning**

Objectives 10-2  
SQL Tuning Process: Overview 10-3  
Automatic SQL Tuning 10-4  
Automatic Tuning Optimizer 10-5  
SQL Tuning Advisor 10-6  
SQL Tuning Advisor Analysis 10-7  
SQL Profiling 10-9  
SQL Access Path Analysis 10-11  
SQL Structure Analysis 10-12  
SQL Tuning Advisor: Usage Model 10-13  
SQL Tuning Set 10-15  
SQL Tuning Views 10-18  
Enterprise Manager: Usage Model 10-19  
SQL Access Advisor 10-20  
SQL Access Advisor: Features 10-22  
SQL Access Advisor: Usage Model 10-23  
SQL Access Advisor: User Interface 10-24  
SQL Tuning Advisor and SQL Access Advisor 10-25  
Summary 10-26



## **11 Index Usage**

- Objectives 11-2
- Indexing Guidelines 11-3
- Types of Indexes 11-4
- When to Index 11-6
- Effect of DML Operations on Indexes 11-8
- Indexes and Constraints 11-9
- Indexes and Foreign Keys 11-10
- Basic Access Methods 11-11
- Identifying Unused Indexes 11-12
- Enabling and Disabling the Monitoring of Index Usage 11-13
- Index Tuning Using the SQL Access Advisor 11-14
- Summary 11-15

## **12 Using Different Indexes**

- Objectives 12-2
- Composite Indexes 12-3
- Composite Index Guidelines 12-4
- Skip Scanning of Indexes 12-5
- Bitmap Index 12-6
- When to Use Bitmap Indexes 12-7
- Advantages of Bitmap Indexes 12-8
- Bitmap Index Guidelines 12-9
- Bitmap Join Index 12-10
- Bitmap Join Index: Advantages and Disadvantages 12-12
- Function-Based Index 12-14
- Function-Based Indexes: Usage 12-15
- Index-Organized Tables: Overview 12-16
- Index-Organized Tables: Characteristics 12-17
- Advantages and Disadvantages of IOTs 12-18
- Summary 12-19

## **13 Optimizer Hints**

- Objectives 13-2
- Optimizer Hints: Overview 13-3
- Types of Hints 13-4
- Specifying Hints 13-5
- Rules for Hints 13-6
- Hint Recommendations 13-7
- Optimizer Hint Syntax: Example 13-8
- Hint Categories 13-9

Optimization Goals and Approaches 13-10  
Hints for Access Paths 13-11  
Hints for Access Paths 13-13  
INDEX\_COMBINE Hint: Example 13-15  
Hints for Query Transformation 13-17  
Hints for Query Transformation 13-18  
Hints for Join Orders 13-20  
Hints for Join Operations 13-22  
Other Hints 13-24  
Hints for Suppressing Index Usage 13-26  
Hints and Views 13-27  
Hints for View Processing 13-29  
Global and Local Hints 13-30  
Specifying a Query Block in a Hint 13-31  
Specifying a Full Set of Hints 13-33  
Summary 13-34

## **14 Materialized Views**

Objectives 14-2  
Materialized Views 14-3  
If Materialized Views Are Not Used 14-4  
How Many Materialized Views? 14-6  
Creating Materialized Views: Syntax Options 14-7  
Creating Materialized Views: Example 14-8  
Types of Materialized Views 14-9  
Refresh Methods 14-10  
Refresh Modes 14-12  
Manual Refresh with DBMS\_MVIEW 14-14  
Materialized Views: Manual Refresh 14-15  
Query Rewrites 14-16  
Enabling and Controlling Query Rewrites 14-18  
Query Rewrite: Example 14-20  
Verifying Query Rewrite 14-22  
SQL Access Advisor 14-23  
Using the DBMS\_MVIEW Package 14-24  
Tuning Materialized Views for Fast Refresh and Query Rewrite 14-25  
Results of Tune\_MVIEW 14-26  
DBMS\_MVIEW.EXPLAIN\_MVIEW Procedure 14-28  
Explain Materialized View: Example 14-29  
Designing for Query Rewrite 14-30

Materialized View Hints 14-32

Summary 14-33

**Appendix A: Practices**

**Appendix B: Workshops**

**Appendix C: Practice Solutions**

**Appendix D: Data Warehouse Tuning Considerations**

**Appendix E: Optimizer Plan Stability**

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

swati bhatia (swati.june@gmail.com) has a non-transferable license to  
use this Student Guide.

# 1

## Oracle Database Architecture: Overview

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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**

Copyright © 2007, Oracle. All rights reserved.

## Objectives

This lesson gives you an understanding of the tuning process and the different components of an Oracle database that may require tuning.

# 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

Copyright © 2007, Oracle. All rights reserved.

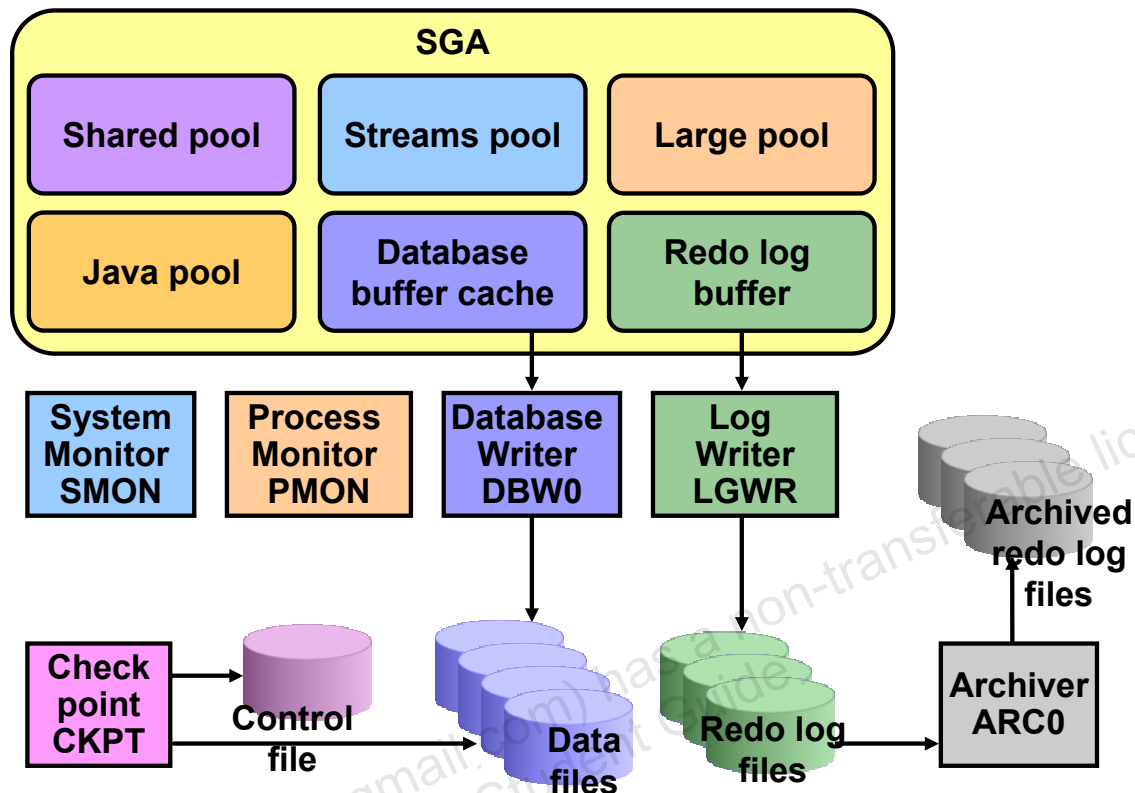
## Oracle Database Architecture: Overview

An Oracle server consists of an Oracle database and an Oracle instance.

- The database consists of physical files such as:
  - The control file where the database configuration is stored
  - The redo log files, which have information required for database recovery
  - The data files where all data is stored
  - The parameter file, which contains the parameters that control the size and properties of an instance
  - The password file, which contains the superuser (SYSOPER and SYSDBA) passwords
- The instance consists of memory structures such as System Global Area (SGA) and Program Global Area (PGA) and background processes that perform tasks within the database as well as the server processes that are initiated for each user session.

The size and structure of an Oracle database and instance impact performance. The physical structure of the database impacts the I/O to hard disks. It is therefore important to both size and place the physical files in such a way that I/O across disks is distributed evenly and waits are minimized. The size of the various memory areas of the instance directly impacts the speed of SQL processing.

# Oracle Instance Management



Copyright © 2007, Oracle. All rights reserved.

## Oracle Instance Management

The instance controls access to the database. Only after the instance is started and the database is opened can users access the database.

An Oracle instance consists of the memory area known as the System Global Area (SGA) and some background processes and server processes.

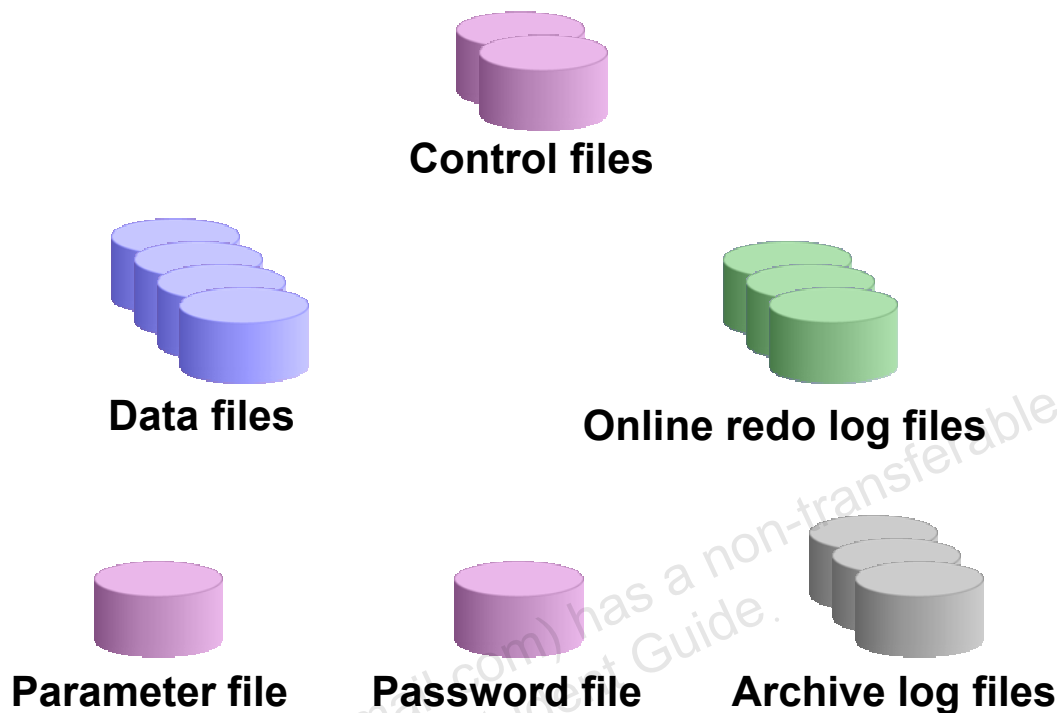
The instance is idle (nonexistent) until it is started. When the instance is started, an initialization parameter file is read and the instance is configured accordingly. The initialization parameter file specifies various parameters that provide the size of the SGA.

You must take care in the initial design of the database system to avoid bottlenecks that may lead to performance problems. In addition, you need to consider:

- Allocating memory to database structures
- Determining I/O requirements of different parts of the database
- Tuning the operating system for optimal performance of the database



# Database Physical Structure



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Database Physical Structure

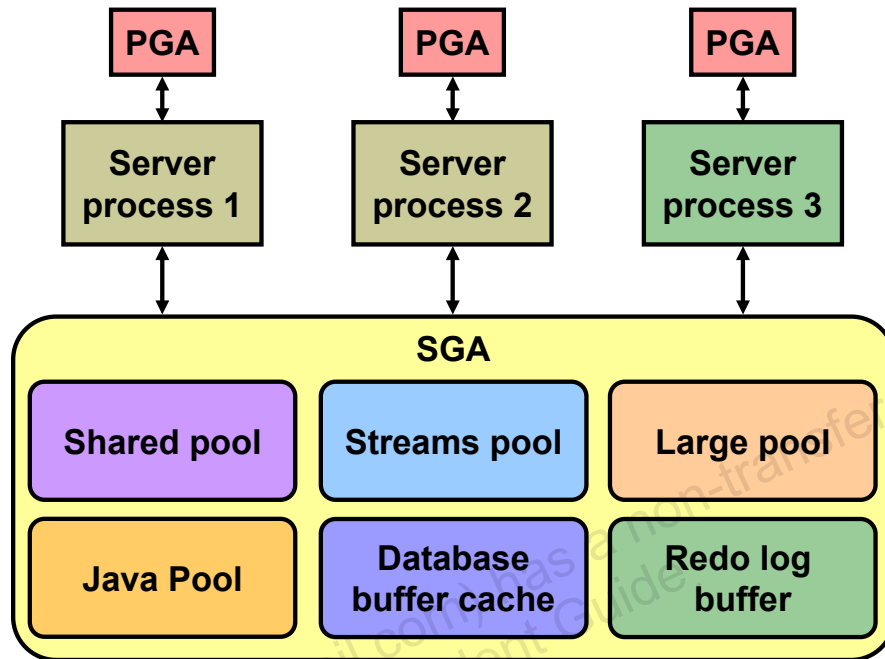
The files that make up an Oracle database affect performance because their size and location affect the distribution of I/O as well as scalability. These files are organized into the following:

- **Control files:** These files contain data about the database itself. These files are critical to the database. Without them you cannot open the database.
- **Data files:** These files contain the data of the database.
- **Online redo log files:** These files allow for instance recovery of the database. If the database were to crash, the database can be recovered with the information in these files.

There are other files that are not officially part of the database but are important to the successful running of the database:

- **Parameter file:** The parameter file is used to define how the instance is configured on startup.
- **Password file:** This file enables super users to connect remotely to the database and perform administrative tasks.
- **Archive log files:** These files ensure database recovery and are copies of the online redo log files. Using these files and a backup of the database, you can recover a lost data file.

# Oracle Memory Structures



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Oracle Memory Structures

The default Oracle database, created by the Oracle Universal Installer (OUI), is preconfigured with initial settings for the memory parameters. The performance of the database depends on the sizing of these memory parameters, so you should fine-tune it to meet the requirements of your growing database.

Two memory parameters, `PGA_AGGREGATE_TARGET` and `SGA_TARGET`, are provided that allow the database to automatically resize the memory structures within the SGA and PGA. These parameters can be set based on the recommendations of Automatic Database Diagnostics Monitor (ADDM), which is available with the Enterprise Edition of Oracle Database 10g, or you can manually run several advisors and use the combined recommendations of these advisors to set the sizes appropriately.

The basic memory structures associated with an Oracle instance include:

- System Global Area (SGA): Shared by all server and background processes
- Program Global Area (PGA): Exclusive to each server and background process.

There is one PGA for each server process.

## Oracle Memory Structures (continued)

The System Global Area (SGA) is a shared memory area that contains data and control information for the instance.

The SGA consists of the following data structures:

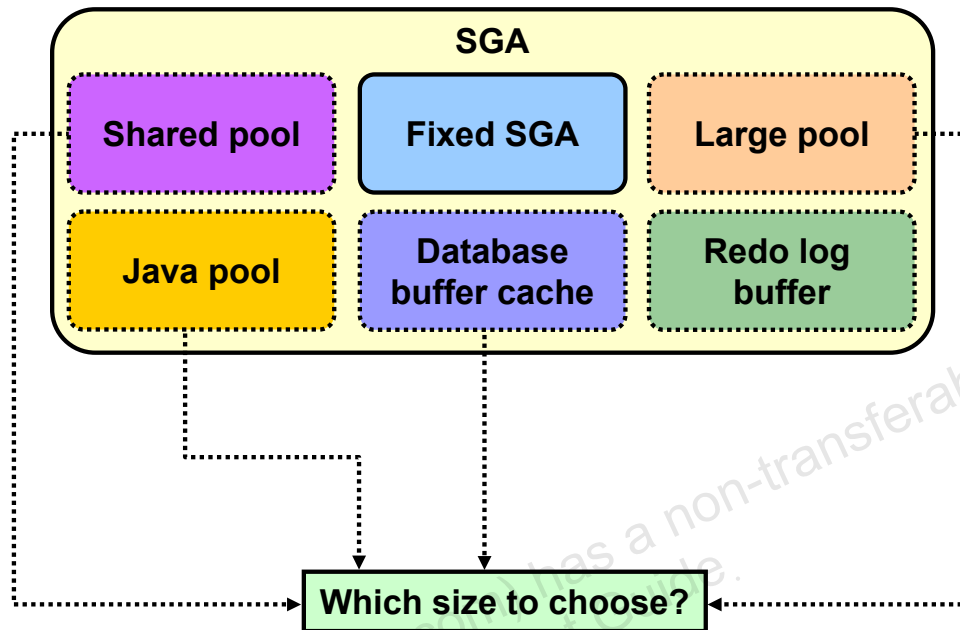
- Database buffer cache: Caches blocks of data retrieved from the data files
- Redo log buffer: Caches redo information (used for instance recovery) until it can be written to the physical redo log files stored on disk
- Shared pool: Caches various constructs that can be shared among users
- Large pool: Optional area in the SGA that provides large memory allocations for Oracle backup and restore operations, I/O server processes, and session memory for the shared server
- Java pool: Used for all session-specific Java code and data within the Java Virtual Machine (JVM)
- Streams pool: Used by Oracle Streams

The Program Global Area (PGA) is a memory region, which contains data and control information for each server process. A server process is a process that services a client's requests. Each server process has its own private PGA that is created when the server process is started. Only a server process can access its own PGA.

Generally, the PGA contains the following:

- Private SQL area: Contains data such as bind information and run-time memory structures. Each session that issues a SQL statement has a private SQL area.
- Session memory: Memory allocated to hold session variables and other information related to the session

# Automatic Shared Memory Management



Copyright © 2007, Oracle. All rights reserved.

## Automatic Shared Memory Management

In earlier Oracle Database releases, you did not have exact control over the total size of the SGA. The reason was that memory was allocated by Oracle for the fixed SGA and for other internal metadata allocations over and above the total size of the user-specified SGA parameters. In prior releases, you needed to manually specify the amount of memory to be allocated for the database buffer cache, shared pool, Java pool, and large pool. It was often a challenge to optimally size these components. Undersizing often resulted in poor performance and out-of-memory errors (ORA-4031), while oversizing wasted memory.

In Oracle Database 10g, you can use the Automatic Shared Memory Management (ASMM) feature to enable the database to automatically determine the size of each of these memory components within the limits of the total SGA size.

Oracle Database 10g uses an SGA size parameter (SGA\_TARGET) that includes all the memory in the SGA, including all the automatically sized components, manually sized components, and any internal allocations during startup. ASMM simplifies the configuration of the SGA by enabling you to specify a total memory amount to be used for all SGA components. The Oracle Database then periodically redistributes memory between these components according to workload requirements.

# Shared Pool

- **The shared pool consists of:**
  - **Data dictionary cache containing information about 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

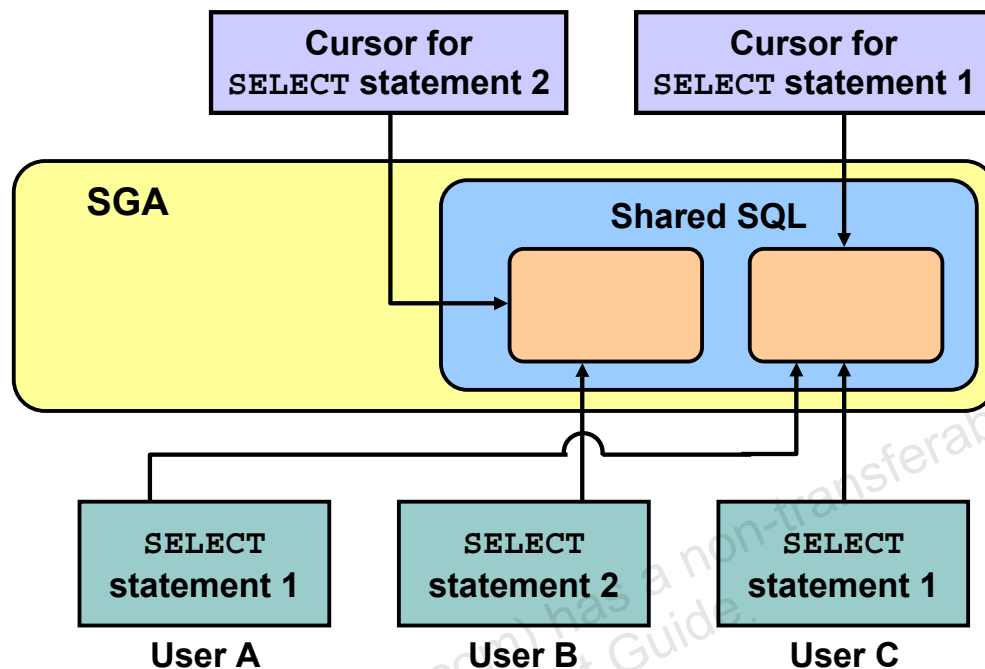
## Shared Pool

The shared pool is a portion of the SGA that contains shared memory constructs such as shared SQL areas, the data dictionary cache, and the fully parsed or compiled representations of PL/SQL blocks.

The main components of the shared pool are the library cache and the dictionary cache. The library cache stores the executable (parsed or compiled) form of recently referenced SQL and PL/SQL code. The dictionary cache stores information such as usernames, segment information, tablespace information, and sequence numbers.

The size of the shared pool affects the number of disk reads. When a SQL statement is executed, the server process checks the dictionary cache for information about object ownership, location, and privileges. If not present, this information is loaded into the dictionary cache through a disk read. The server process then verifies if a parsed form of the same statement exists in the library cache. If this information exists in the cache, then the statement is executed immediately. If not, then the statement is parsed and stored in the library cache for future use and then executed. Because disk reads and parsing are expensive operations, it is preferable that repeated executions of the same statement find required information in memory. Information in the dictionary and library cache are aged out using a least recently used (LRU) algorithm.

# Shared SQL Areas



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Shared SQL Areas

- A shared SQL area is required to process every unique SQL statement submitted to a database.
- A shared SQL area contains information such as the execution plan for the corresponding statement.
- A single shared SQL area is used by multiple users that issue the same statement. In an OLTP environment, a single area therefore needs fewer hard parses, resulting in better resource utilization as well as leaving more shared memory for other uses.

## Cursors

Inside the shared SQL area, each SQL statement is parsed in its own area, known as a cursor. If two users issue the exact same SQL statement, they may use the same cursor. Sharing cursors improves memory utilization and overall performance in SQL execution efficiency.

Each cursor holds the following information:

- The parsed statement
- The execution plan
- A list of referenced objects

## 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Program Global Area (PGA)

A PGA is a memory region containing data and control information for a single process. PGA is a nonshared memory area and a PGA is allocated for each server process. Only that server process can read and write to it. The Oracle Database allocates a PGA when a user connects to an Oracle database. A session is then created.

#### SQL Work Areas

The size of a work area can be controlled and tuned. Generally, bigger database areas can significantly improve the performance of a particular operator at the cost of higher memory consumption. Optimally, the size of a work area is big enough to accommodate the input data and auxiliary memory structures allocated by its associated SQL operator.

Complex operations such as sort-based operators (order by, group-by, rollup, window function) and joins require a big portion of the run-time area called *work areas* allocated by memory-intensive operators. For example, a sort operator uses a work area (sometimes called the *sort area*) to perform the in-memory sort of a set of rows. Similarly, a hash-join operator uses a work area (also called the *hash area*) to build a hash table from its left input. If the amount of data to be processed by these two operators does not fit into a work area, then the input data is divided into smaller pieces. This allows some data pieces to be processed in memory while the rest are spilled to temporary disk storage to be processed later.

## Program Global Area (PGA) (continued)

If not, response time increases, because part of the input data must be spilled to temporary disk storage. In the extreme case, if the size of a work area is far too small compared to the input data size, multiple passes over the data pieces must be performed. This can dramatically increase the response time of the operator.

## Content of the PGA

The content of PGA memory can be classified as follows:

### Private SQL Area

A private SQL area contains data such as bind information and run-time memory structures. Each session that issues a SQL statement has a private SQL area. Each user that submits the same SQL statement has his or her own private SQL area that uses a single shared SQL area. Thus, many private SQL areas can be associated with the same shared SQL area.

The private SQL area of a cursor is itself divided into two areas whose lifetimes are different:

- The persistent area contains, for example, bind information. It is freed only when the cursor is closed.
- The run-time area is freed when the execution is terminated

The Oracle Database creates the run-time area as the first step of an execute request. For DML statements, the Oracle Database frees the run-time area after the statement has been run. For queries, the Oracle Database frees the run-time area only after all rows are fetched or the query is canceled.

### Cursors and SQL Areas

The user process manages the private SQL areas. The allocation and deallocation of private SQL areas depends on which application tool you are using. The number of private SQL areas that a user process can allocate is limited by the initialization parameter `OPEN_CURSORS`. The default value of this parameter is 50.

A private SQL area exists until the corresponding cursor is closed or the statement handle is freed. Although the Oracle Database frees the run-time area after the statement completes, the persistent area of the private SQL area remains waiting until the cursor is closed. Application developers close all open cursors that will not be used again to free the persistent area and to minimize the amount of memory required for users of the application.

**Session memory** is the memory allocated to hold a session's variables (log-on information) and other information related to the session. For a shared server, the session memory is shared and not private.



# 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.**

ORACLE

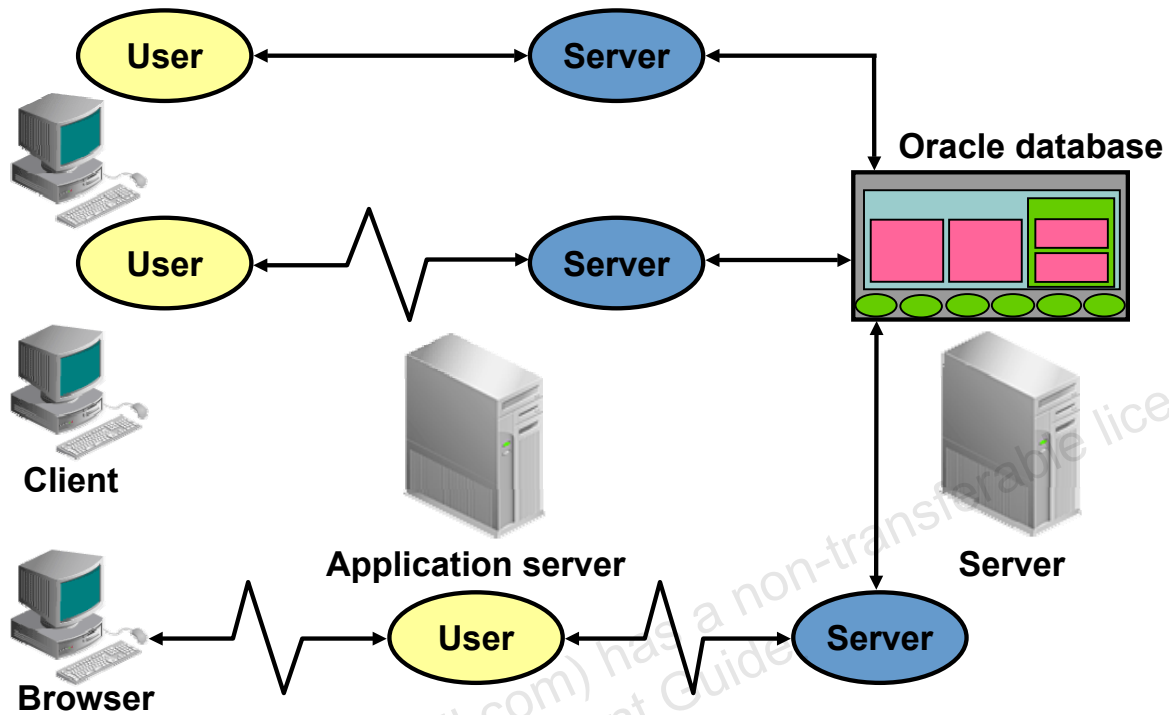
Copyright © 2007, Oracle. All rights reserved.

## Automated SQL Execution Memory (PGA) Management

This feature provides an automatic mode for allocating memory to working areas in the PGA. It simplifies and improves the way PGA memory is allocated and tuned by the Oracle Database. DBAs can use the `PGA_AGGREGATE_TARGET` parameter to specify the total amount of memory that should be allocated to the PGA areas of the instance's sessions. In automatic mode, working areas that are used by memory-intensive operators (sorts and hash-joins) can be automatically and dynamically adjusted.

This feature offers several performance and scalability benefits for DSS workloads or mixed workloads with complex queries. The overall system performance is maximized, and the available memory is allocated more efficiently among queries to optimize both throughput and response time. In particular, the savings that are gained from improved use of memory translate to better throughput at high loads.

## Connecting to an Instance



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Connecting to an Instance

When a user starts a tool such as SQL\*Plus or connects to the database using an application, the application or tool is executed in a *user process*. When a user actually logs on to the Oracle database, a process is created on the computer running the Oracle database. The *listener* on the Oracle database actually establishes the connection and directs the request to an available *server process*. The server process communicates with the Oracle instance on behalf of the user process that runs on the client. The server process executes SQL statements on behalf of the user.

### Connection

A connection is a communication pathway between a user process and an Oracle database. A database user can connect to an Oracle database in one of three ways:

- The user logs on to the machine running the Oracle instance and starts an application or tool that accesses the database on that system. The communication pathway is established using the interprocess communication mechanisms available on the host operating system.

## Connecting to an Instance (continued)

### Connection (continued)

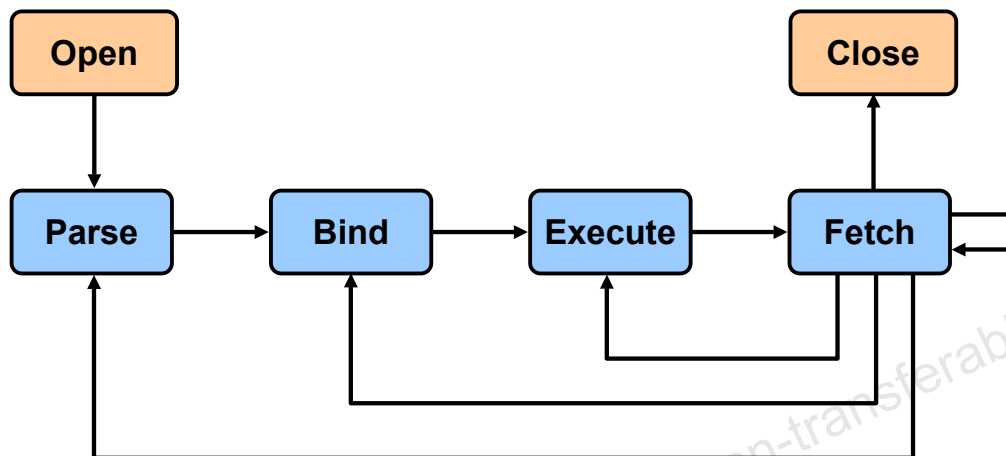
- The user starts the application or tool on a local computer and connects over a network to the computer running the Oracle instance. In this configuration, called client/server, network software is used to communicate between the user and the Oracle database.
- In a three-tiered connection, the user's computer communicates over the network to an application or a network server, which is connected through a network to the machine running the Oracle instance. For example, the user runs a browser on a network computer to use an application residing on an NT server that retrieves data from an Oracle database running on a UNIX host.

### Sessions

A session is a specific connection of a user to an Oracle database. The session starts when the user is validated by the Oracle database, and it ends when the user logs out or when there is an abnormal termination. For a given database user, many concurrent sessions are possible if the user logs on from many tools, applications, or terminals at the same time. Except for some specialized database administration tools, starting a database session requires that the Oracle database be available for use.

Good database connection management offers benefits in minimizing the number of connections, thereby increasing scalability.

# SQL Statement Processing Phases



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Statement Processing Phases

A good understanding of SQL processing is essential for writing optimal SQL statements. In SQL statement processing, there are four important phases: parsing, binding, executing, and fetching.

The reverse arrows indicate processing scenarios (for example, Fetch—(Re)Bind—Execute—Fetch).

The Fetch phase applies only to queries and DML statements with a returning clause.

**Note:** A detailed description of SQL statement processing can be found in *Oracle Database 10g Application Developers Guide: Fundamentals* and *Oracle Database 10g: Concepts*.

# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Parse Phase

Parsing is one of the stages in the processing of a SQL statement. When an application issues a SQL statement, the application makes a parse call to the Oracle database. During the parse call, the Oracle database:

- Checks the statement for syntactic and semantic validity
- Determines whether the process issuing the statement has privileges to run it
- Allocates a private SQL area for the statement

The Oracle database first checks whether there is an existing parsed representation of the statement in the library cache. If so, the user process uses this parsed representation and runs the statement immediately. If not, the Oracle database generates the parsed representation of the statement, and the user process allocates a shared SQL area for the statement in the library cache and stores its parsed representation there.

A parse operation by the Oracle database allocates a shared SQL area for a SQL statement. After a shared SQL area has been allocated for a statement, it can be run repeatedly without being reparsed. Both parse calls and parsing can be expensive relative to execution, so they should be minimized. Ideally, a statement should be parsed once and executed many times rather than reparsing for each execution.

## Parse Phase (continued)

There are two types of parse operations:

- Hard parsing: A SQL statement is submitted for the first time, and no shareable match is found in the shared pool. Hard parses are the most resource-intensive and unscalable, because they perform all the operations involved in a parse.
- Soft parsing: A SQL statement is submitted, and a match *is* found in the shared pool. The match can be the result of a previous execution by another user. The SQL statement is shared, which is good for performance. However, soft parses still require syntax and security checking, which consume system resources.

When bind variables are used properly, more soft parses are possible, thereby reducing hard parses and keeping parsed statements in the library cache for a longer period.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Bind Phase

During the bind phase:

- The Oracle Database checks the statement for references of bind variables
- The Oracle Database assigns or reassigns a value to each variable

When bind variables are used in a statement, the optimizer assumes that cursor sharing is intended and that different invocations should use the same execution plan. If different invocations of the cursor would significantly benefit from different execution plans, then using bind variables may adversely affect the performance of the SQL statement.

# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Execute Phase

The Oracle Database uses the execution plan to identify the required rows of data from the data buffers. Multiple users can share the same execution plan. The Oracle Database performs physical reads or logical reads/writes for DML statements and also sorts the data when needed.

**Note:** Physical reads are disk reads; logical reads are blocks already in memory in the buffer cache. Physical reads are more expensive because they require I/O from disk.

## Fetch Phase

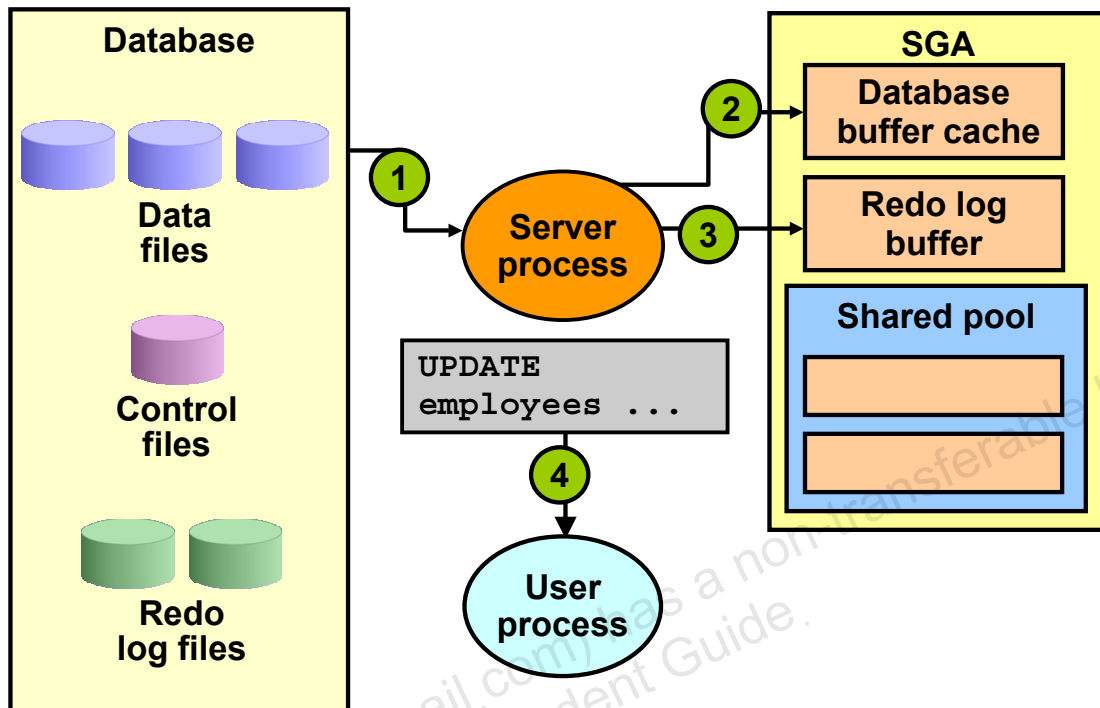
The Oracle Database retrieves rows for a `SELECT` statement during the fetch phase. Each fetch typically retrieves multiple rows, using an array fetch. Each Oracle tool offers its own ways of influencing the array size; in SQL\*Plus, you do so by using the `ARRAYSIZE` setting:

```
SQL> show arraysize
arraysize 15
SQL> set arraysize 1
```

SQL\*Plus processes one row at a time with this setting. The default value is 15. There is no advantage in setting array sizes greater than 100 in SQL\*Plus.



# Processing a DML Statement



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## DML Processing Steps

A data manipulation language (DML) statement requires only two phases of processing:

- Parse is the same as the parse phase used for processing a query.
- Execute requires additional processing to make data changes.

### DML Execute Phase

To execute a DML statement:

1. If the data and rollback blocks are not already in the buffer cache, the server process reads them from the data files into the buffer cache. The server process locks the rows that are to be modified.
2. The server process records the changes to be made to the data buffers as well as the undo changes. These changes are written to the redo log buffer before the in-memory data and rollback buffers are modified. This is called *write-ahead logging*.
3. The rollback buffers contain values of the data before it is modified. The rollback buffers are used to store the before image of the data so that the DML statements can be rolled back if necessary. The data buffers record the new values of the data.
4. The user gets the feedback from the DML operation (such as how many rows were affected by the operation).

## **DML Processing Steps (continued)**

### **DML Execute Phase (continued)**

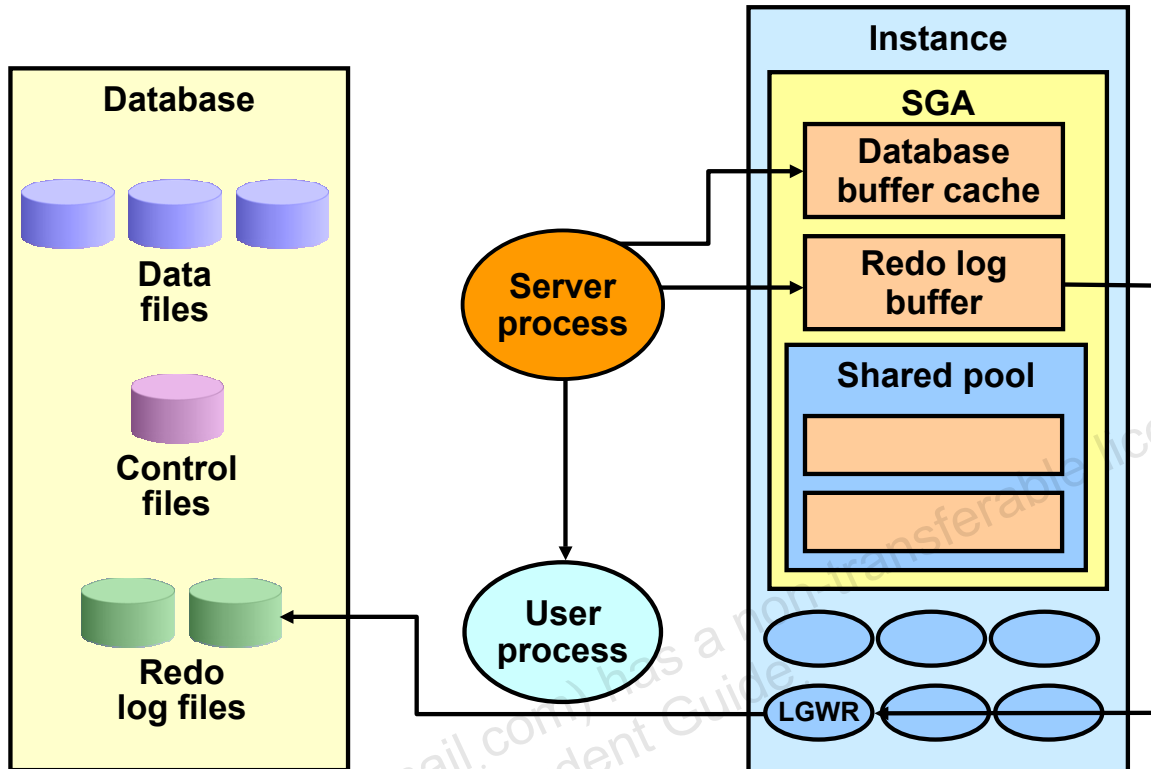
Any changed blocks in the buffer cache are marked as dirty buffers; that is, the buffers are not the same as the corresponding blocks on the disk. These buffers are not immediately written to disk by the Database Writer (DBWR) process. When a transaction is committed, the changes made to the blocks are immediately recorded in the redo logs by the Log Writer process and are later written to disk by DBWR based on an internal algorithm.

The processing of a DELETE or INSERT command uses similar steps. The before image for a DELETE contains the column values in the deleted row, and the before image of an INSERT contains the row location information.

Until a transaction is committed, the changes made to the blocks are only recorded in memory structures and are not written immediately to disk. A computer failure that causes the loss of the SGA can also lose these changes.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# COMMIT Processing



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Fast COMMIT

The Oracle Database uses a Fast COMMIT mechanism that guarantees the committed changes can be recovered in case of instance failure.

### System Change Number

Whenever a transaction commits, the Oracle Database assigns a commit system change number (SCN) to the transaction. The SCN is monotonically incremented and is unique within the database. It is used by the Oracle Database as an internal time stamp to synchronize data and to provide read consistency when data is retrieved from the data files. Using the SCN enables the Oracle Database to perform consistency checks without depending on the date and time of the operating system.

## **Fast COMMIT (continued)**

### **System Change Number (continued)**

When a COMMIT is issued, the following steps are performed:

- The server process places a commit record, along with the SCN, in the redo log buffer.
- The background Log Writer process (LGWR) performs a contiguous write of all the redo log buffer entries up to and including the commit record to the redo log files. After this point, the Oracle Database can guarantee that the changes will not be lost even if there is an instance failure.
- The server process provides feedback to the user process about the completion of the transaction.

DBWR eventually writes the actual changes back to disk based on its own internal timing mechanism.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Functions of the Oracle Query Optimizer

The optimizer is the part of the Oracle Database that creates the execution plan for a SQL statement. The determination of the execution plan is an important step in the processing of any SQL statement and can greatly affect execution time.

The execution plan is a series of operations that are performed in sequence to execute the statement. You have seen the various steps in executing a SQL statement in previous slides. The optimizer considers many factors related to the objects referenced and the conditions specified in the query. The information necessary to the optimizer includes:

- Statistics gathered for the system (I/O, CPU, and so on) as well as schema objects (number of rows, index, and so on)
- Information in the dictionary
- WHERE clause qualifiers
- Hints supplied by the developer

When you use diagnostic tools such as Enterprise Manager, EXPLAIN PLAN, and SQL\*Plus AUTOTRACE, you can see the execution plan that the optimizer chooses.

**Note:** In Oracle Database 10g, the optimizer has two names based on its functionality: the query optimizer and the Automatic Tuning Optimizer.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Top Database Performance Issues

**Bad connection management:** The application connects and disconnects for each database interaction. This problem is common with stateless middleware in application servers. Additionally, simultaneous connections from the same client are also a waste of system and network resources

**Poor use of cursors and the shared pool:** Not reusing cursors results in repeated parses. If bind variables are not used, then there is hard parsing of all SQL statements. This has an order-of-magnitude impact in performance, and it is totally unscalable. Use cursors with bind variables that open the cursor and execute it many times. Be suspicious of applications generating dynamic SQL.

**Bad SQL:** Bad SQL is SQL that uses more resources than appropriate for the application requirement. This can be a decision support systems (DSS) query that runs for more than 24 hours or a query from an online application that takes more than a minute. SQL that consumes significant system resources should be investigated for potential improvement. ADDM identifies high-load SQL, and the SQL Tuning Advisor can be used to provide recommendations for improvement.

## Top Database Performance Issues (continued)

**Use of nonstandard initialization parameters:** These might have been implemented based on poor advice or incorrect assumptions. Most systems will give acceptable performance using only the set of basic parameters. In particular, parameters associated with `SPIN_COUNT` on latches and undocumented optimizer features can cause a great deal of problems that can require considerable investigation

**I/O issues:** If you configure your database to use multiple disks by disk space and not I/O bandwidth, then there will be excessive I/O to certain disks and little I/O to others. Frequently and simultaneously accessed objects (a table and its index) should be designed to be stored over different disks.

**Long full-table scans:** Long full-table scans for high-volume or interactive online operations could indicate poor transaction design, missing indexes, or poor SQL optimization.

**In-disk sorting:** In-disk sorts for online operations could indicate poor transaction design, missing indexes, or poor SQL optimization. Disk sorts, by nature, are I/O-intensive and unscalable.

**High amounts of recursive SQL:** Large amounts of recursive SQL executed by SYS could indicate space management activities, such as extent allocations, taking place. This is unscalable and impacts user response time. Recursive SQL executed under another user ID is probably SQL and PL/SQL, and this is not a problem.

**Schema errors and optimizer problems:** In many cases, an application uses too many resources because the schema owning the tables has not been successfully migrated from the development environment or from an older implementation. Examples of this are missing indexes or incorrect statistics. These errors can lead to suboptimal execution plans and poor interactive user performance. When migrating applications of known performance, you should export the schema statistics to maintain plan stability by using the `DBMS_STATS` package.

Likewise, optimizer parameters set in the initialization parameter file can override proven optimal execution plans. For these reasons, schemas, schema statistics, and optimizer settings should be managed together as a group to ensure consistency of performance.

# Practice 1: Overview

**This practice covers the following topics:**

- **Using EM**

ORACLE

Copyright © 2007, Oracle. All rights reserved.



# Summary

**In this lesson, you should have learned about the Oracle Database architecture and various components that require tuning.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

In this lesson, you have seen a brief overview of the Oracle Database architecture and different components that are involved in the tuning process.

swati bhatia (swati.june@gmail.com) has a non-transferable license to  
use this Student Guide.

## 2 Following a Tuning Methodology

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Objectives

This lesson introduces you to tuning methodologies. By applying a tuning methodology, you can optimize performance and improve response time and scalability. Tuning is an ongoing process in the life cycle of a project that must start in the development phase and continue through the maintenance phase.

This lesson shows you the different steps in tuning SQL and the various methods available for each step. It also introduces different tools that can be used in each step. Subsequent lessons discuss each of the steps and tools in greater detail.

# Performance Problems

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Performance Problems

Performance problems occur when a task takes longer to perform than the time allowed and uses excessive resources. Usually problems occurs when the supply of a resource is insufficient to meet the demand. The resource may be a physical resource (such as available memory buffers that store data blocks) or a nontangible resource (such as a lock). Possible causes of performance problems are the following.

### Inadequate Consumable Resources

Sometimes a resource may simply be inadequate to meet the need under any circumstances. For example, if you want a function to complete in less than one second, a network with a message turnaround time of two seconds will never meet the target.

If the limiting factor is a consumable resource (such as CPU power), all the users of that resource are affected.

### High-Load SQL

In some cases, a single SQL statement may take up a large amount of resources, thereby affecting the performance of other statements.

### Contention

If many users are trying to update the same set of tables, contention due to locking might be a problem.

# Factors to Be Managed

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Factors to Be Managed

Performance management can be divided into the following four areas. Although the areas are separate, they are also interdependent and require different skill sets.

**Schema tuning** deals with the physical structure of the data. If an application has inadequate or inappropriate data design, then tuning the physical allocation, providing indexes, or rewriting programs will not overcome the problem.

**Application tuning** deals with such business requirements as 24/7 availability, OLAP, OLTP, and so on as well as the program modules or applications that implement the functions. Tuning the procedural code for the type of application and tuning the embedded SQL statements are also included in this factor. If an application is well designed, it may still perform badly. A common reason for this is badly written SQL.

**Instance tuning** deals with the configuration of the Oracle server for memory utilization.

**Database tuning** deals with managing the physical arrangement of data on the disk.

## Factors to Be Managed (continued)

**User expectations:** Usually users expect consistent performance on all applications. However, they may accept certain applications (such as OLAP operations) as slower if the project team builds realistic user expectations. An application may include messages to warn operators that they are requesting resource-intensive operations. The best time to do this is before the design and build phases and as part of the transition phase.

**Hardware and network tuning** deals with performance issues arising from the CPU and from network traffic on all machines supporting the application. The main hardware components are:

- **CPU:** There can be one or more CPUs, and they can vary in processing power from simple CPUs found in hand-held devices to high-powered server CPUs. Sizing of other hardware components is usually a multiple of the CPUs on the system.
- **Memory:** Databases require considerable amounts of memory to cache data and avoid time-consuming disk access.
- **I/O subsystem:** The I/O subsystem can vary between the hard disk on a client PC and high-performance disk arrays. Disk arrays can perform thousands of I/Os each second and provide availability through redundancy in terms of multiple I/O paths and hot pluggable mirrored disks.
- **Network:** The primary concerns with network specifications are bandwidth (volume) and latency (speed).

**Note:** This course concentrates on schema and SQL statement tuning. Instance, hardware, and database tuning are covered in other courses and in literature for database administrators.

# Tuning Goals

- **Reduce the response time.**
- **Reduce resource usage.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Tuning Goals

The goal of tuning a system is either to reduce the response time for end users or to reduce the resources used to process the same work. You can accomplish both of these objectives in several ways:

- **Reduce the workload:** SQL tuning commonly involves finding more efficient ways to process the same workload. It is possible to change the execution plan of the statement without altering the functionality to reduce the resource consumption. Two examples of how resource usage can be reduced are:
  1. If a query needs to access a small percentage of data in the table, this can be executed more efficiently by using an index. By creating such an index, you reduce the amount of resources used.
  2. If a user is looking at the first 20 rows of the 10,000 rows returned in a specific sort order, and if the query (and sort order) can be satisfied by an index, then the user does not need to access and sort the 10,000 rows to see the first 20 rows.



## Tuning Goals (continued)

- **Balance the workload:** Peak usage is usually during the day (when OLTP users are connected to the system), with low usage at night. If noncritical reports and batches can be scheduled to run at night, then daytime resources are freed up for other programs.
- **Parallelize the workload:** Queries that access large amounts of data (typically data warehouse queries) often can be parallelized. This is extremely useful for reducing the response time in low-concurrency data warehouses.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Overview of SQL Tuning

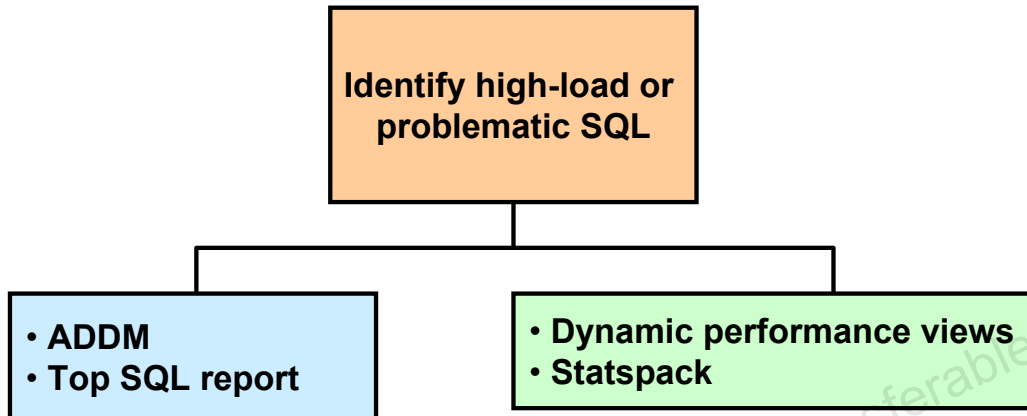
The slide lists the steps in the SQL tuning process.

### Step 1: Identify causes of poor performance.

Some of the causes for poor performance in SQL are:

- **The volume of data being accessed.** If more full-table scans are performed on large tables, then the time taken to retrieve the data can be very long. This can also cause a large number of data blocks to be read from disk and loaded into memory, filling up the buffer cache and prematurely aging other data. This can cause the I/O on the system to increase, resulting in poor performance.
- **Poorly written application code.** If the code used in applications uses full-table scans or does not use necessary indexes, then the optimizer is unable to use the best execution plan. This slows query operations.
- **OPTIMIZER\_MODE setting.** If the appropriate optimizer mode settings are not set, then the optimizer may make wrong decisions and use slower execution plans.
- **Optimizer statistics.** Usually the database automatically gathers statistics and keeps them current by monitoring for stale and missing statistics. However, if objects change dramatically between the automatic statistics-gathering intervals, this can cause otherwise well-written statements to slow down. In such cases, the manual gathering of statistics may be required.

# Identifying High-Load SQL



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Identifying High-Load SQL

The next step in the tuning process is identifying the SQL statements that need tuning. These are usually the statements that are frequently used and use a large amount of system resources during execution. Oracle Database 10g offers many features for identifying these statements, such as Automatic Database Diagnostic Monitor (ADDM) reports, the Top SQL pages of Enterprise Manager, and the V\$SQL dynamic performance views. Statspack can also be used in the standard edition of the database.

Once you have identified the problematic SQL, you can choose to tune the SQL either manually or automatically. Both these methods are described in the next few slides.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Manual Tuning

The steps involved in tuning SQL manually are listed in the slide. The following slides cover these steps in detail.

## Gather Information About Referenced Objects

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Gather Information About Referenced Objects

The tuning process begins by determining the structure of the underlying tables and indexes. The information gathered includes the following:

1. Complete SQL text from V\$SQLTEXT
2. Structure of the tables referenced in the SQL statement, usually by describing the table in SQL\*Plus
3. Definitions of any indexes (columns, column orderings) and whether the indexes are unique or non-unique
4. Optimizer statistics for the segments (including the number of rows in each table and the selectivity of the index columns), including the date when the segments were last analyzed
5. Definitions of any views referred to in the SQL statement
6. Repeat steps 2, 3, and 4 for any tables referenced in the view definitions found in step 5.
7. Optimizer plan for the SQL statement (from either EXPLAIN PLAN or the TKPROF output)
8. Any previous optimizer plans for that SQL statement

# Gathering Optimizer Statistics

- **Gather statistics for all tables.**
- **Gather new statistics when existing statistics become stale.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Gathering Optimizer Statistics

Statistics must be regularly gathered on database objects because those database objects are modified over time. In order to determine whether or not a given database object needs new database statistics, Oracle provides a table-monitoring facility. This monitoring is enabled by default when `STATISTICS_LEVEL` is set to `TYPICAL` or `ALL`. Monitoring tracks the approximate number of `INSERTs`, `UPDATEs`, and `DELETEs` for that table, as well as whether the table has been truncated, since the last time statistics were gathered. The information about changes of tables can be viewed in the `USER_TAB_MODIFICATIONS` view. Following a data modification, there may be a delay of a few minutes while the system propagates the information to this view. Use the `DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO` procedure to immediately reflect the outstanding monitored information kept in the memory.

In Oracle Database 10g, the statistics gathering is done automatically by default. The Oracle Database gathers statistics on all database objects automatically and maintains those statistics in a regularly scheduled maintenance job. Automated statistics collection eliminates many of the manual tasks associated with managing the query optimizer, and significantly reduces the chances of getting poor execution plans because of missing or stale statistics.

## Gathering Optimizer Statistics (continued)

Optimizer statistics are automatically gathered with GATHER\_STATS\_JOB. This job gathers statistics on all objects in the database that have missing statistics or stale statistics.

This job is created automatically at database creation time and is managed by the Scheduler. The Scheduler runs this job when the maintenance window is opened. By default, the maintenance window opens every night from 10:00 p.m. to 6:00 a.m. and all day on weekends. GATHER\_STATS\_JOB continues until it finishes, even if it exceeds the allocated time for the maintenance window. The default behavior of the maintenance window can be changed.

If these statistics have not been gathered, or if the statistics are no longer representative of the data stored in the database, the optimizer does not have sufficient information to generate the best plan.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Reviewing the Execution Plan

Tuning a SQL statement requires that you know your data. The factors affecting an execution plan are:

- Data structure and volume
- Indexes
- PRIMARY keys or UNIQUE keys
- Views
- Data skews
- Cardinality
- Selectivity

When examining the optimizer execution plan, look for the following factors. If any of these conditions are not optimal, then consider restructuring the SQL statement or the indexes available on the tables.

- The plan is such that the driving table has the best filter.
- The join order in each step means that the fewest number of rows are being returned to the next step (that is, the join order should reflect, where possible, going to the best not-yet-used filters).



## Reviewing the Execution Plan (continued)

- The join method is appropriate for the number of rows being returned. For example, nested loop joins through indexes may not be optimal when many rows are being returned.
- Views are used efficiently. Look at the `SELECT` list to see whether access to the view is necessary.
- There are unintentional Cartesian products (even with small tables).

In an OLTP environment, the goal is to drive from the table that has the most selective filter. This means that there are fewer rows passed to the next step. If the next step is a join, then this means that fewer rows are joined. Check to see whether the access paths are optimal.

### Each table is being accessed efficiently:

- Consider the predicates in the SQL statement and the number of rows in the table. Look for suspicious activity, such as full table scans on tables with a large number of rows with predicates in the `WHERE` clause. Determine why an index is not used for such a selective predicate.
- A full table scan does not mean inefficiency. It might be more efficient to perform a full table scan on a small table, or to perform a full table scan to leverage a better join method (for example, `hash_join`) for the number of rows returned.

### Tools for Obtaining Execution Plans

There are many tools available to review the execution plans. The performance pages in Enterprise Manager provide execution plans for Top SQL. The `EXPLAIN PLAN` command provides the execution plan for SQL statements. You can also use the SQL Trace facility to generate execution plans at both the session and system level. These plans can then be interpreted using the `TKPROF` command line utility.

**Note:** Using SQL Trace at the system level causes significant performance overhead and is not recommended.

The SQL\*Plus and iSQL\*Plus tools also have an `AUTOTRACE` facility that allows you to view execution plans. Using these tools, you can view the execution plan for a SQL statement without actually executing it.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Restructuring the SQL Statements

Rewriting an inefficient SQL statement is often easier than repairing it. If you understand the purpose of a given statement, then you might be able to quickly and easily write a new statement that meets the requirement.

### **Compose predicates by using AND and =**

To improve SQL efficiency, use equijoins whenever possible. Statements that perform equijoins on untransformed column values are the easiest to tune.

### **Avoid transformed columns in the WHERE clause**

Do not use a transformed column (a column with SQL functions) in predicate clauses or WHERE clauses. Any expression using a column (such as a function having the column as its argument) causes the optimizer to ignore the possibility of using an index on that column, even a unique index, unless there is a function-based index defined that can be used. For example "WHERE UPPER(last\_name) " will not allow use of an index on the LAST\_NAME column.

### **Beware of implicit type conversions**

These expressions prevent the optimizer from assigning valid cardinality or selectivity estimates and can in turn affect the overall plan and the join method.

## Restructuring the SQL Statements (continued)

For example:

### Write separate SQL statements for specific tasks

SQL is not a procedural language. Using one piece of SQL to do many different things usually results in a less-than-optimal result for each task. If you want SQL to accomplish different goals, then write different statements rather than writing one statement to accomplish those goals depending on the parameters you give it.

It is always better to write separate SQL statements for different tasks, but if you must use one SQL statement, then you can make a very complex statement slightly less complex by using the UNION ALL operator.

Optimization (determining the execution plan) takes place before the database knows what values will be substituted into the query. An execution plan cannot, therefore, depend on what those values are.

### Using EXISTS versus IN for subqueries

In certain circumstances, it is better to use IN rather than EXISTS. In general, if the selective predicate is in the subquery, use IN. If the selective predicate is in the parent query, use EXISTS. Sometimes, Oracle can rewrite a subquery when used with an IN clause to take advantage of selectivity specified in the subquery. This is most beneficial when the most selective filter appears in the subquery and there are indexes on the join columns. Conversely, using EXISTS is beneficial when the most selective filter is in the parent query. This allows the selective predicates in the parent query to be applied before filtering the rows against the EXISTS criteria.

### Control the access path and join order with hints

You can influence the optimizer's choices by setting the optimizer approach and goal, and by gathering comprehensive statistics. Sometimes, the application designer, who has more information about a particular application's data than is available to the optimizer, can choose a more effective way to execute a SQL statement. You can use hints in SQL statements to specify how a statement should be executed.

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

- Avoid a full-table scan if it is more efficient to get the required rows through an index.
- Avoid using an index that fetches a large number of rows from the driving table if you can instead use another index that fetches fewer rows.
- Choose the join order to join fewer rows to tables later in the join order.

# Restructuring the Indexes

- **Remove unnecessary indexes to speed up 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Restructuring the Indexes

There is often a beneficial impact on performance from the restructuring of indexes. This can involve:

- Removing nonselective indexes to speed up the DML
- Indexing performance-critical access paths
- Considering the reordering of columns in existing concatenated indexes so that the first column in the index is the one most frequently used in WHERE clauses
- Adding columns to the index to improve selectivity
- The Oracle database has many different index options that (when used correctly) can significantly improve performance. These are discussed in later lessons.
- Considering the use of index-organized tables

Indexes must be created with caution. Application developers sometimes think that performance improves if they create more indexes. However, indiscriminately creating indexes will probably hamper performance.

The tools that can be used to identify which indexes should be created or dropped are the SQL Access Advisor and SQL Tuning Advisor. These are discussed later in this lesson.

You can also determine whether an index is actually being used and whether it is beneficial by creating the index and verifying its execution plan, and determining whether the cost of execution benefits from the index.

# Maintaining Execution Plans over Time

- **Stored outlines**
- **Stored statistics**
- **Locking statistics**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Maintaining Execution Plans over Time

You can maintain the existing execution plan of SQL statements over time using either stored statistics or stored SQL execution plans. Storing optimizer statistics for tables applies to all SQL statements that refer to those tables. Storing an execution plan (that is, plan stability) maintains the plan for a single SQL statement. If both statistics and a stored plan are available for a SQL statement, then the optimizer uses the stored plan.

With Oracle Database 10g, you have the ability to lock statistics for a volatile table at an optimal point in time and then allow the optimizer to reuse those statistics. SQL profiles enable tuning of queries without requiring any syntactical changes, thereby providing a unique database solution to tune the SQL statements that are embedded in packaged applications.

# Automatic SQL Tuning

- **Automatic SQL tuning facilitates these steps:**
  - Gather information about 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Automatic SQL Tuning

Manual SQL tuning is a complex process that presents many challenges. It requires expertise in several areas, is very time consuming, and requires an intimate knowledge of the schema structures and the data usage model of the application. All these factors make manual SQL tuning a challenging and resource intensive task that is ultimately very expensive for businesses.

Automatic SQL tuning, available in Oracle Database 10g, makes the SQL tuning process easier using analytical processes. All of the steps discussed in earlier pages can be completed using these kinds of analysis. These processes are performed by the database engine's query optimizer.

Four types of analysis are performed in automatic SQL tuning:

- **Statistics analysis:** The query optimizer needs up-to-date object statistics to generate good execution plans. In this type of analysis, objects with stale or missing statistics are identified and appropriate recommendations are made to remedy the problem.
- **SQL profiling:** This is a new feature introduced in Oracle Database 10g that revolutionizes the approach to SQL tuning. Traditional SQL tuning involves manual manipulation of application code using optimizer hints.

## Automatic SQL Tuning (continued)

- SQL profiling eliminates the need for this manual process and tunes the SQL statements without requiring any change to the application code. This ability to tune SQL without changing the application code also helps solve the problem of tuning packaged applications. Packaged application users now no longer need to log a bug with the application vendor and wait for several weeks or months to obtain a code fix for tuning the statement. With SQL profiling, tuning is automatic and immediate.
- **Access path analysis:** Indexes can tremendously enhance performance of a SQL statement by reducing the need for full table scans. Effective indexing is, therefore, a common tuning technique. In this type of analysis, new indexes that can significantly enhance query performance are identified and recommended.
- **SQL structure analysis:** Problems with the structure of SQL statements can lead to poor performance. These could be syntactic, semantic, or design problems with the statement. In this type of analysis, relevant suggestions are made to restructure the SQL statements for improved performance.

# Automatic Tuning Mechanisms

**You can perform automatic SQL tuning using:**

- **SQL Tuning Advisor**
- **SQL Access Advisor**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Automatic Tuning Mechanisms

Automatic SQL tuning offers a comprehensive SQL tuning solution through the use of two new advisors that are part of Oracle Tuning Pack 10g: SQL Tuning Advisor and SQL Access Advisor. These two advisors provide a complete solution for tuning applications and automate all manual tuning techniques currently practiced. Together they form the core of Oracle's automatic SQL tuning solution.



# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Tuning Advisor

The SQL Tuning Advisor takes one or more SQL statements as input and invokes the automatic SQL tuning process on it. The output of the SQL Tuning Advisor is in the form of recommendations, along with a rationale for each recommendation and its expected benefit. The recommendation relates to collection of statistics on objects, creation of new indexes, restructuring of the SQL statement, or creation of a SQL Profile. The SQL Tuning Advisor is designed to accept input from several SQL sources, such as Automatic Database Diagnostic Monitor (ADDM), Automatic Workload Repository (AWR), cursor cache, and custom SQL as defined by the user. SQL statements from these input sources are typically first loaded in a new object called SQL Tuning Set, which is then submitted as input to the advisor.

After analyzing the SQL statements, the SQL Tuning Advisor provides advice on improving the performance of the SQL statements, the rationale for the proposed advice, estimated performance benefit, and the command to implement the advice. You simply choose whether or not to accept the recommendations to optimize the SQL statements.

# SQL Access Advisor

**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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Access Advisor

The design of the database schema can significantly impact overall application performance. SQL Access Advisor is a new solution in Oracle Database 10g that provides comprehensive advice on how to optimize schema design in order to maximize application performance. The SQL Access Advisor accepts input from all possible sources of interest, such as the cursor cache, the Automatic Workload Repository (AWR), any user-defined workload. It will even generate a hypothetical workload if a schema contains dimensions or primary/foreign key relationships. It comprehensively analyzes the entire workload and provides recommendations to create new indexes if required, drop any unused indexes, and create new materialized views and materialized view logs. The analysis considers the cost of INSERT, UPDATE, and DELETE operations in addition to the queries. The recommendation generated by the SQL Access Advisor is accompanied by a quantifiable measure of expected performance gain as well as the scripts needed to implement the recommendation.

The SQL Access Advisor takes the mystery out of the access design process. It tells the user exactly what types of indexes and materialized views are required to maximize application performance. By automating this very critical function, SQL Access Advisor obviates the need for the error-prone, lengthy, and expensive manual tuning process. It is fast, precise, and easy to use.

# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

Applying a tuning methodology to your design and efforts optimizes performance. Tuning SQL is a major component of a tuning methodology.

After completing this lesson, you should be able to:

- State the procedural steps in managing performance
- Describe the causes of performance problems
- Identify the main system areas that you can address by using the tuning process
- Describe the tuning methodology
- Explain the advantage of following the steps of the tuning methodology in their proper sequence
- List the tuning steps that are the responsibility of the application developer

Anyone involved in tuning should follow a tuning methodology to achieve maximum performance. Tuning SQL statements is an important step that is least expensive when performed at the proper point in the overall tuning methodology.

# 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

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Summary (continued)

Tuning SQL statements involves the effective use of analysis tools, SQL tuning techniques, and the optimizer. In addition, tuning the schema can create additional access paths for the optimizer, such as the use of an index.

# 3

## Designing and Developing for Performance

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# Objectives

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Objectives

After completing this lesson, you should be able to:

- Understand scalability
- Tune for single-user versus multiuser environments
- Balance multiple applications
- Determine OLAP requirements versus OLTP requirements
- Employ proactive tuning methodologies
- List application design principles
- Use dynamic SQL in application development
- List the benefits of bind variables and writing SQL to share cursors
- Describe the steps for implementing and testing
- List the trends in application development
- Compare rollout strategies
- Use a performance checklist

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Understanding Scalability

Scalability is a system's ability to process additional workload, with a proportional increase in system resource use. In other words, if you double the workload in a scalable system, then the system must use twice as many system resources. However, while some resource utilization may increase in proportion to the scale up, other resources may scale differently depending on the resource in question and the application use.

Examples of bad scalability due to resource conflicts include the following:

- Applications requiring significant concurrency management as user populations increase
- Increased data consistency workload
- Increased operating system workload
- Transactions requiring increases in data access as data volumes increase
- Poor SQL and index design, resulting in a higher number of logical I/Os for the same number of rows returned
- Reduced availability because database objects take longer to maintain

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Scalability with Application Design, Implementation, and Configuration

Poor application design, implementation, and configuration have a significant impact on scalability.

However, the design is not the only problem. The physical implementation of the application can be the weak link, as in the following examples:

- Systems can move to production environments with poorly written SQL that are causing high I/O.
- Infrequent transaction COMMITs or ROLLBACKs can cause locks on resources.
- The production environment could use different execution plans than those generated in testing.
- Memory-intensive applications that allocate a large amount of memory without much thought for freeing the memory can cause excessive memory leakage.
- Inefficient memory usage and memory leaks put a high stress on the operating virtual memory subsystem. This affects performance and availability.



# Configuring the Appropriate System Architecture for Your Requirements

- **Interactive applications (OLTP)**
- **Process-driven applications (OLAP)**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Configuring the Appropriate System Architecture for Your Requirements

Configuring the initial system architecture is a largely iterative process. Architects must satisfy the system requirements within budget and schedule constraints. If the system requires interactive users to transact business or make decisions based on the contents of a database, then user requirements drive the architecture. If there are few interactive users on the system, then the architecture is process driven. Process-driven applications draw from the skill sets that are used in both user-based applications and data warehousing.

Some of the things to consider are:

- Number of users (Web application versus few users)
- User interaction method (query versus transaction)
- User location (LAN versus WAN)
- Network speed
- Amount of read-only data (query versus DML)
- User response-time requirement
- 24/7 operation (high availability)
- Real-time changes (OLTP)
- Size of database

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Proactive Tuning Methodology

Tuning usually implies fixing a performance problem. However, tuning should be part of the life cycle of an application, through the analysis, design, coding, production, and maintenance stages. The tuning phase is frequently left until the system is in production. At such a time, tuning becomes a reactive exercise, where the most important bottleneck is identified and fixed.

The slide lists some of the things that affect performance and that should be tuned proactively instead of reactively. These are discussed in more detail in the following slides.

# Simplicity In Application Design

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Simplicity In Application Design

Applications are no different than any other designed and engineered product. If the design looks right, then it probably is right. This principle should always be kept in mind when building applications. Consider some of the following design issues:

- If the table design is so complicated that nobody can fully understand it, the table is probably designed badly.
- If SQL statements are so long and involved that it would be impossible for any optimizer to effectively optimize it in real time, then there is probably a bad statement, underlying transaction, or table design.
- If there are many indexes on a table and the same columns are repeatedly indexed, there is probably a bad index design.
- If queries are submitted without suitable qualification (WHERE clause) for rapid response for online users, there is probably a bad user interface or transaction design.

# Data Modeling

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Data Modeling

Data modeling is important in successful relational application design. This should be done in a way that quickly and accurately represents the business practices. Apply your greatest modeling efforts to those entities affected by the most frequent business transactions. Use of modeling tools can then rapidly generate schema definitions and can be useful when a fast prototype is required.

Normalizing data prevents duplication. When data is normalized, you have a clear picture of the keys and relationships. It is then easier to perform the next step of creating tables, constraints, and indexes. A good data model ultimately means that your queries will be written more efficiently.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Table Design

Table design is largely a compromise between flexibility and performance of core transactions. To keep the database flexible and able to accommodate unforeseen workloads, the table design should be very similar to the data model, and it should be normalized to at least third normal form. However, certain core transactions required by users can require selective denormalization for performance purposes.

Use the features supplied with Oracle Database 10g to simplify table design for performance, such as storing tables prejoined in clusters, the addition of derived columns, aggregate values, and using materialized views. Additionally, create check constraints and columns with default values to prevent bad data from getting into the tables.

Design should be focused on business-critical tables so that good performance can be achieved in areas that are the most used. For noncritical tables, shortcuts in design can be adopted to enable a more rapid application development. If, however, a noncore table becomes a performance problem during prototyping and testing, then remedial design efforts should be applied immediately.

# Index Design

- **Index keys**
  - **Primary key**
  - **Unique key**
  - **Foreign keys**
- **Index data that is frequently queried**
- **Use SQL as a guide to index design**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Index Design

Index design is also a largely iterative process, based on the SQL that is generated by application designers. However, it is possible to make a sensible start by building indexes that enforce foreign key constraints (to reduce response time on joins between primary key tables and foreign key tables) and creating indexes on frequently accessed data, such as a person's name. Primary keys and unique keys are automatically indexed. As the application evolves and testing is performed on realistic sizes of data, certain queries will need performance improvements, for which building a better index is a good solution.

The following indexing design ideas should be considered when building a new index.

### Appending Columns to an Index or Using Index-Organized Tables

One of the easiest ways to speed up a query is to reduce the number of logical I/Os by eliminating a table scan from the execution plan. This can be done by appending to the index all columns referenced by the query. These columns are the select list columns and any required join or sort columns. This technique is particularly useful in speeding up an online application's response times when time-consuming I/Os are reduced. This is best applied when testing the application with properly sized data for the first time.

## Index Design (continued)

### Using a Different Index Type

The most aggressive form of this technique is to build an index-organized table (IOT).

There are several index types available, and each index has benefits for certain situations.

- **B\*tree index:** This is the standard index type and the default when other types are not specified. Used as concatenated indexes, B-tree indexes can be used to retrieve data sorted by the index columns.
- **Bitmap index:** This is suitable for low-cardinality data. Combining bitmap indexes on nonselective columns allows efficient AND and OR operations with a great number of row IDs with minimal I/O. Bitmap indexes are particularly efficient in queries with COUNT ( ) , because the query can be satisfied within the index.
- **Function-based index:** Function-based indexes are particularly useful when querying on composite columns to produce a derived result or to overcome limitations in the way data is stored in the database. For example, you may search for orders in which an expression such as *(sales price – discount) \* quantity* exceeds a given value (assuming these are columns in the table). Another example is to apply the function to the data to allow case-insensitive searches (UPPER or LOWER).
- **Reverse key index:** These indexes are excellent for insert performance, but they are limited in that they cannot be used for index range scans. Use of sequences, or time stamps, to generate key values that are indexed themselves can lead to database hotspot problems due to a monotonically growing key, which affects response time and throughput. Generating keys that insert over the full range of the index results in a well-balanced index that is more scalable and space efficient. You can achieve this by using a reverse key index or using a cycling sequence to prefix and sequence values.

### Cost of an Index

Building and maintaining an index structure can be expensive, and it can consume resources such as disk space, CPU, and I/O capacity. You must ensure that the benefits of any index outweigh the negatives of index maintenance.

Use this simple estimation guide for the cost of index maintenance: Each index maintained by an INSERT, DELETE, or UPDATE of the indexed keys requires about three times as many resources as the actual DML operation on the table. What this means is that if you INSERT into a table with three indexes, then it will be approximately 10 times slower than an INSERT into a table with no indexes. For DML, and particularly for INSERT-heavy applications, the index design should be seriously reviewed, which might require a compromise between the query and INSERT performance.

### Ordering Columns in an Index

You should be flexible in defining any rules for index building. Order columns that have the most selectivity first. This method is the most commonly used because it provides the fastest access with minimal I/O to the actual row IDs required. This technique is used mainly for primary keys and for very selective range scans. You can also place the most frequently used column first so that the index is beneficial to several queries.

# Using Views

- **Simplifies application design**
- **Is transparent to the end user**
- **Can cause suboptimal execution plans**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Using Views

Views can speed up and simplify application design. A simple view definition can mask data model complexity from the programmers whose priorities are to retrieve, display, collect, and store data.

However, while views provide clean programming interfaces, they can cause suboptimal, resource-intensive queries when nested too deeply. The worst type of view use is creating joins on views that reference other views, which in turn reference other views. In many cases, developers can satisfy the query directly from the table without using a view. Because of their inherent properties, views usually make it difficult for the optimizer to generate the optimal execution plan.



# SQL Execution Efficiency

- **Good database connectivity**
- **Using cursors**
- **Minimizing parsing**
- **Using bind variables**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Execution Efficiency

An application that is designed for SQL execution efficiency must support the following characteristics:

**Good database connection management:** Connecting to the database is an expensive operation. Therefore, the number of concurrent connections from each client to the database should be minimized. However, in a Web-based or multitiered application where application servers are used to multiplex database connections to users, this can be difficult. Design efforts should ensure that database connections are pooled and are not reestablished for each user request.

**Good cursor usage and management:** Parsing should be minimized as much as possible. Applications should be designed to parse SQL statements once and execute them many times.

Use bind variables to represent the parts of the query that change from execution to execution. If this is not done, then the SQL statement is likely to be parsed once and never reused by other users. To ensure that SQL is shared, use bind variables and do not use string literals with SQL statements.

# Importance of Sharing Cursors

- **Reduces parsing**
- **Dynamically adjusts memory**
- **Improves memory usage**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Importance of Sharing Cursors

When a SQL statement is found in the shared SQL area, then the parse phase is shortened and the existing cursor is used. The following are the main benefits of sharing cursors:

- Sharing cursors reduces parsing and saves time.
- Memory dynamically adjusts to the SQL being executed because the same area of library cache can be reused.
- Memory usage may improve dramatically, even for tools that store SQL within the application.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Writing SQL to Share Cursors

In order for Oracle to share cursors, the code must be written exactly the same. You should therefore develop coding conventions for SQL statements in ad hoc queries, SQL scripts, and Oracle Call Interface (OCI) calls.

### Use generic shared code:

- Write and store procedures that can be shared across applications.
- Use database triggers.
- Write referenced triggers and procedures when using application development tools.
- Write library routines and procedures in other environments.

### Write to format standards:

- Develop format standards for all statements, including those in PL/SQL code.
- Develop rules for use of uppercase and lowercase characters.
- Develop rules for use of white space (spaces, tabs, returns).
- Develop rules for use of comments (preferably keeping them out of the SQL statements themselves).
- Use the same names to refer to identical database objects.

# Controlling Shared Cursors

**The `CURSOR_SHARING` initialization parameter can be set to:**

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Controlling Shared Cursors

One of the first stages of parsing is to compare the text of the statement with existing statements in the shared pool to see if the statement can be shared. If the statement differs textually in any way, then the Oracle Database does not share the statement. Exceptions to this are possible using the `CURSOR_SHARING` parameter. Setting `CURSOR_SHARING` to `EXACT` allows SQL statements to share the SQL area only when their texts match exactly. This is the default behavior. Using this setting, similar statements cannot be shared; only textually exact statements can be shared.

Setting `CURSOR_SHARING` to either `SIMILAR` or `FORCE` allows similar statements to share SQL. The difference between `SIMILAR` and `FORCE` is that `SIMILAR` forces similar statements to share the SQL area without deteriorating execution plans. Setting `CURSOR_SHARING` to `FORCE` forces similar statements to share the executable SQL area, potentially deteriorating execution plans. Therefore, `SIMILAR` and `FORCE` should be used as a last resort, when the risk of suboptimal plans is outweighed by the improvements in cursor sharing.

### Example:

```
SQL> ALTER SESSION SET CURSOR_SHARING = 'EXACT' ;
```

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Performance Checklist

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

# Summary

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

In this lesson, you should have learned how to:

- Understand scalability
- Tune for single-user versus multiuser environments
- Balance multiple applications
- Determine OLAP requirements versus OLTP requirements
- Employ proactive tuning methodologies
- List application design principles
- Use dynamic SQL in application development
- List the benefits of bind variables and writing SQL to share cursors
- Describe steps in implementing and testing
- List the trends in application development
- Compare rollout strategies
- Use a performance checklist

## Practice Lesson 3

**This practice covers understanding the use of shared cursors**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

swati bhatia (swati.june@gmail.com) has a non-transferable license to  
use this Student Guide.



# 4

## Introduction to the Optimizer

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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**

Copyright © 2007, Oracle. All rights reserved.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Oracle Optimizer

The optimizer is the part of the Oracle Server that creates the execution plan for a SQL statement. An execution plan is a series of operations that are performed to execute the statement. The optimizer uses various pieces of information to determine the best path:

- Hints supplied by the developer
- Statistics
- Information in the dictionary
- WHERE clause

The optimizer usually works in the background. However, with diagnostic tools such as EXPLAIN and SQL\*Plus AUTOTRACE, you can see the decisions that the optimizer makes.

The optimizer determines the least-cost plan (most efficient way) to execute a SQL statement after considering many factors related to the objects referenced and the conditions specified in the query. This determination is an important step in the processing of any SQL statement and can greatly affect execution time.

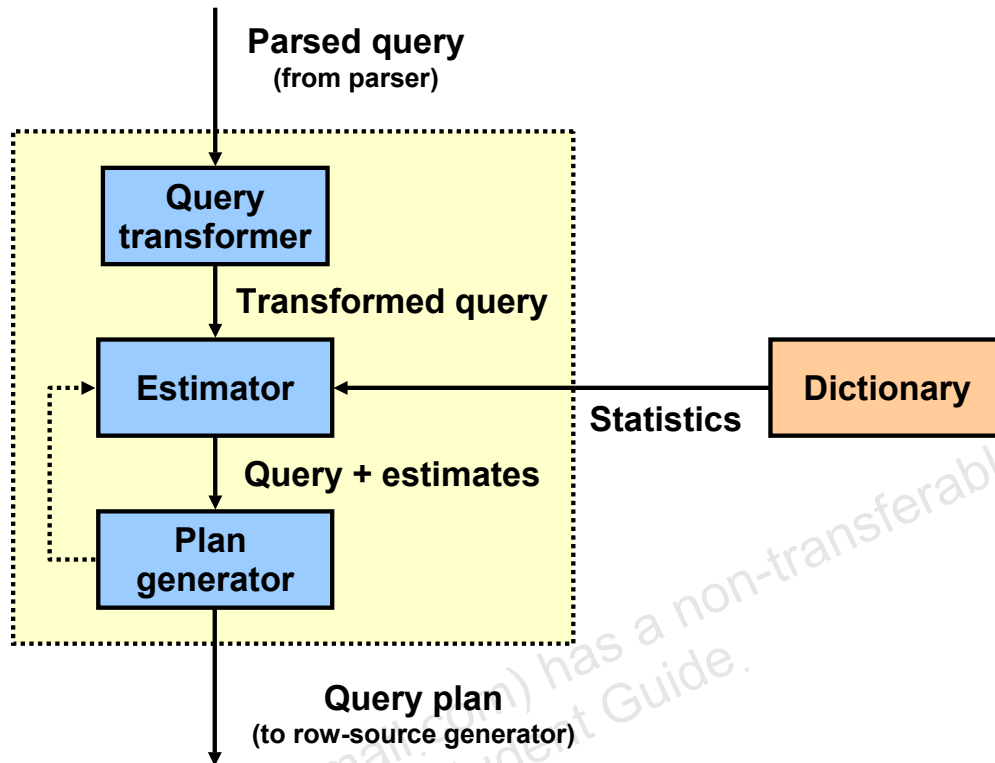
**Note:** The optimizer may not make the same decisions from one version of the Oracle Database to the next. In recent versions, the optimizer may make different decisions because better information is available.

## Oracle Optimizer (continued)

**Optimizer operations:** For any SQL statement processed by the Oracle Server, the optimizer performs the operations listed in the slide.

- **Evaluation of expressions and conditions:** The optimizer first evaluates expressions and conditions containing constants as fully as possible.
- **Statement transformation:** For complex statements involving, for example, correlated subqueries or views, the optimizer might transform the original statement into an equivalent join statement.
- **Choice of optimizer approaches:** The optimizer determines the goal of the optimization
- **Choice of access paths:** For each table accessed by the statement, the optimizer chooses one or more of the available access paths to obtain table data.
- **Choice of join orders:** For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, then which table is joined to the result, and so on.
- **Choice of join methods:** For any join statement, the optimizer chooses an operation to use to perform the join.

# Functions of the Query Optimizer



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Functions of the Query Optimizer

Optimizer (also known as query optimizer) operations include:

- Transforming queries
- Estimating
- Generating plans

This illustration depicts a query entering the query transformer. The transformed query is then sent to the estimator. Statistics are retrieved from the dictionary, and the query and estimates are sent to the plan generator. The plan generator either returns the plan to the estimator for comparison with other plans or sends the query plan to the row-source generator.

## Functions of the Query Optimizer (continued)

### Transforming Queries

The input to the query transformer is a parsed query, which is represented by a set of query blocks. A query block can be defined as a complete query, nested subquery, or nonmerged view. The query blocks are nested or interrelated to each other. The form of the query determines how the query blocks are interrelated to each other. The main objective of the query transformer is to determine if it is advantageous to change the form of the query so that it enables generation of a better query plan.

### Estimating

The estimator generates three types of measures:

- Selectivity
- Cardinality
- Cost

These measures are related to each other, and one is derived from another. The end goal of the estimator is to estimate the overall cost of a given plan. If statistics are available, then the estimator uses them to compute the measures. In the absence of statistics, the optimizer may choose a less-than-optimal plan.

### Generating Plans

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

The plan for a query is established by first generating subplans for each of the nested subqueries and nonmerged views. Each nested subquery or nonmerged view is represented by a separate query block. The query blocks are optimized separately in a bottom-to-top order. That is, the innermost query block is optimized first, and a subplan is generated for it. The outermost query block, which represents the entire query, is optimized last.

The plan generator explores various plans for a query block by trying out different access paths, join methods, and join orders. The number of possible plans for a query block is proportional to the number of join items in the FROM clause. This number rises factorially with the number of join items.

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

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Selectivity

Selectivity represents a fraction of rows from a row set. The row set can be a base table, a view, or the result of a join or a GROUP BY operator. The selectivity depends on the predicate, such as `last_name = 'Smith'`, or a combination of predicates, such as `last_name = 'Smith' AND salary > 2000`. The selectivity of a predicate indicates how many rows from a row set will pass the predicate test.

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

When statistics are available, the estimator uses them to estimate selectivity. For example, for an equality predicate (`last_name = 'Smith'`), selectivity is set to the reciprocal of the number of  $n$  distinct values of `last_name`, because the query selects rows that all contain one out of  $n$  distinct values. If a histogram is available on the `last_name` column, then the estimator uses it instead of the number of distinct values. If no statistics are available, then the optimizer uses either dynamic sampling or an internal default value, depending on the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter. Assumption of a default value can result in a less-than-optimal plan being used.

# Cardinality and Cost

- **Cardinality** represents the number of rows in a row set.
- **Cost** represents the units of work or resource that are used.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Cardinality and Cost

**Cardinality:** Represents the number of rows in a row source. Here, the row source can be a base table, a view, or the result of a join or GROUP BY operator. If a select from a table is performed, the table is the row source and the cardinality is the number of rows in that table

**Cost:** Represents the number of units of work (or resource) that are used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work. So the cost used by the query optimizer represents an estimate of the number of disk I/Os and the amount of CPU and memory used in performing an operation. The operation can be scanning a table, accessing rows from a table by using an index, joining two tables together, or sorting a row set. The cost of a query plan is the number of work units that are expected to be incurred when the query is executed and its result is produced.

The access path determines the number of units of work that are required to get data from a base table. The access path can be a table scan, a fast full index scan, or an index scan.



# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Query Optimizer Statistics in the Data Dictionary

The query optimizer requires information about data uniqueness and distribution. Using this information, the query optimizer is able to compute plan costs with a high degree of accuracy. This enables the query optimizer to choose the best execution plan based on the least cost. The information required by the optimizer is obtained from accurate statistics, which are stored in the data dictionary. In Oracle Database 10g, statistics are automatically gathered for the physical storage characteristics and data distribution of schema objects.

To maintain the effectiveness of the query optimizer, you must have statistics that are representative of the data. For table columns that contain values with large variations in the number of duplicates (called *skewed data*), histograms can prove useful. Histograms are created automatically by the database when needed.

If no statistics are available when using query optimization, the optimizer will do dynamic sampling depending on the setting of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter. Because this may cause slower parse times, the optimizer should have representative optimizer statistics for best performance.

# 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;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Enabling Query Optimizer Features

The `OPTIMIZER_FEATURES_ENABLE` parameter acts as an umbrella parameter for the query optimizer. This parameter can be used to enable a series of optimizer-related features, depending on the release. It accepts one of a list of valid string values corresponding to the release numbers, such as 8.0.4, 8.1.7, and 9.2.0.

The `OPTIMIZER_FEATURES_ENABLE` parameter allows you to upgrade the Oracle Server yet preserve the old behavior of the query optimizer after the upgrade. For example, when you upgrade the Oracle Server from release 8.1.5 to release 8.1.6, the default value of the `OPTIMIZER_FEATURES_ENABLE` parameter changes from 8.1.5 to 8.1.6. This upgrade results in the query optimizer enabling optimization features based on 8.1.6 as opposed to 8.1.5. For plan stability or backward compatibility reasons, you might not want the query plans to change because of new optimizer features in a new release. In such a case, you can set the `OPTIMIZER_FEATURES_ENABLE` parameter to an earlier version.

For example, the following setting enables the use of the optimizer features in generating query plans in Oracle9i, Release 2:

```
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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Controlling the Behavior of the Optimizer

### **CURSOR\_SHARING**

This parameter converts literal values in SQL statements to bind variables. Converting the values improves cursor sharing and can affect the execution plans of SQL statements. The optimizer generates the execution plan based on the presence of the bind variables and not the actual literal values.

### **DB\_FILE\_MULTIBLOCK\_READ\_COUNT**

This parameter specifies the number of blocks that are read in a single I/O during a full-table scan or index fast-full scan. The optimizer uses the value of **DB\_FILE\_MULTIBLOCK\_READ\_COUNT** to cost full-table scans and index fast-full scans. Larger values result in a lower cost for full-table scans and can result in the optimizer choosing a full-table scan over an index scan.

## Controlling the Behavior of the Optimizer (continued)

### **OPTIMIZER\_INDEX\_CACHING**

This parameter controls the costing of an index probe in conjunction with a nested loop. The range of values *0* to *100* for `OPTIMIZER_INDEX_CACHING` indicates the percentage of index blocks in the buffer cache, which modifies the optimizer's assumptions about index caching for nested loops and IN-list iterators. A value of *100* infers that 100% of the index blocks are likely to be found in the buffer cache, and the optimizer adjusts the cost of an index probe or nested loop accordingly. Use caution when using this parameter because execution plans can change in favor of index caching.

### **OPTIMIZER\_INDEX\_COST\_ADJ**

This parameter can be used to adjust the cost of index probes. The range of values is *1* to *10000*. The default value is *100*, which means that indexes are evaluated as an access path based on the normal costing model. A value of *10* means that the cost of an index access path is one-tenth the normal cost of an index access path. Setting this parameter influences the optimizer when considering an index as part of the execution plan.

### **OPTIMIZER\_MODE**

This initialization parameter sets the mode of the optimizer at instance startup. The possible values are `ALL_ROWS`, `FIRST_ROWS_n`, and `FIRST_ROWS`.

### **PGA\_AGGREGATE\_TARGET**

This parameter automatically controls the amount of memory allocated for sorts and hash joins when the `WORKAREA_SIZE_POLICY` parameter is set to `AUTO`. Larger amounts of memory allocated for sorts or hash joins reduce the optimizer cost of these operations. This parameter, if set to `true`, enables the query optimizer to cost a star transformation for star queries. The star transformation combines the bitmap indexes on the various fact table columns.

# Choosing an Optimizer Approach

- **OPTIMIZER\_MODE initialization parameter**
- **OPTIMIZER\_MODE parameter of the ALTER SESSION statement**
- **Optimizer statistics in the data dictionary**
- **Optimizer SQL hints for influencing the optimizer decision**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Choosing an Optimizer Approach

The optimizer's behavior when choosing an optimization approach for a SQL statement is affected by the following factors:

- **OPTIMIZER\_MODE initialization parameter:** Possible values are FIRST\_ROW, FIRST\_ROWS\_n, and ALL\_ROWS
- **OPTIMIZER\_MODE parameter of ALTER SESSION statement:** Possible values are FIRST\_ROW, FIRST\_ROWS\_n, and ALL\_ROWS. The statistics used by the optimizer are stored in the data dictionary. You can collect exact or estimated statistics about physical storage characteristics and data distribution in these schema objects by using the DBMS\_STATS package or the ANALYZE statement.
- **Optimizer SQL hints for influencing the optimizer:** Possible hints are FIRST\_ROW, FIRST\_ROWS\_n, ALL\_ROWS, CPU\_COSTING, and NO\_CPU\_COSTING.

# 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}
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Setting the Optimizer Approach

Oracle has the following basic optimizer approaches:

**FIRST\_ROWS [\_n]**: Optimizes to reduce time until the first result comes back (for example, for interactive applications). The value *n* can be 1, 10, 100, or 1000.

Choose this approach when your goal is fast response time.

**ALL\_ROWS**: Optimizes overall throughput (for example, for reports and batch jobs).

Choose this approach for best throughput.

## Optimizer Approach Levels

At the instance level, you can set the optimizer approach by using the **OPTIMIZER\_MODE** initialization parameter. When this parameter is not set, it defaults to **ALL\_ROWS**.

At the session level, the optimizer approach set at the instance level can be overruled by using the **ALTER SESSION** command. For example, the following command sets the optimizer to process for the fastest retrieval of the first row required by the query:

```
ALTER SESSION SET OPTIMIZER_MODE = first_rows_1;
```

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## FIRST\_ROWS Optimization

The optimizer can optimize a SQL statement for fast response when the parameter `OPTIMIZER_MODE` is set to `FIRST_ROWS_n` (where *n* is 1, 10, 100, or 1000) or to `FIRST_ROWS`. The hint `FIRST_ROWS (_n)` (where *n* is any positive integer) or `FIRST_ROWS` can be used to optimize an individual SQL statement for fast response.

Fast-response optimization is suitable for online users, such as those using Oracle Forms or Web access. Typically, online users are interested in seeing the first few rows and seldom look at the entire query result, especially when the result size is large. For such users, it makes sense to optimize the query to produce the first few rows as quickly as possible, even if the time to produce the entire query result is not minimized.

With fast-response optimization, the optimizer generates a plan with the lowest cost to produce the first row or the first few rows. The optimizer employs two different fast-response optimizations, referred to here as the *old* and *new* methods. The old method is used with the `FIRST_ROWS` hint or parameter value. With the old method, the optimizer uses a mixture of costs and rules to produce a plan. It is retained for backward compatibility reasons.

## FIRST\_ROWS Optimization (continued)

The `FIRST_ROWS  $n$`  mode is based solely on costs, and it is sensitive to the value of  $n$ . With small values of  $n$ , the optimizer tends to generate plans that consist of nested loop joins with index lookups. With large values of  $n$ , the optimizer tends to generate plans that consist of hash joins and full-table scans.

The value of  $n$  should be chosen based on the requirements for the online user; it depends specifically on how the result is displayed to the user. Oracle Forms users generally see the result one row at a time, and they are typically interested in seeing the first few screens. Other online users see the result one group of rows at a time.

With the fast-response method, the optimizer explores different plans and computes the cost to produce the first  $n$  rows for each plan. It picks the plan that produces the first  $n$  rows at the lowest cost. Remember that with fast-response optimization, a plan that produces the first  $n$  rows at lowest cost might not be the optimal plan to produce the entire result. If the requirement is to obtain the entire result of a query, then you should not use fast-response optimization. Instead, use the `ALL_ROWS` parameter value or hint.



# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Optimizing SQL Statements

Choose a goal for the optimizer based on the needs of your application:

- For applications performed in batch (such as Oracle Reports applications), optimize for best throughput. Throughput is usually more important in batch applications because the user initiating the application is concerned with only the time necessary for the application to complete. Response time is less important because the user does not examine the results of individual statements while the application is running.
- For interactive applications (such as Oracle Forms applications or SQL\*Plus queries), optimize for best response time. Response time is usually important in interactive applications because the interactive user is waiting to see the first row or first few rows accessed by the statement. You can set the number of rows that should be retrieved first by using the optimizer mode of `FIRST_ROWS_n`, where *n* specifies the number of rows and can be customized for the different pages in the application.

**Note:** This topic is discussed in more detail in the lesson titled “Optimizer Operations.”

# How the Query Optimizer Executes Statements

The factors considered by the optimizer are:

- **Access path**
- **Join order**
- **Join method**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## How the Query Optimizer Executes Statements

To choose an execution plan for a join statement, the optimizer must make these interrelated decisions:

**Access paths:** For simple statements, the optimizer must choose an access path to retrieve data from each table in the join statement.

**Join method:** To join each pair of row sources, Oracle Database must perform a join operation. Join methods include nested loop, sort merge, Cartesian, and hash joins.

**Join order:** To execute a statement that joins more than two tables, Oracle Database joins two of the tables and then joins the resulting row source to the next table. This process is continued until all tables are joined into the result.

# Access Paths

- **Full-table scans**
- **Row ID scans**
- **Index scans**
- **Cluster scans**
- **Hash scans**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Access Paths

Access paths are ways in which data is retrieved from the database. Any row can be located and retrieved with one of the methods listed in the slide.

# Join Orders

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Join Orders

A join order is the order in which different join items (such as tables) are accessed and joined together. For example, in a join order of table1, table2, and table3, table1 is accessed first. Next table2 is accessed, and its data is joined to table1 data to produce a join of table1 and table2. Finally, table3 is accessed, and its data is joined to the result of the join between table1 and table2.

# Join Methods

**The different join methods considered by the optimizer are:**

- **Nested-loop join**
- **Hash join**
- **Sort-merge join**
- **Cartesian join**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Join Methods

A join operation combines the output from two row sources (such as tables or views ) and returns one resulting row source (data set). The Oracle Server supports the join methods that are listed in the slide.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

This lesson introduced the Oracle optimizer. The optimizer uses statistics to calculate the cost of access paths, taking into account the number of logical I/Os, CPU usage, and network traffic.

You also learned how to set the optimizer approach at the instance and session level with different settings:

- `FIRST_ROWS_n` Optimizes response time for the first result set
- `ALL_ROWS` Optimizes overall throughput

# 5

## Optimizer Operations

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Objectives

After completing this lesson, you should be able to understand the behavior of the Oracle query optimizer at the following levels:

- Access path
- Sorts
- Joins



## Review: How the Query Optimizer Executes Statements

The factors considered by the optimizer are:

- **Access path**
- **Join order**
- **Join method**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Review: How the Query Optimizer Executes Statements

To choose an execution plan for a statement, the optimizer must make these interrelated decisions.

**Access path:** The optimizer must choose an access path to retrieve data from each table in the join statement such as use indexes, full table scans, or materialized views.

**Join order:** To execute a statement that joins more than two tables, Oracle Database joins two of the tables and then joins the resulting row source to the next table. This process is continued until all tables are joined into the result.

**Join method:** If more than one table is involved, Oracle Database must perform a join operation to join each pair of row sources. Join methods include nested loop join, sort-merge join, Cartesian join, and hash join.

# Access Paths

- **Full table scan**
- **Row ID scan**
- **Index scan**
- **Sample table scan**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Access Paths

Access paths are the ways in which data is retrieved from the database. Any row can be located and retrieved with one of the methods mentioned in the slide. Access paths are ways in which data is retrieved from the database.

In general, index access paths should be used for statements that retrieve a small subset of table rows, while full scans are more efficient when accessing a large portion of the table. Online transaction processing (OLTP) applications, which consist of short-running SQL statements with high selectivity, often are characterized by the use of index access paths. Decision support systems, on the other hand, tend to use full scans of the relevant objects.

# Choosing an Access Path

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Choosing an Access Path

The optimizer chooses an access path based on the following factors:

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

To choose an access path, the optimizer first determines which access paths are available by examining the conditions in the statement's WHERE clause and its FROM clause. The optimizer then generates a set of possible execution plans using available access paths and estimates the cost of each plan, using the statistics for the index, columns, and tables accessible to the statement. Finally, the optimizer chooses the execution plan with the lowest estimated cost.

When choosing an access path, the optimizer is influenced by the following:

- **Optimizer hints:** The optimizer's choice among available access paths can be overridden with hints
- **Old statistics:** For example, if a table has not been analyzed since it was created, and if the table is small, then the optimizer uses a full table scan. The LAST\_ANALYZED and BLOCKS columns in the ALL\_TABLES table reflect the statistics used by the optimizer.

# Full Table Scans

- **Lack of index**
- **Large amount of data**
- **Small table**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Full Table Scans

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

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

Full table scans are cheaper than index range scans when accessing a large fraction of the blocks in a table. They are cheaper because full table scans can use larger I/O calls; making fewer large I/O calls is cheaper than making many smaller calls.

## Full Table Scans (continued)

The optimizer uses a full table scan in each of the following cases:

- Lack of index: If the query is unable to use any existing indexes, then it uses a full table scan. For example, if there is a function used on the indexed column in the query, the optimizer is unable to use the index and instead uses a full table scan.
- Large amount of data: If the optimizer thinks that the query will access most of the blocks in the table, then it uses a full table scan, even though indexes might be available.
- Small table: If a table contains blocks fewer than the value of `DB_FILE_MULTIBLOCK_READ_COUNT` under the high-water mark, then a full table scan might be cheaper because this can be read in a single I/O call.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

## 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Row ID Scans

The row ID of a row specifies the data file and data block containing the row as well as the location of the row in that block. Locating a row by specifying its row ID is the fastest way to retrieve a single row, because the exact location of the row in the database is specified.

To access a table by row ID, Oracle Database first obtains the row IDs of the selected rows, either from the statement's WHERE clause or through an index scan of one or more of the table's indexes. Oracle Database then locates each selected row in the table based on its row ID. This is generally the second step after retrieving the row ID from an index.

The table access might be required for any columns in the statement that are not present in the index. Access by row ID does not need to follow every index scan. If the index contains all the columns needed for the statement, then table access by row ID might not occur.

**Note:** Due to row migration, a row ID can sometimes point to a different address than the actual row location, resulting in more than one block being read to locate a row. This is caused by the value of the PCTFREE initialization parameter, which influences the space reserved for updates, making the space too small. In this case, an update to a row causes the row to be placed in another block with a pointer in the original block. The row ID, however, will still have only the address of the original block.

# Index Scans

## Types of index scans:

- **Index unique scan**
- **Index range scan**
- **Index range scan descending**
- **Index skip scan**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Index Scans

In this method, the indexed column values specified by the statement are used to retrieve the row. An index scan retrieves data from an index based on the value of one or more columns in the index. To perform an index scan, Oracle Database searches the index for the indexed column values accessed by the statement. If the statement accesses only columns of the index, then Oracle Database reads the indexed column values directly from the index rather than from the table.

The index contains not only the indexed value but also the row IDs of rows in the table having that value. Therefore, if the statement accesses other columns in addition to the indexed columns, then Oracle Database can find the rows in the table by using either a table access by row ID or a cluster scan.

### Index Unique Scan

An *index unique scan* returns, at most, a single row ID. Oracle Database performs a unique scan if a statement contains a `UNIQUE` or a `PRIMARY KEY` constraint that guarantees that only a single row is accessed. This access path is used when all columns of a unique (B-tree) index are specified with equality conditions. In general, you do not need to use a hint to do a unique scan.

## Index Scans (continued)

### Index Range Scan

An *index range scan* is a common operation for accessing selective data. It can be bounded (on both sides) or unbounded (on one or both sides). Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted in ascending order by row ID. If data must be sorted by order, then use the `ORDER BY` clause and do not rely on an index. If an index can be used to satisfy an `ORDER BY` clause, then the optimizer uses this option and avoids a sort. The optimizer uses a range scan when it finds one or more leading columns of an index specified in conditions, such as `col1 = :b1`, `col1 < :b1`, `col1 > :b1`, and any combination of the preceding conditions for leading columns in the index. Wildcard searches (`col1 like '%ASD'`) should not be in a leading position, as this does not result in a range scan. Range scans can use unique or nonunique indexes. Range scans avoid sorting when index columns constitute the `ORDER BY` clause. A `NOT NULL` constraint can influence the optimizer to use an available index on the column to satisfy an `ORDER BY` clause, thus avoiding further sorting.

A hint might be required if the optimizer chooses some other index or uses a full table scan. The hint `INDEX(table_alias index_name)` specifies the index to use. Suppose that the indexed column has a skewed distribution; then the column should have histograms so that the optimizer knows about the distribution.

### Index Range Scan Descending

An *index range scan descending* is identical to an index range scan, except that the data is returned in descending order. Indexes, by default, are stored in ascending order. This scan is usually used when ordering data in a descending order to return the most recent data first, or when seeking a value less than a specified value. The optimizer uses index range scan descending when an order by descending clause can be satisfied by an index. The hint `INDEX_DESC(table_alias index_name)` is used for this access path.

### Index Skip Scan

*Index skip scans* improve index scans by skipping blocks that could never contain keys matching the filter column values. Scanning index blocks is often faster than scanning table data blocks. Skip scanning can happen when the initial column of the composite index is not specified in a query. You can use the `INDEX_SS` hint to influence the optimizer to perform a skip scan.



# Index Scans

## Types of index scans:

- **Full scan**
- **Fast-full index scan**
- **Index join**
- **Bitmap join**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Index Scans (continued)

### Full Scans

A *full scan* is available if a predicate references one of the columns in the index. The predicate does not need to be an index driver. A full scan is also available when there is no predicate if both the following conditions are met:

- All of the columns in the table referenced in the query are included in the index.
- At least one of the index columns is not null.

A full scan can be used to eliminate a sort operation, because the data is ordered by the index key. It reads the blocks singly.

### Fast-Full Index Scans

*Fast-full index scans* are an alternative to full table scans when the index contains all the columns that are needed for the query and at least one column in the index key has the NOT NULL constraint. A fast-full scan accesses the data in the index itself without accessing the table. It cannot be used to eliminate a sort operation because the data is *not* ordered by the index key. It reads the entire index using multiblock reads (unlike a full index scan) and can be parallelized. You can specify it with the initialization parameter OPTIMIZER\_FEATURES\_ENABLE or the hint INDEX\_FFS. Fast-full index scans cannot be performed against bitmap indexes. A fast-full scan is faster than a normal full index scan because it can use multiblock I/O and can be parallelized just like a table scan.

## Index Scans (continued)

### Index Joins

An *index join* is a hash join of several indexes that together contain all the table columns that are referenced in the query. If an index join is used, then no table access is needed because all the relevant column values can be retrieved from the indexes. An index join cannot be used to eliminate a sort operation. You can specify an index join with the initialization parameter `OPTIMIZER_FEATURES_ENABLE` or the hint `INDEX_JOIN`.

### Bitmap Joins

A *bitmap join* uses a bitmap for key values and a mapping function that converts each bit position to a `row ID`. Bitmaps can efficiently merge indexes that correspond to several conditions in a `WHERE` clause, using Boolean operations to resolve `AND` and `OR` conditions.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Joining Multiple Tables

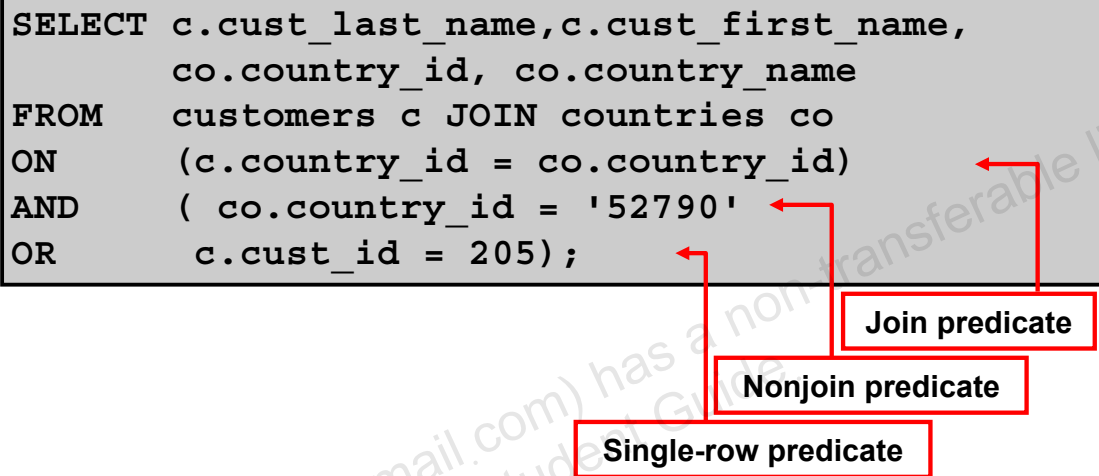
The Oracle Server can join only two row sources at a time. Join operations (such as nested loops and sort-merge) are used as building blocks if the join statement contains more than two tables. Except for the limitations that are inherent in each kind of join statement, there are no restrictions on the combination of join operations involved in a join with more than two tables.

For each join statement, the optimizer determines the following (in sequence):

1. Order in which to join the tables
2. Best join operation to apply for each join
3. Access path for each row source

# Join Terminology

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



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Join Terminology

A join statement is a select statement with more than one table in the FROM clause.

However, the optimizer uses the techniques usually associated with joins when executing other kinds of statements where more than one table is referenced (as in subqueries, which are discussed later in this lesson). This slide gives an example of an SQL:1999-compliant join statement that is available in Oracle Database.

A *join predicate* is a predicate in the WHERE clause that combines the columns of two of the tables in the join.

A *nonjoin predicate* is a predicate in the WHERE clause that references only one table.

A *single-row predicate* is an equality predicate on a column with a unique or primary key constraint, or a column with a unique index without a corresponding constraint. The optimizer knows that these predicates always return either one row or no rows at all.

The corresponding join that is proprietary to Oracle Database is:

```
SELECT  c.cust_last_name, c.cust_first_name,
        co.country_id, co.country_name
FROM    customers c , countries co
WHERE   c.country_id = co.country_id
AND     co.country_id = '52790'
OR      c.cust_id = 205;
```

# Join Terminology

- **Natural join**

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

- **Join with nonequal predicate**

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

- **Cross join**

```
SELECT *
FROM   customers c CROSS JOIN countries co;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Join Terminology (continued)

A *natural join* is a join statement in which the join predicate automatically joins between all the matched columns in both tables using the equality (=) operator. The first statement in the slide is an example of a natural join, otherwise known as an *equijoin*. It can also be written as:

```
SELECT c.cust_last_name, co.country_name
FROM   customers c, countries co
WHERE  c.country_id = co.country_id;
```

The second example in the slide is also called a *non-equijoin*. This can be written as:

```
SELECT s.amount_sold, p.promo_name
FROM   sales s, promotions p
WHERE  s.promo_id=p.promo_id
AND s.time_id BETWEEN p.promo_begin_date AND p.promo_end_date;
```

A *cross join* is a join statement in which there is no join predicate. These joins attempt to fetch every combination of rows in both tables. They occur infrequently and may indicate a poor database design. They are the equivalent of the following Cartesian join:

```
SQL> SELECT *
      2   FROM customers ,countries;
```

**Note:** Cartesian joins and natural joins are best avoided by specifying the join columns, because they can use up resources and get poor results.

# SQL:1999 Outer Joins

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

```
SELECT s.time_id, t.time_id
FROM   sales s
RIGHT OUTER JOIN times t
ON      (s.time_id = t.time_id);
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Outer Joins

Oracle Database supports SQL:1999-compliant join syntax. You can specify RIGHT, LEFT, and FULL outer joins to create the outer joins as well as using the plus (+) sign (as shown in the next slide).

A join between two tables that returns the results of the INNER join, as well as unmatched rows in the left (right) table, is considered a LEFT (RIGHT) OUTER JOIN.

A join between two tables that returns the results of an INNER join, as well as the results of both the LEFT and RIGHT outer joins, is considered a FULL OUTER JOIN. This is not intended to improve performance.

The use of the plus (+) sign is still supported.

# 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;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Oracle Proprietary Outer Joins

An *outer join* is a join in which the join predicates have a plus (+) sign added to signify that the join returns a row even if the table with this special predicate (the outer-joined table) does not have a row that satisfies the join condition.

### Outer Join: Example with Predicates

```
SELECT s.time_id, t.time_id
FROM sales s, times t
WHERE s.time_id(+) = t.time_id --join predicate with +
AND s.promo_id(+) = 19      -- non-join predicate with +
AND s.quantity_sold(+) > 45 --non-join predicate with +
```

**Note:** Predicates without a plus (+) sign on the table that is outer-joined disable the outer join functionality.

## 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.

```
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);
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### 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. In other words, with full outer joins, you can join tables together yet still show rows that do not have corresponding rows in joined-to tables.



# 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';
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Outer Joins: Examples

```
SELECT  c.cust_id,  co.country_name
FROM    customers c LEFT OUTER JOIN countries co
ON      ( c.country_id = co.country_id) ;
```

The optimizer chooses to use a HASH\_JOIN.

```
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 = 52780;
```

The optimizer can use an index on the co.country\_id column (a single-row predicate). COUNTRIES is the driving table.

```
SELECT  co.country_name, c.cust_id
FROM    customers c LEFT OUTER JOIN countries co
ON      ( c.country_id = co.country_id )
and     c.cust_id = 780;
```

This statement is an outer join; the indexes on both c.country\_id and c.cust\_id are usable.

**Note:** Either the nonjoin predicates on the outer-joined table or the join predicate—but not both—can use indexes.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Join Order Rules

- A single-row predicate forces its row source to be placed first in the join order.
- For outer join statements, the outer-joined table must come after (in the join order) the other table in the join predicate.

By making a single-row predicate table the driving table of a join operation, the optimizer effectively eliminates one join operation. For example, for a nested loop join, the main loop is no longer a real loop. The single-row predicate results in one row (or no rows at all), so the inner loop is executed only once (or not at all).

The reason for the second rule (about outer joins) is that you would otherwise be unable to produce the correct outer join results. For example, if you join the COUNTRIES table with the CUSTOMERS table and you also want to see countries that have no corresponding customers, you must start with the COUNTRIES table.

After considering these two rules, determine the join order and join operations based on the optimizer goal that is used.

# Join Optimization

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

| Number of Tables | Join Orders |
|------------------|-------------|
| -----            | -----       |
| 2                | $2! = 2$    |
| 3                | $3! = 6$    |
| 4                | $4! = 24$   |

- **Parse time grows factorially when adding tables to a join.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Join Optimization

The number of tables in the join and the specified join predicates determine the number of possible execution plans. Theoretically, the number of possible execution plans is as follows:

| Number of Tables | Possible Plans         |
|------------------|------------------------|
| 2                | 2                      |
| 3                | $3! = 3 * 2 = 6$       |
| 4                | $4! = 4 * 3 * 2 = 24$  |
| 5                | $5! = 5 * 4 * 3 * 2 =$ |
| 120              |                        |
| ...              | ...                    |

Because only the execution plans that follow a join predicate are considered, the number of possible execution plans usually does not reach this value when the number of tables is greater than 3. However, joining 20 tables still results in many possible execution plans. Remember that all tables involved in a join, including those that are hidden in views, count as separate tables.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Join Methods

A join operation combines the output from two row sources (such as tables or views) and returns one resulting row source (data set). The Oracle Server supports different join methods such as the following:

**Nested loop join:** Useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the second table

**Hash join:** Used for joining large data sets. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows. This method is best used when the smaller table fits in available memory. The cost is then limited to a single read pass over the data for the two tables.

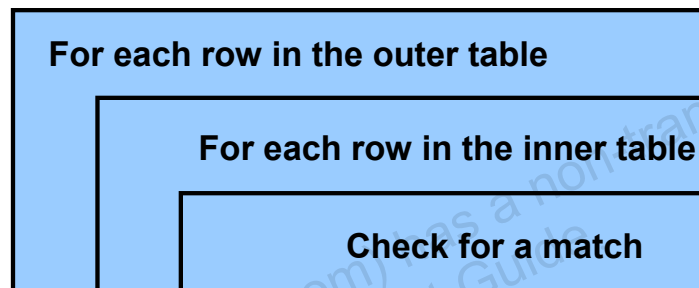
**Sort-merge join:** Can be used to join rows from two independent sources. Hash joins generally perform better than sort-merge joins. On the other hand, sort-merge joins can perform better than hash joins if both of the following conditions are met:

- The row sources are already sorted.
- A sort operation does not have to be done.

These methods are covered in the following slides.

# 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.



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Basic Execution Plan of Nested Loop Joins

NESTED LOOPS

TABLE ACCESS (...) OF our\_outer\_table

TABLE ACCESS (...) OF our\_inner\_table

Because they are often used with indexed table accesses, nested loops are common with an execution plan that is similar to the following:

NESTED LOOPS

TABLE ACCESS (BY row ID) OF our\_outer\_table

INDEX (... SCAN) OF outer\_table\_index (.....)

TABLE ACCESS (BY row ID) OF our\_inner\_table

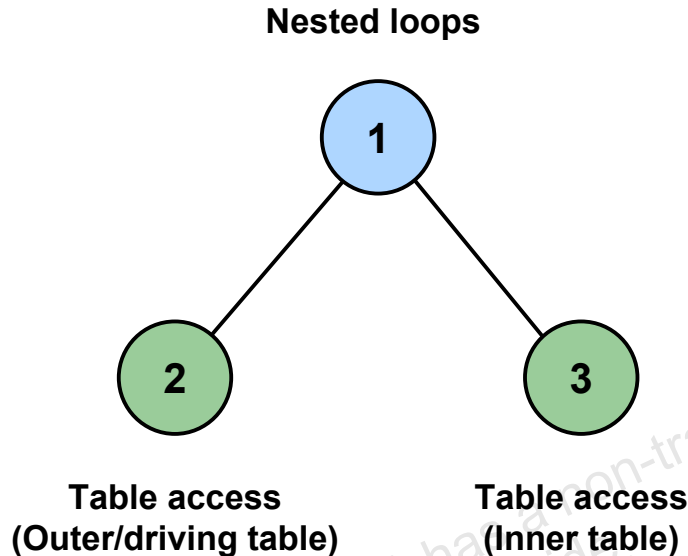
INDEX (RANGE SCAN) OF inner\_table\_index (NON-UNIQUE)

The outer (driving) table is usually accessed with a full table scan.

## Non-Equi Joins

If you have a non-equi join, a hash join is not possible. Depending on the nonjoin predicates, non-equi joins can result in a nested loop with full table scans on both the outer and the inner tables. This means that a full table scan is performed on the outer table; in addition, for each row, a full table scan is performed (again and again) on the inner table. This usually results in poor performance.

# Nested Loop Join Plan



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Nested Loop Join Plan

For each row in the outer (driving) table, all rows in the inner table that satisfy the join predicate are retrieved.

Any nonjoin predicates on the inner table are considered after this initial retrieval, unless a composite index (combining both the join and the nonjoin predicate) is used.

The nested loop join can solve all kinds of joining requirements, no matter what the join predicate is. Therefore, you should use the nested loop join when you cannot use sort-merge, hash, or cluster joins. These join techniques are discussed later in this lesson.

Oracle Database introduces an internal optimization called *data block prefetching*, which can in some cases improve response times of queries significantly. Data block prefetching is used by table lookup. When an index access path is chosen and the query cannot be satisfied by the index alone, the data rows indicated by the row ID also need to be fetched. This behavior is automatic and cannot be influenced by hints. In joins using data block prefetching, you may see an explain plan such as the following:

```
...  
TABLE ACCESS (BY INDEX row ID) OF (...)  
  NESTED LOOPS (Cost=...)  
    TABLE ACCESS (FULL) OF (...)  
      INDEX (RANGE SCAN) OF (...)
```

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## When Are Nested Loop Joins Used?

The optimizer uses nested loop joins when joining small number of rows that have a good driving condition between the two tables. You drive from the outer loop to the inner loop, so the order of tables in the execution plan is important.

The outer loop is the driving row source. It produces a set of rows for driving the join condition. The row source can be a table accessed using an index scan or a full table scan. Also, the rows can be produced from any other operation. For example, the output from a nested loop join can be used as a row source for another nested loop join.

The inner loop is iterated for every row that is returned from the outer loop, ideally by an index scan. If the access path for the inner loop is not dependent on the outer loop, then you can end up with a Cartesian product; for every iteration of the outer loop, the inner loop produces the same set of rows. Therefore, you should use other join methods when two independent row sources are joined together.

If the optimizer chooses to use some other join method, you can use the `USE_NL(table1 table2)` hint, where `table1` and `table2` are the aliases of the tables being joined.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hash Joins

Hash joins are performed only for equijoins. These are the steps performed for a hash join:

1. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory.
2. The optimizer then scans the larger table, probing the hash table to find the joined rows.

### Basic Execution Plan

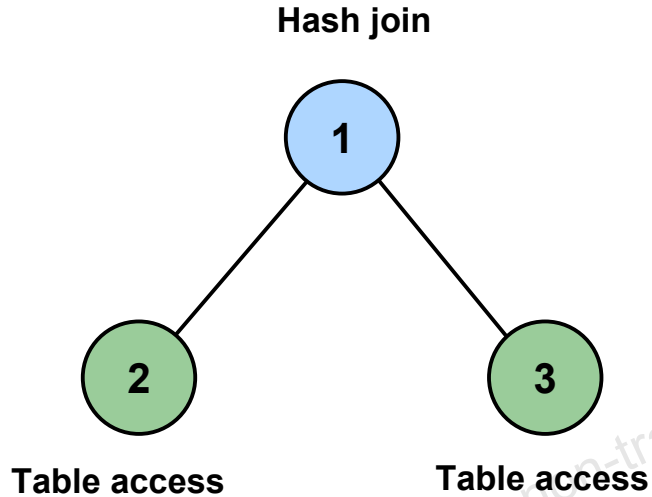
```
HASH JOIN
  TABLE ACCESS (..) OF table_A
  TABLE ACCESS (..) OF table_B
```

### Performance Considerations

As a general rule, hash joins outperform sort-merge joins. In some cases, a sort-merge join can outperform a nested loop join; it is even more likely that a hash join will do so.



# Hash Join Plan



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hash Join Plan

The slides illustrates a *parse tree* (sequence of steps) for the hash join. To execute a hash join, the Oracle Server performs the steps that are indicated in the slide:

- Steps 2 and 3 perform full table scans of the tables that must be joined.
- Step 1 builds a hash table out of the rows coming from step 2 and probes it with each row coming from step 3.

## 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.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### When Are Hash Joins Used?

The optimizer uses a hash join to join two tables if they are joined using an equijoin and if either of the following conditions is true:

- A large amount of data needs to be joined.
- A large fraction of the table needs to be joined.

Apply the `USE_HASH` hint to advise the optimizer to use a hash join when joining two tables together.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Sort-Merge Joins

In the sort operations, the two row sources are sorted on the values of the columns used in the join predicate. If a row source has already been sorted in a previous operation, the sort-merge operation skips the sort on that row source.

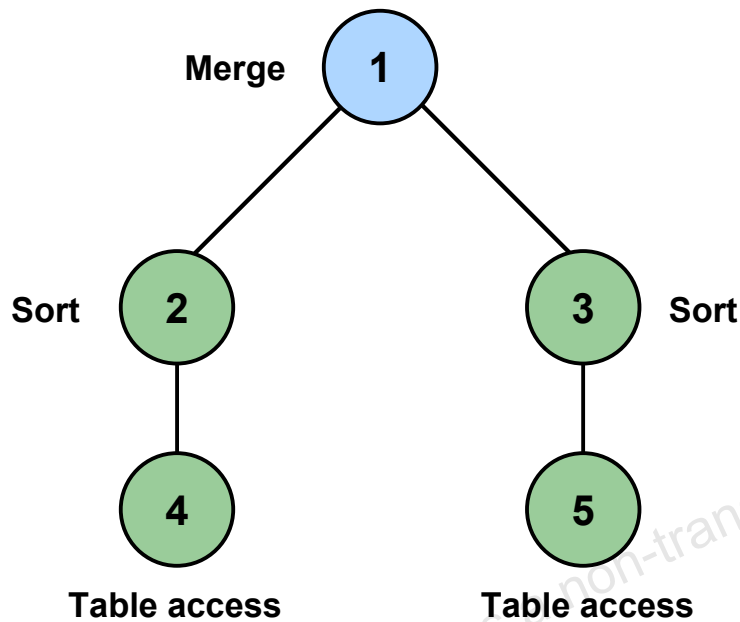
Sorting could make this join technique expensive, especially if sorting cannot be performed in memory.

The merge operation combines the two sorted row sources to retrieve every pair of rows that contain matching values for the columns used in the join predicate.

### Basic Execution Plan

```
MERGE (JOIN)
  SORT (JOIN)
    TABLE ACCESS (...) OF table_A
  SORT (JOIN)
    TABLE ACCESS (...) OF table_B
```

## Sort-Merge Join Plan



ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Sort-Merge Join Plan

The table accesses (row sources 4 and 5) can be based on index scans if there are nonjoin predicates that can use an index. However, this join operation most often appears with full table scans on both tables.

The sequence (which table is sorted first) does not matter; there is no concept of an outer or driving table. Remember that the sorting can result in the creation of temporary segments on disk (giving more I/O than just the table scans themselves).

Sort-merge joins tend to perform better than an indexed nested loop join if the number of rows satisfying the join condition represents a major part of the total number of rows in the two tables.

**Note:** The breakeven point for a sort-merge join and an indexed nested loop join depends on the sorting performance of the database. Sort performance can be influenced by defining temporary tablespaces which are of the TEMPORARY type.

# 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.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

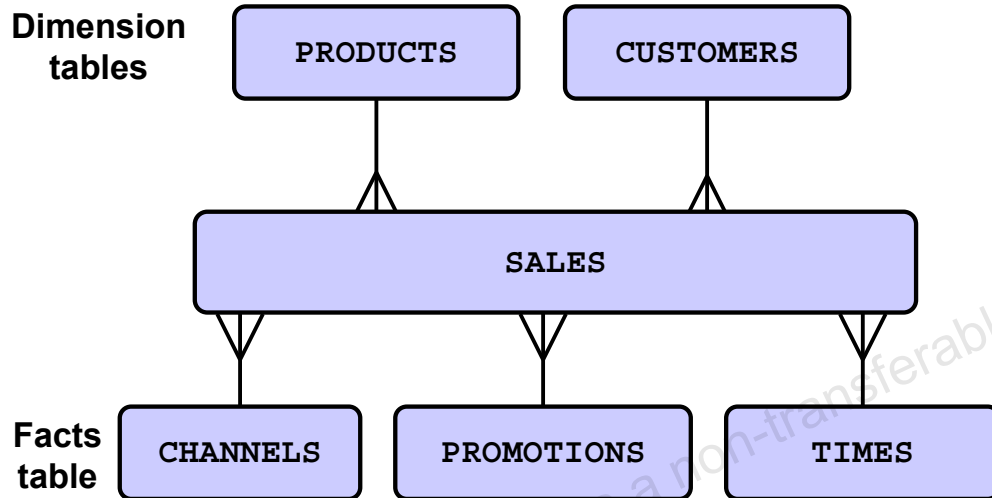
## When Are Sort-Merge Joins Used?

The optimizer can choose a sort-merge join over a hash join for joining large amounts of data if either of the following conditions is true:

- The join condition between two tables is not an equijoin.
- Because of sorts already required by other operations, the optimizer finds it is cheaper to use a sort-merge than a hash join.

**Note:** A sort-merge operation can occur on equijoins with hints.

# Star Joins



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Star Joins

One type of data warehouse design focuses on a *star schema*. A star schema is characterized by one or more large fact tables that contain the primary information in the data warehouse and a number of much smaller dimension tables (or lookup tables), each of which contains information about the entries for a particular attribute in the fact table.

A *star query* is a join between a fact table and a number of lookup tables. Each lookup table is joined to the fact table using a primary key-to-foreign key join, but the lookup tables are not joined to each other.

**Note:** Star joins are primarily used in data warehousing applications. For more information on this topic, see Appendix D (“Data Warehouse Tuning Considerations”).

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## How the Query Optimizer Chooses Execution Plans for Joins

The query optimizer considers the following when choosing an execution plan:

- The optimizer first determines whether joining two or more tables definitely results in a row source containing at most one row. The optimizer recognizes such situations based on UNIQUE and PRIMARY KEY constraints on the tables. If such a situation exists, then the optimizer places these tables first in the join order. The optimizer then optimizes the join of the remaining set of tables.
- For join statements with outer join conditions, the table with the outer join operator must come after the other table in the condition in the join order. The optimizer does not consider join orders that violate this rule. Similarly, when a subquery has been converted into an antijoin or semijoin, the tables from the subquery must come after those tables in the outer query block to which they were connected or correlated. However, hash antijoins and semijoins are able to override this ordering condition in certain circumstances.

## How the Query Optimizer Chooses Execution Plans for Joins (continued)

- The optimizer generates a set of execution plans according to possible join orders, join methods, and available access paths. The optimizer then estimates the cost of each plan and chooses the one with the lowest cost. The optimizer estimates costs in the following ways:
  - The cost of a nested loop operation is based on the cost of reading each selected row of the outer table and each of its matching rows of the inner table into memory. The optimizer estimates these costs using the statistics in the data dictionary.
  - The cost of a sort-merge join is based largely on the cost of reading all the sources into memory and sorting them.
  - The cost of a hash join is based largely on the cost of building a hash table on one of the input sides to the join and using the rows from the other of the join to probe it.
- The optimizer also considers other factors when determining the cost of each operation. Here is an example: A smaller sort area size is likely to increase the cost for a sort-merge join because sorting takes more CPU time and I/O in a smaller sort area. A larger multiblock read count is likely to decrease the cost for a sort-merge join in relation to a nested loop join. If a large number of sequential blocks can be read from disk in a single I/O, then an index on the inner table for the nested loop join is less likely to improve performance over a full table scan. The multiblock read count is specified by the `DB_FILE_MULTIBLOCK_READ_COUNT` initialization parameter.
- With the query optimizer, the optimizer's choice of join orders can be overridden with the `ORDERED` hint. If the `ORDERED` hint specifies a join order that violates the rule for an outer join, then the optimizer ignores the hint and chooses the order. Also, you can override the optimizer's choice of join method with hints.



# Subqueries and Joins

- **Subqueries (like joins) are statements that reference multiple tables.**
- **Subquery types:**
  - **Noncorrelated subquery**
  - **Correlated subquery**
  - **NOT IN subquery (antijoin)**
  - **EXISTS subquery (semijoin)**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Noncorrelated Subqueries

A *noncorrelated subquery* does not contain references to the outer (main) query and can be executed independently, as in the following example:

```
SELECT c.*
FROM   customers c
WHERE  c.country_id IN
      (SELECT co.country_id
       FROM   countries co
       WHERE  co.country_subregion = 'Asia');
```

This statement is typically executed as a hash join with the subquery as the outer table. It retrieves all customers in location “Asia.” This is the execution plan structure:

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=363 Card=6250
      Bytes= 918750)
1    0    HASH JOIN (Cost=363 Card=6250 Bytes=918750)
2    1      TABLE ACCESS (FULL) OF 'COUNTRIES' (Cost=1 Card=2
      Bytes=28)
3    1      TABLE ACCESS (FULL) OF 'CUSTOMERS' (Cost=360
      Card=50000 Bytes=6650000)
```

## Correlated Subqueries

This example of a correlated subquery finds the latest promotion date for every promotion:

```
SELECT p1.*
FROM   promotions p1
WHERE  p1.promo_cost =
      (SELECT MAX(p2.promo_cost)
       FROM   promotions p2
       WHERE  p1.promo_category = p2.promo_category)
Execution Plan
```

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=9 ...)
1    0      HASH JOIN (Cost=9 ...)
2    1        VIEW OF 'VW_SQ_1' (Cost=5 ...)
3    2          SORT (GROUP BY) (Cost=5 ...)
4    3            TABLE ACCESS (FULL) OF 'PROMOTIONS' (Cost=3 ...)
5    1            TABLE ACCESS (FULL) OF 'PROMOTIONS' (Cost=3 ...)
```

## Antijoins

An *antijoin* is a select statement with a subquery with a NOT IN predicate, as in the following example:

```
SELECT c.*
FROM   customers c
WHERE  c.cust_income_level = 'F: 110,000 - 129,999'
AND    c.country_id NOT IN
      (SELECT co.country_id
       FROM countries co
       WHERE co.country_subregion = 'Europe');
Execution Plan
```

```
-----
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=188 ...)
...
3    2      HASH JOIN* (RIGHT ANTI) (Cost=188 ...)
...
7    6        TABLE ACCESS (FULL) OF 'COUNTRIES' (TABLE)
              (Cost=3 ...)
9    8          TABLE ACCESS* (FULL) OF 'CUSTOMERS' (TABLE)
              (Cost= ...)
```

**Note:** This example is abbreviated to avoid showing the parallel execution.

## Semijoins

A *semijoin* returns rows that match an EXISTS subquery without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery. Here is an example:

```
SELECT e.employee_id, e.first_name, e.last_name, e.salary
FROM hr.employees e
WHERE e.department_id = 80
AND e.job_id          = 'SA_REP'
AND EXISTS (SELECT 1
            FROM oe.orders o
            WHERE e.employee_id = o.sales_rep_id);
```

Execution Plan

```
-----
0  SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 ...)
1    0    NESTED LOOPS (SEMI) (Cost=3 Card=9 Bytes=261)
2      1    TABLE ACCESS (FULL) OF 'EMPLOYEES' (TABLE) (Cost=3 ...)
3      1      INDEX (RANGE SCAN) OF 'ORD_SALES_REP_IX' (INDEX)
              (Cost=0...)
```

# Sort Operations

- **SORT UNIQUE**
- **SORT AGGREGATE**
- **SORT GROUP BY**
- **SORT JOIN**
- **SORT ORDER BY**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Sort Operations

Sort operations result when users specify an operation that requires a sort. Commonly encountered operations include the following:

- **SORT UNIQUE** occurs if a user specifies a **DISTINCT** clause or if an operation requires unique values for the next step.
- **SORT AGGREGATE** does not actually involve a sort. It is used when aggregates are being computed across the whole set of rows.
- **SORT GROUP BY** is used when aggregates are being computed for different groups in the data. The sort is required to separate the rows into different groups.
- **SORT JOIN** happens during a sort-merge join if the rows need to be sorted by the join key.
- **SORT ORDER BY** is required when the statement specifies an **ORDER BY** that cannot be satisfied by one of the indexes.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Tuning Sort Performance

If possible, write SQL statements in such a way that sorting is not needed. Note that several SQL language components cause implicit sorts, such as `DISTINCT`, `GROUP BY`, `UNION`, `MINUS`, and `INTERSECT`. Make sure that you get only the sorts that are needed for your result.

Sorts can be avoided by creating indexes. Concatenated indexes, with the appropriate `DESC` or `ASC` attributes, can especially help avoid sorts. Note, however, that you lose the advantage of multiblock I/O because, to avoid a sort, the index must be scanned sequentially. An index fast-full scan (using multiblock I/O) does not guarantee that the rows are returned in the correct order.

# 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;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Top-N SQL

The significance of the Top-N SQL feature is that you do not need to sort a full set if, for example, you are interested in only the four highest (or lowest) values. If a set is too big to be sorted in memory, the performance is significantly degraded. This is caused by the I/O to and from the temporary storage of intermediate results on disk.

If you want to know only the four highest values, you need an array with four slots to scan the set and keep the four highest values in that array.

The `WHERE` clause of the statement in the slide is merged into the inline view to prevent the full `PRODUCTS` table from being sorted by `PROD_LIST_PRICE`. This appears in the execution plan as follows:

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=248 ...)
1    0      COUNT (STOPKEY)
2    1      VIEW (Cost=248 Card=10000 Bytes=660000)
3    2      SORT (ORDER BY STOPKEY) (Cost=248 ...)
4    3      TABLE ACCESS (FULL) OF 'PRODUCTS' (Cost=109 ...)
```

# 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 multipass execution.**
- **A reasonable amount of PGA memory allows single-pass executions.**
- **The size of PGA is controlled with:**
  - **PGA\_AGGREGATE\_TARGET**
  - **WORKAREA\_SIZE\_POLICY**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Memory and Optimizer Operations

Memory-intensive, sort-based operators, such as ORDER BY, GROUP BY and hash-joins, use work areas in the PGA. A sort operator uses a work area (the sort area) to perform the in-memory sort of a set of rows. Similarly, a hash-join operator uses a work area (the hash area) to build a hash table from its left input. If the amount of data to be processed by these two operators does not fit into a work area, then the input data is divided into smaller pieces. This allows some data pieces to be processed in memory while the rest are spilled to temporary disk storage to be processed later.

Optimally, the size of a work area is big enough to accommodate the input data and auxiliary memory structures allocated by its associated SQL operator. If not, response time increases, because part of the input data must be spilled to temporary disk storage. If the size of a work area is far too small compared to the input data size, multiple passes over the data pieces must be performed. This can dramatically increase the response time of the operator. A system configured with a reasonable amount of PGA memory does not need to perform multiple passes over the input data.

Automatic PGA memory management simplifies and improves the way PGA memory is allocated. The database administrator simply needs to specify the total size dedicated to PGA memory for the Oracle instance in the PGA\_AGGREGATE\_TARGET initialization parameter and set WORKAREA\_SIZE\_POLICY to AUTO.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

This lesson introduced you to performance issues with sorting and with optimizing join statements.

There are several join operations that Oracle Database supports:

- Nested loop join
- Sort-merge join
- Hash join
- Cluster join

The Oracle optimizer optimizes joins in three phases. First, the optimizer generates a list of possible join orders. Then, for each possible join order, the best join operation is selected. Finally, the Oracle optimizer determines the best access path for each row source of the execution plan.

Throwaway of rows occurs when both input row sources have more rows than the result of the join operation. You should minimize throwaway of rows to improve performance.

The star transformation is a query transformation aimed at executing star queries efficiently. Star transformations are used in data warehousing. For more information on this topic, see Appendix D (“Data Warehouse Tuning Considerations”).



# 6

## Execution Plans

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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\_XPLAN 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Objectives

You can use various analysis tools to evaluate the performance of SQL statements.

After completing this lesson, you should be able to:

- Use the EXPLAIN PLAN command to show how a statement is processed
- Use the DBMS\_XPLAN package
- Query the V\$SQL\_PLAN performance view to examine the execution plan for cursors that were recently executed
- Use the SQL\*Plus AUTOTRACE setting to show SQL 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## What Is an Execution Plan?

When a statement is executed, the optimizer performs many steps. Each of these steps either retrieves rows of data physically from the database or prepares them in some way for the user issuing the statement. The combination of steps that are used to run a statement is called an *execution plan*.

An execution plan includes an access method for each table that the statement accesses and an ordering of the tables (the join order). The optimizer also uses different methods to combine the rows from multiple tables (join method). The steps of the execution plan are not performed in the order in which they are numbered.

# Methods for Viewing Execution Plans

- **EXPLAIN PLAN**
- **SQL Trace**
- **Statspack**
- **Automatic Workload Repository**
- **V\$SQL\_PLAN**
- **SQL\*Plus AUTOTRACE**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Methods for Viewing Execution Plans

- The **EXPLAIN PLAN** command allows you to view the execution plan that the optimizer uses to execute a SQL statement. This command is very useful in improving SQL performance because it outlines the plan that the optimizer may use and inserts it in a table called **PLAN\_TABLE** without executing the SQL statement.
- The trace utility **SQL Trace** is used to measure timing statistics for a SQL statement. It helps identify potential bottlenecks. It provides statistics about a query that has been executed.
- **Statspack** is a utility that collects information and stores performance statistics data permanently in Oracle tables, which can later be used for reporting and analysis. The data collected can be analyzed using the report provided, which includes an instance health and load summary page, high-resource SQL statements, and the traditional wait events and initialization parameters.
- The **Automatic Workload Repository (AWR)** is a built-in repository in Oracle Database 10g. At regular intervals, the Oracle Database makes a snapshot of all its vital statistics and workload information and stores this in the AWR.
- The captured data allows both system-level and user-level analysis to be performed, again reducing the requirement to repeat the workload in order to diagnose problems.

## Methods for Viewing Execution Plans

- The `V$SQL_PLAN` view contains information about executed SQL statements and their execution plan.
- The `AUTOTRACE` command available in `SQL*Plus` generates the `PLAN_TABLE` output and statistics about the performance of a query. This command provides many of the same statistics as SQL Trace, such as disk reads and memory reads.

You can use the `DBMS_XPLAN` package methods to display the execution plan generated by the `EXPLAIN PLAN` command and query from `V$SQL_PLAN` and AWR.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Using Execution Plans

There are several uses for viewing execution plans:

- Determining the current execution plan
- Identifying the effect of creating an index on a table
- Finding cursors containing a certain access path (for example, full table scan or index range scan)
- Identifying indexes that are, or are not, selected by the optimizer
- Determining whether the optimizer selects the particular execution plan (for example, nested loops join) expected by the developer

You can use an execution plan to make decisions such as:

- Dropping or creating indexes
- Generating statistics on the database objects
- Modifying initialization parameter values
- Migrating the application or the database to a new release

If previous plans are kept in user-defined tables, it is then possible to identify how changes in the performance of a SQL statement can be correlated with changes in the execution plan for that statement.

## 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### DBMS\_XPLAN Package: Overview

The DBMS\_XPLAN package provides an easy way to display the output of the EXPLAIN PLAN command in several predefined formats. You can also use the DBMS\_XPLAN package to display the plan of a statement stored in the AWR. Furthermore, it provides a way to display the SQL execution plan and SQL execution run-time statistics for cached SQL cursors based on the information stored in the 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 formats and displays the contents of a plan table from V\$SQL\_PLAN.
- DISPLAY\_CURSOR formats and displays the contents of the execution plan of any loaded cursor.
- DISPLAY\_AWR formats and displays the contents of the execution plan of a stored SQL statement in the AWR.

## Using DBMS\_XPLAN to Display the Execution Plan

The methods of this package contain a `FORMAT` parameter that allows you to specify the detail level of displayed plans.

- **BASIC**: Displays the minimum information in the plan (the operation ID, the object name, and the operation option)
- **TYPICAL**: Default. Displays the most relevant information in the plan. Partition pruning, parallelism, and predicates are displayed only when available.
- **ALL**: Maximum level. Includes information displayed with the **TYPICAL** level and adds projection information as well as SQL statements generated for parallel execution servers (only if parallel).
- **SERIAL**: Similar to **TYPICAL**, except that the parallel information is not displayed, even if the plan executes in parallel.

This package runs with the privileges of the calling user, not the package owner (SYS).

The table function `DISPLAY_CURSOR` requires select privileges on the following fixed views: `V$SQL_PLAN`, `V$SESSION`, and `V$SQL_PLAN_STATISTICS_ALL`. Using the `DISPLAY_AWR` function requires `SELECT` privileges on `DBA_HIST_SQL_PLAN`, `DBA_HIST_SQLTEXT`, and `V$DATABASE`. All these privileges are automatically granted as part of the `SELECT_CATALOG_ROLE`. However, granting this role indiscriminately is not advised as it may cause security concerns.

Both the `DISPLAY` and `DISPLAY_AWR` functions accept `SQL_ID` as an argument (as shown in the third example). The `SQL_ID` of a statement can be obtained by querying `V$SQL` or `DBA_HIST_SQLTEXT`.



## EXPLAIN PLAN Command

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### EXPLAIN PLAN Command

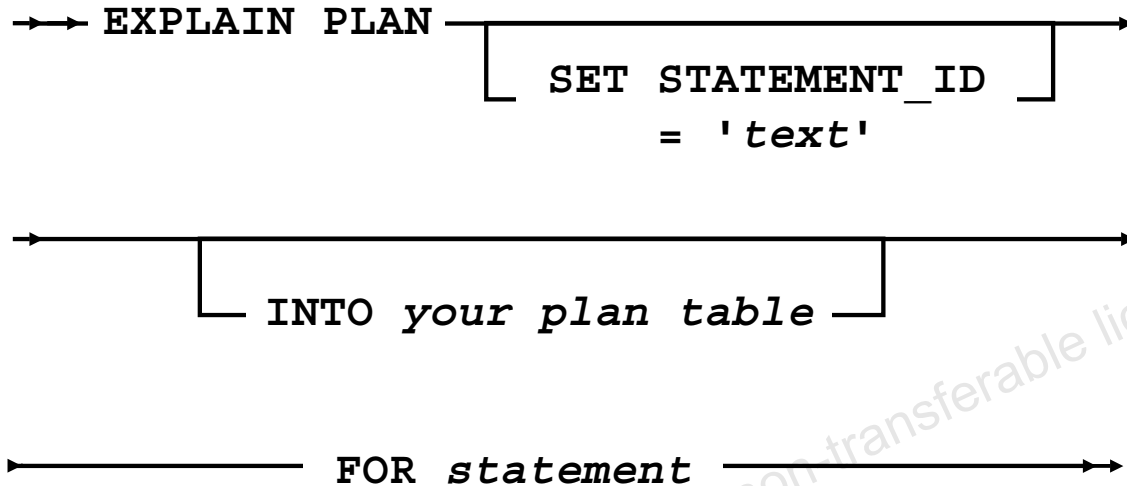
The `EXPLAIN PLAN` command is used to generate the execution plan that the optimizer uses to execute a SQL statement. It does not execute the statement but simply produces the plan that may be used and inserts this plan into a table. If you examine the plan, you can see how the Oracle Server executes the statement.

To use `EXPLAIN PLAN`, you must:

- First use the `EXPLAIN PLAN` command to explain a SQL statement
- Retrieve the plan steps by using the methods in the `DBMS_XPLAN` package

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

# EXPLAIN PLAN Command



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## EXPLAIN PLAN Command (continued)

This command inserts a row in the plan table for each step of the execution plan.

In the syntax diagram shown in the slide, the fields in *italic* have the following meanings:

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### EXPLAIN PLAN Command: Example

This command inserts the execution plan of the SQL statement in the plan table and adds the name tag demo01 for future reference. The tag is optional. You can also use the following syntax:

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

## EXPLAIN PLAN Command: Output

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

Plan hash value: 2933537672

| Id  | Operation                   | Name        | Rows | Bytes | Cost (%CPU) |
|-----|-----------------------------|-------------|------|-------|-------------|
| 0   | SELECT STATEMENT            |             | 106  | 2862  | 6 (17)      |
| 1   | MERGE JOIN                  |             | 106  | 2862  | 6 (17)      |
| 2   | TABLE ACCESS BY INDEX ROWID | DEPARTMENTS | 27   | 432   | 2 (0)       |
| 3   | INDEX FULL SCAN             | DEPT_ID_PK  | 27   |       | 1 (0)       |
| * 4 | SORT JOIN                   |             | 107  | 1177  | 4 (25)      |
| 5   | TABLE ACCESS FULL           | EMPLOYEES   | 107  | 1177  | 3 (0)       |

Predicate Information (identified by operation id):

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

18 rows selected.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### EXPLAIN PLAN Command: Output

The DISPLAY function of the DBMS\_XPLAN package can be used to format and display the last statement stored in a plan table

The slide shows the result of using the DBMS\_XPLAN package as shown on the previous page to retrieve the information from the PLAN table in that example.

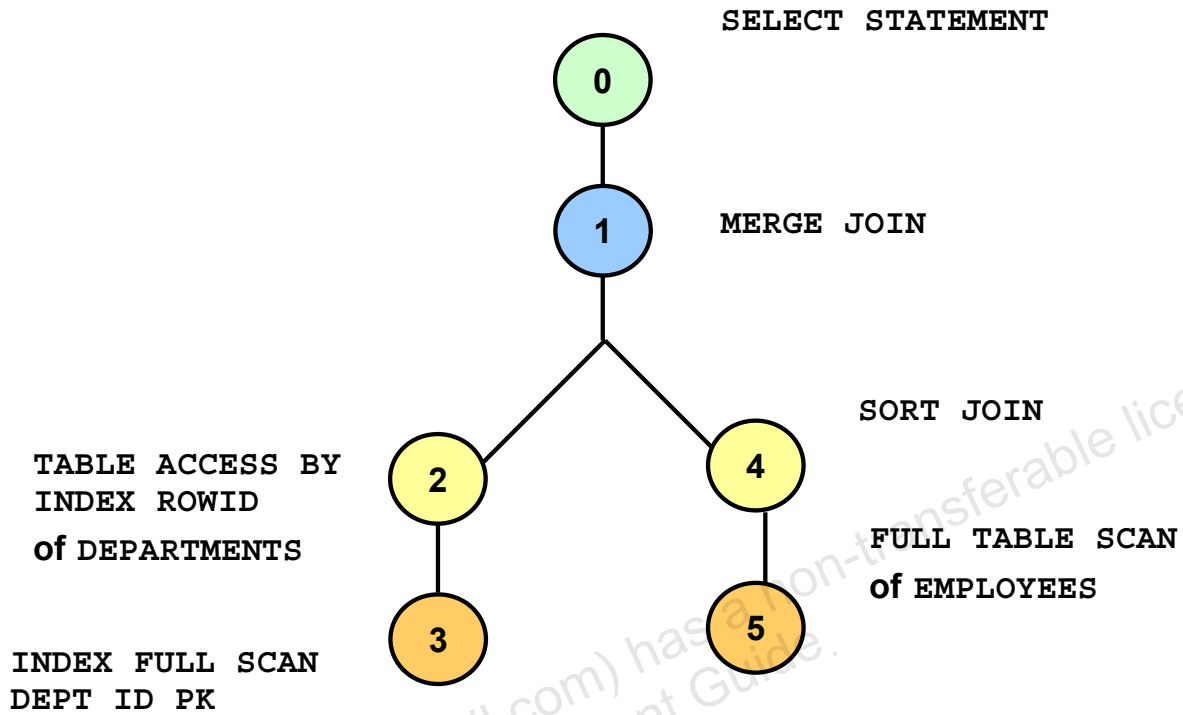
You can also use the syntax shown below to retrieve from the PLAN table.

```
SELECT plan_table_output FROM
TABLE(dbms_xplan.display('plan_table','demo01','serial'));
```

The output is the same as the one shown in the slide. In this example, you can substitute the name of another plan table instead of PLAN\_TABLE and 'demo01' represents the statement ID.

You can run the utlxpls.sql script (located in the ORACLE\_HOME/rdbms/admin/ directory) to display the EXPLAIN PLAN of the last statement explained. This script uses the DISPLAY table function from the DBMS\_XPLAN package.

# Parse Tree



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Reading an Execution Plan

Based on the execution plan, an execution tree (or *parse tree*) can be constructed to get a better idea of how a statement is processed. To construct the tree, you should start with step 1. Then find all steps with a parent of step 1 and draw them in. Continue until all steps are accounted for. The order resembles a tree structure. To each step in the execution plan, Oracle Database assigns a number representing the ID column of the `PLAN_TABLE`. Each step is depicted by a “node.” The result of each node’s operation is passed to its parent node, which uses it as input.

The sequence of steps is determined by the parent-child relationships of the steps. Each step of the execution plan either retrieves rows from the database or accepts rows as input from one or more other steps, also known as *row sources*. The child step is performed at least once and feeds the parent. When a parent has multiple children, each child is performed sequentially in order of step position. If the lower step child steps are arranged left to right, the plan can be read left to right and bottom to top.

In the diagram, the numbers correspond to the ID values in the `PLAN` table (see previous slide). The optimizer performs a `FULL TABLE SCAN` and `SORT` operation on the `EMPLOYEES` table. It retrieves the rows from the `DEPARTMENTS` table using an index scan by performing a `FULL INDEX SCAN` on the primary key column. These two result sets are then `MERGED` to get the end result for the query.

## 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Using V\$SQL\_PLAN

This view provides a way of examining the execution plan for cursors that were recently executed. The information in this view is very similar to the output from the PLAN\_TABLE. However, EXPLAIN PLAN shows a theoretical plan that can be used if this statement were to be executed, whereas V\$SQL\_PLAN contains the actual plan used. The execution plan obtained by the EXPLAIN PLAN statement can be different from the execution plan used to execute the cursor, because the cursor might have been compiled with different values of session parameters.

V\$SQL\_PLAN shows the plan for a cursor rather than for all cursors associated with a SQL statement. The difference is that a SQL statement can have more than one cursor associated with it, with each cursor further identified by a CHILD\_NUMBER. For example, the same statement executed by different users has different cursors associated with it if the object being referenced is in a different schema. Similarly, different hints can cause different cursors. The V\$SQL\_PLAN table can be used to see the different plans for different child cursors of the same statement.

**Note:** Another useful view is V\$SQL\_PLAN\_STATISTICS, which provides the execution statistics of each operation in the execution plan for each cached cursor. Also, the V\$SQL\_PLAN\_STATISTICS\_ALL view concatenates information from V\$SQL\_PLAN with execution statistics from V\$SQL\_PLAN\_STATISTICS and V\$SQL\_WORKAREA.

## V\$SQL\_PLAN Columns

|              |   |
|--------------|---|
| HASH_VALUE   | Hash value of the parent statement in the library cache                       |
| ADDRESS      | Object number of the table or the index                                       |
| CHILD_NUMBER | Child cursor number using this execution plan                                 |
| POSITION     | Order of processing for operations that all have the same PARENT_ID           |
| PARENT_ID    | ID of the next execution step that operates on the output of the current step |
| ID           | Number assigned to each step in the execution plan                            |

**Note:** This is only a partial listing of the columns.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### V\$SQL\_PLAN Columns

The view contains almost all PLAN\_TABLE columns, in addition to new columns. The columns that are also present in the PLAN\_TABLE have the same values:

- ADDRESS
- HASH\_VALUE

The columns ADDRESS and HASH\_VALUE can be used to join with V\$SQLAREA to add the cursor-specific information.

The ADDRESS, HASH\_VALUE, and CHILD\_NUMBER columns can be used to join with V\$SQL to add the child cursor-specific information.

## Querying V\$SQL\_PLAN

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

```
SQL_ID 47ju6102uvq5q, child number 0
```

```
-----
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
```

| Id  | Operation                   | Name        | Rows | Bytes | Cost (%CPU) |
|-----|-----------------------------|-------------|------|-------|-------------|
| 0   | SELECT STATEMENT            |             |      |       | 6 (100)     |
| 1   | MERGE JOIN                  |             | 106  | 2862  | 6 (17)      |
| 2   | TABLE ACCESS BY INDEX ROWID | DEPARTMENTS | 27   | 432   | 2 (0)       |
| 3   | INDEX FULL SCAN             | DEPT_ID_PK  | 27   |       | 1 (0)       |
| * 4 | SORT JOIN                   |             | 107  | 1177  | 4 (25)      |
| 5   | TABLE ACCESS FULL           | EMPLOYEES   | 107  | 1177  | 3 (0)       |

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

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

```
24 rows selected.
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Querying V\$SQL\_PLAN

You can query V\$SQL\_PLAN using the DBMS\_XPLAN.DISPLAY\_CURSOR() function to display the current or last executed statement (as shown in the example). You can pass the value of the SQL\_ID for the statement as a parameter to get the execution plan for a given statement. To obtain the SQL\_ID:

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

```
SELECT SQL_ID, SQL_TEXT FROM V$SQL
WHERE SQL_TEXT LIKE '%SELECT e.last_name,%' ;
```

```
13saxr0mmzls3 select SQL_id, sql_text from v$SQL ...
47ju6102uvq5q SELECT e.last_name, d.department_name ...
```

The FORMAT parameter controls the level of detail for the plan. In addition to the standard values (BASIC, TYPICAL, SERIAL, and ALL), there are two additional supported values to display run-time statistics for the cursor:

- RUNSTATS\_LAST: Displays the run-time statistics for the last execution of the cursor
- RUNSTATS\_TOT: Displays the total aggregated run-time statistics for all executions of a specific SQL statement since the statement was first parsed and executed



## 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### V\$SQL\_PLAN\_STATISTICS View

The V\$SQL\_PLAN\_STATISTICS view provides the actual execution statistics for every operation in the plan, such as the number of output rows and elapsed time. All statistics, except the number of output rows, are cumulative. For example, the statistics for a join operation also includes the statistics for its two inputs. The statistics in V\$SQL\_PLAN\_STATISTICS are available for cursors that have been compiled with the STATISTICS\_LEVEL initialization parameter set to ALL.

The V\$SQL\_PLAN\_STATISTICS\_ALL view contains memory-usage statistics for row sources that use SQL memory (sort or hash join). This view concatenates information in V\$SQL\_PLAN with execution statistics from V\$SQL\_PLAN\_STATISTICS and V\$SQL\_WORKAREA.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Automatic Workload Repository

The AWR is part of the new intelligent infrastructure of Oracle Database 10g that is used by many of the new solutions (such as ADDM for analysis). The AWR automatically collects, processes, and maintains system-performance statistics for problem-detection and self-tuning purposes and stores the statistics persistently in the database.

The statistics collected and processed by the AWR include:

- Object statistics that determine both access and usage statistics of database segments
- Time-model statistics based on time usage for activities, displayed in the V\$SYS\_TIME\_MODEL and V\$SESS\_TIME\_MODEL views
- Some of the system and session statistics collected in the V\$SYSSTAT and V\$SESSTAT views
- SQL statements that produce the highest load on the system, based on criteria such as elapsed time, CPU time, buffer gets, and so on
- Active Session History (ASH) statistics, representing the history of recent sessions

## Automatic Workload Repository

The database automatically generates snapshots of the performance data once every hour and collects the statistics in the workload repository. The data in the snapshot interval is then analyzed by ADDM. The ADDM compares the differences between snapshots to determine which SQL statements to capture based on the effect on the system load. This reduces the number of SQL statements that need to be captured over time.

**Note:** By using PL/SQL packages such as `DBMS_WORKLOAD_REPOSITORY` or Oracle Enterprise Manager, you can manage the frequency and retention period of SQL that is stored in the AWR.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# Managing AWR with PL/SQL

- **Creating snapshots**
- **Dropping snapshots**
- **Managing snapshot settings**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Managing AWR with PL/SQL

Although the primary interface for managing the AWR is Oracle Enterprise Manager Database Control, monitoring functions can be managed with procedures in the DBMS\_WORKLOAD\_REPOSITORY package.

Snapshots are automatically generated for an Oracle database; however, you can use DBMS\_WORKLOAD\_REPOSITORY procedures to manually create, drop, and modify the snapshots and baselines that are used by automatic database diagnostic monitoring. Snapshots and baselines are sets of historical data for specific time periods that are used for performance comparisons. To invoke these procedures, a user must be granted the DBA role.

### Creating Snapshots

You can manually create snapshots with the CREATE\_SNAPSHOT procedure if you want to capture statistics at times different than those of the automatically generated snapshots. Here is an example:

```
Exec DBMS_WORKLOAD_REPOSITORY.CREATE_SNAPSHOT ('ALL');
```

In this example, a snapshot for the instance is created immediately with the flush level specified to the default flush level of TYPICAL. You can view this snapshot in the DBA\_HIST\_SNAPSHOT view.

## Managing AWR with PL/SQL (continued)

### Dropping Snapshots

You can drop a range of snapshots using the `DROP_SNAPSHOT_RANGE` procedure. To view a list of the snapshot IDs along with database IDs, check the `DBA_HIST_SNAPSHOT` view. For example, you can drop the following range of snapshots:

```
Exec DBMS_WORKLOAD_REPOSITORY.DROP_SNAPSHOT_RANGE -  
(low_snap_id => 22, high_snap_id => 32, dbid => 3310949047);
```

In the example, the range of snapshot IDs to drop is specified from 22 to 32. The optional database identifier is 3310949047. If you do not specify a value for `dbid`, the local database identifier is used as the default value.

Active Session History (ASH) data that belongs to the time period specified by the snapshot range is also purged when the `DROP_SNAPSHOT_RANGE` procedure is called.

### Modifying Snapshot Settings

You can adjust the interval and retention of snapshot generation for a specified database ID. However, note that this can affect the precision of the Oracle diagnostic tools.

The `INTERVAL` setting specifies how often (in minutes) snapshots are automatically generated. The `RETENTION` setting specifies how long (in minutes) snapshots are stored in the workload repository. To adjust the settings, use the `MODIFY_SNAPSHOT_SETTINGS` procedure, as in the following example:

```
Exec DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS( -  
    retention => 43200, interval => 30, dbid => 3310949047);
```

In this example, the retention period is specified as 43,200 minutes (30 days), and the interval between each snapshot is specified as 30 minutes. If `NULL` is specified, the existing value is preserved. The optional database identifier is 3310949047. If you do not specify a value for `dbid`, the local database identifier is used as the default value. You can check the current settings for your database instance with the `DBA_HIST_WR_CONTROL` view.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## AWR Views

You can view the AWR data through Oracle Enterprise Manager screens or AWR reports. However, you can also view the statistics directly from the following views:

**V\$ACTIVE\_SESSION\_HISTORY:** This view displays active database session activity, sampled once every second.

**V\$metric** views provide metric data to track the performance of the system. The metric views are organized into various groups, such as event, event class, system, session, service, file, and tablespace metrics. These groups are identified in the **V\$METRICGROUP** view.

The **DBA\_HIST** views contain historical data stored in the database. This group of views includes:

- **DBA\_HIST\_ACTIVE\_SESS\_HISTORY** displays the history of the contents of the in-memory active session history for recent system activity.
- **DBA\_HIST\_BASELINE** displays information about the baselines captured on the system.
- **DBA\_HIST\_DATABASE\_INSTANCE** displays information about the database environment.
- **DBA\_HIST\_SNAPSHOT** displays information about snapshots in the system.
- **DBA\_HIST\_SQL\_PLAN** displays SQL execution plans.
- **DBA\_HIST\_WR\_CONTROL** displays the settings for controlling AWR.

## Querying the AWR

```
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

| Id | Operation         | Name        | Rows | Bytes | Cost (%CPU) | Time     |
|----|-------------------|-------------|------|-------|-------------|----------|
| 0  | SELECT STATEMENT  |             |      |       | 6 (100)     |          |
| 1  | HASH JOIN         |             | 11   | 968   | 6 (17)      | 00:00:01 |
| 2  | TABLE ACCESS FULL | DEPARTMENTS | 11   | 220   | 2 (0)       | 00:00:01 |
| 2  | TABLE ACCESS FULL | DEPARTMENTS | 11   | 220   | 2 (0)       | 00:00:01 |
| 3  | TABLE ACCESS FULL | EMPLOYEES   | 107  | 7276  | 3 (0)       | 00:00:01 |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Querying the AWR

You can use the DBMS\_XPLAN.DISPLAY\_AWR() function to display all stored plans in the AWR. In the example shown, you are passing in a SQL\_ID as an argument. The steps to complete this example are as follows:

1. Execute the SQL statement.

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

2. Query V\$SQL\_TEXT to obtain the SQL\_ID.

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

```
SQL_ID          SQL_TEXT
-----
```

```
F8tc4anpz5cdb select sql_id, sql_text from v$SQL ...
454rug2yva18w select /* example */ * from ...
```

3. Using the SQL\_ID, verify that this statement has been captured in the DBA\_HIST\_SQLTEXT dictionary view. If the query does not return rows, then it indicates that the statement has not yet been loaded in the AWR.

## Querying the AWR (continued)

```
SQL> SELECT SQL_ID, SQL_TEXT FROM dba_hist_sqltext WHERE
      SQL_ID = ' 454rug2yva18w';
no rows selected
```

You can take a manual AWR snapshot rather than wait for the next snapshot (which occurs every hour). Then check to see if it has been captured in

DBA\_HIST\_SQLTEXT:

```
SQL> exec dbms_workload_repository.create_snapshot;
```

PL/SQL procedure successfully completed.

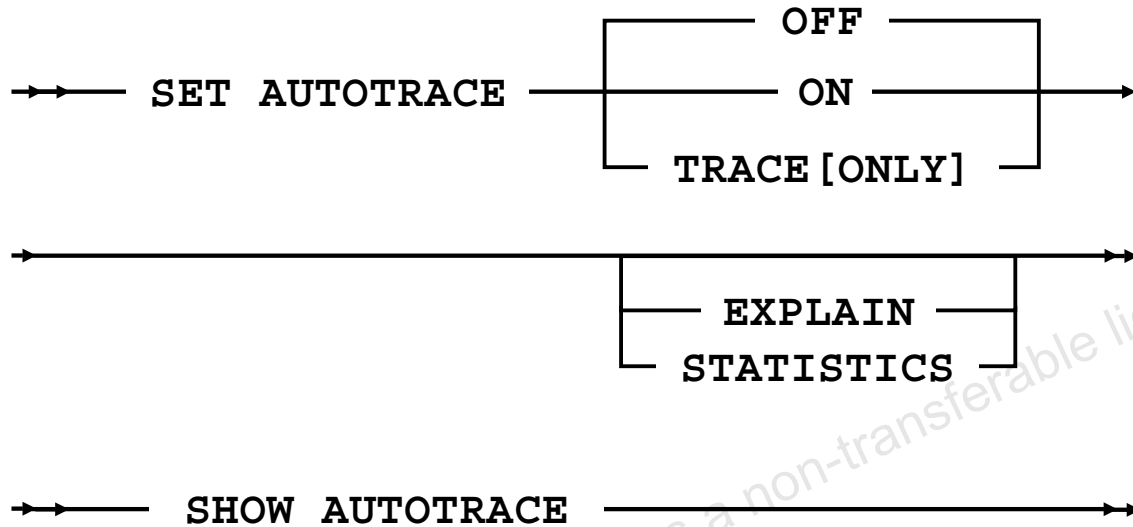
```
SQL> SELECT SQL_ID, SQL_TEXT FROM dba_hist_sqltext
      WHERE SQL_ID = ' 454rug2yva18w';
SQL_ID          SQL_TEXT
-----
454rug2yva18w   select /* example */ * from ...
```

4. Use the DBMS\_XPLAN.DISPLAY\_AWR () function to retrieve the execution plan:

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



# SQL\*Plus AUTOTRACE



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL\*Plus AUTOTRACE

In SQL\*Plus, you can automatically obtain the execution plan and some additional statistics about the running of a SQL command by using the AUTOTRACE setting. Unlike the EXPLAIN PLAN command, the statement is actually run. However, you can choose to suppress statement execution by specifying AUTOTRACE TRACEONLY EXPLAIN.

AUTOTRACE is an excellent diagnostic tool for SQL statement tuning. Because it is purely declarative, it is easier to use than EXPLAIN PLAN.

### Command Options

|            |  |
|------------|--|
| OFF        | Disables autotracing SQL statements                                |
| ON         | Enables autotracing SQL statements                                 |
| TRACEONLY  | Enables autotracing SQL statements and suppresses statement output |
| EXPLAIN    | Displays execution plans but does not display statistics           |
| STATISTICS | Displays statistics but does not display execution plans           |

**Note:** If both the EXPLAIN and STATISTICS command options are omitted, execution plans and statistics are displayed by default.

# 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:

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Prerequisites for Using AUTOTRACE

To access STATISTICS data, you must have access to several dynamic performance tables. The DBA can grant access by using the role `plustrace` created in the `plustrce.sql` script. (This script name may vary by operating system.) The DBA must run the script as the SYS user and grant the role to users who want to use the STATISTICS option of AUTOTRACE.

## Controlling the AUTOTRACE Execution Plan Layout

The execution plan consists of four columns displayed in the following order:

|                      |   |
|----------------------|---|
| ID_PLUS_EXP          | Line number of each step                      |
| PARENT_ID_PLUS_EXP   | Parent step line number                       |
| PLAN_PLUS_EXP        | Report step                                   |
| OBJECT_NODE_PLUS_EXP | Database links or parallel query servers used |

You can change the format of these columns, or suppress them, by using the SQL\*Plus COLUMN command. For additional information, see *Oracle SQL\*Plus User's Guide and Reference*.

## 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
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### SQL\*Plus AUTOTRACE: Statistics

AUTOTRACE displays several statistics, not all of them relevant to the discussion at this stage. The most important statistics are the following:

|                 |  |
|-----------------|--|
| db block gets   | Number of logical I/Os for current gets                |
| consistent gets | Reads of buffer cache blocks                           |
| physical reads  | Number of blocks read from disk                        |
| redo size       | Amount of redo generated (for DML statements)          |
| sorts (memory)  | Number of sorts performed in memory                    |
| sorts (disk)    | Number of sorts performed using temporary disk storage |

**Note:** DB block gets are reads of the current blocks in the buffer cache. Consistent gets are reads of buffer cache blocks that have undo data. Physical reads are reads of blocks from disk. DB block gets, consistent gets, and physical reads are the three statistics that are usually monitored. These should be low compared to the number of rows retrieved. Sorts should be performed in memory rather than on disk.

# 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 **SQL\*Plus AUTOTRACE** to run statements and display execution plans and statistics

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

Tuning involves careful analysis of execution plans and execution statistics. Several tools are available to help you test and diagnose performance problems with a SQL statement:

- **EXPLAIN PLAN** determines how the optimizer processes a statement (the statement is not executed) and stores the execution plan in the plan table.
- You can also use **V\$SQL\_PLAN** to view the execution plan for recently executed cursors.
- The **SET AUTOTRACE** command in **SQL\*Plus** enables you to view the execution plan and a variety of statistics about the execution whenever you execute a SQL statement.

You can use several other tools to monitor performance, alerting you to the need to tune SQL statements. Some of these tools are covered in the remaining lessons of this course.

## Practice 6: Overview

**This practice covers the following topics:**

- **Using AUTOTRACE**
- **Using EXPLAIN PLAN**
- **Using AWR**
- **Retrieving the execution plan using DBMS\_XPLAN**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

swati bhatia (swati.june@gmail.com) has a non-transferable license to  
use this Student Guide.

# 7

## Gathering Statistics

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Objectives

- Describe the importance of statistics.
- Identify table, index, column; view these statistics in the Oracle Database data dictionary views.
- Describe the Automatic Statistics Gathering mechanism.
- Use the DBMS\_STATS package to collect and manage optimizer statistics:
  - Identify predicate selectivity calculations, assuming even data distribution.
  - Identify the consequences of using bind variables for predicate selectivity.
  - Create histograms for columns with skewed data.
  - Choose appropriate values for:
    - Sample size
    - Number of histogram buckets



# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## What Are Optimizer Statistics?

Optimizer statistics are a collection of data that describes details about the database and the objects in the database. These statistics are used by the query optimizer to choose the best execution plan for each SQL statement.

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

To execute a SQL statement, Oracle Database might need to perform many steps. Each of these steps either retrieves rows of data physically from the database or prepares them in some way for the user issuing the statement. The combination of the steps that Oracle Database uses to execute a statement is called an *execution plan*. An execution plan includes an *access path* for each table that the statement accesses and an ordering of the tables (the *join order*) with the appropriate *join method*.

# Types of Optimizer Statistics

- **Object statistics**
  - Table statistics
  - Column statistics
  - Index statistics
- **System statistics**
  - I/O performance and utilization
  - CPU performance and utilization

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Types of Optimizer Statistics

To effectively diagnose performance problems, statistics must be available. Oracle Database generates many types of cumulative statistics for the system, sessions, and individual SQL statements. Oracle Database also tracks cumulative statistics on segments and services.

When analyzing a performance problem in any of these scopes, you typically look at the change in statistics (delta value) over the period of time you are interested in. Specifically, you look at the difference between the cumulative value of a statistic at the start of the period and the cumulative value at the end.

### Table statistics

- Number of rows
- Number of blocks
- Average row length

### Column statistics

- Number of distinct values (NDV) in column
- Number of nulls in column
- Data distribution (histogram)

## **Types of Optimizer Statistics (continued)**

### **Index statistics**

- Number of leaf blocks
- Levels

### **System costing statistics**

- I/O performance and utilization
- CPU performance and utilization

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# How Statistics Are Gathered

- **Automatic statistics gathering**
  - `GATHER_STATS_JOB`
- **Manual statistics gathering**
  - `DBMS_STATS` package
- **Dynamic sampling**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## How Statistics Are Gathered

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

# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Automatic Statistics Gathering

Oracle Database 10g automatically gathers statistics on all database objects and maintains those statistics in a regularly scheduled maintenance job. Automated statistics collection eliminates all of the manual tasks associated with managing the optimizer statistics, and significantly reduces the chances of getting poor execution plans because of missing or stale statistics. DML monitoring is enabled by default and is used by the Automatic Statistics Gathering feature to determine which objects have stale statistics as well as their degree of staleness. This information is used to identify candidates for statistics update.

Optimizer statistics are automatically gathered with the GATHER\_STATS\_JOB job. This Scheduler job does the following:

- Gathers statistics on all objects in the database that have missing or stale statistics in the predefined maintenance window
- Determines the appropriate sample size for each object
- Creates histograms as required

The job is created automatically at database creation time and when upgrading from an earlier release. It is managed by the Scheduler.

**Note:** System statistics are *not* gathered by the automatic gathering job.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Using the `DBMS_STATS` Package

The `DBMS_STATS` package enables you to manually generate and manage statistics for optimization. You can use this package to gather, modify, view, export, import, and delete statistics. You can also use this package to identify or name gathered statistics. The `DBMS_STATS` package can gather statistics on indexes, tables, columns, and partitions, as well as statistics on all schema objects in a schema or database.

`DBMS_STATS` gathers only statistics needed for optimization; it does not gather other statistics. For example, the table statistics gathered by `DBMS_STATS` include the number of rows, number of blocks currently containing data, and average row length, but not the number of chained rows, average free space, or number of unused data blocks. Manual statistics gathering is recommended for frequently accessed objects that change significantly between the automatic statistics gathering intervals. Example new tables or tables that have bulk operations performed on them.

# Managing Automatic Statistics Collection

- **Job configuration options**
- **Statistics-collection configuration options**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Managing Automatic Statistics Collection

There are two parts to statistics gathering:

- Job-configuration aspect
- Statistics-collection aspect

The job-configuration aspect allows you to maintain the job windows and enable or disable the statistics gathering jobs.

The statistics-collection aspect allows you to either let the database automatically gather statistics and make decisions on sampling, histograms, parallelization, and so on.

# Job Configuration Options

- **Setting status: ENABLED or DISABLED**
- **Maintaining schedule: Maintenance window**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Job Configuration Options

The job configuration options are:

- Setting the status of the Automatic Statistics Gathering mechanism to ENABLED or DISABLED. The default status is ENABLED.
- Scheduling the 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;

/
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Managing the Job Scheduler

The DBA\_SCHEDULER\_JOBS data dictionary view gives information about the GATHER\_STATS\_JOB. You can verify that the Automatic Statistics Gathering job exists by querying this view as shown in the first example. The output is as follows:

| OWNER | JOB_NAME         | ENABLED |
|-------|------------------|---------|
| SYS   | GATHER_STATS_JOB | TRUE    |

GATHER\_STATS\_JOB automatically refreshes stale statistics or statistics on new objects every night by default. You can change the frequency and maintenance window as suited to your system.

If you want to disable Automatic Statistics Gathering, you can disable GATHER\_STATS\_JOB as shown in the second example. This is not recommended. However if you disable statistics gathering, the statistics already in the dictionary are frozen until Automatic Statistics Gathering is enabled again or statistics are gathered manually.

# Managing the Maintenance Window

- **WEEKNIGHT\_WINDOW**
- **WEEKEND\_WINDOW**

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Managing the Maintenance Window

Two Scheduler windows are predefined on installation of Oracle Database 10g:

- **WEEKNIGHT\_WINDOW** starts at 10 p.m. and ends at 6 a.m. every Monday through Friday.
- **WEEKEND\_WINDOW** covers the whole days Saturday and Sunday.

Together these windows constitute the **MAINTENANCE\_WINDOW\_GROUP**, in which all system maintenance tasks are scheduled. Oracle Database uses the maintenance windows for automatic statistics collection and for other internal system maintenance jobs. If you are using the Resource Manager, you can also assign resource plans to these windows.

You can adjust the predefined maintenance windows to a time suitable to your database environment using the **DBMS\_SCHEDULER.SET\_ATTRIBUTE** procedure.

In the slide is an example of changing a maintenance window.

A Scheduler program **GATHER\_STATS\_PROG** and Scheduler job **GATHER\_STATS\_JOB** are predefined on installation of Oracle Database. **GATHER\_STATS\_PROG** collects optimizer statistics for all objects in the database for which there are no statistics or only stale statistics. **GATHER\_STATS\_JOB** is defined on **GATHER\_STATS\_PROG** and is scheduled to run in the **MAINTENANCE\_WINDOW\_GROUP**.

# Changing the GATHER\_STATS\_JOB Schedule

ORACLE Enterprise Manager 10g Database Control

Database: orcl > Scheduler Windows

Scheduler Windows

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

View Edit Delete Create Like Go Create

| Select                           | Name             | Resource Plan | Enabled | Next Open Date           | End Date | Duration (min) | Active | Description                           |
|----------------------------------|------------------|---------------|---------|--------------------------|----------|----------------|--------|---------------------------------------|
| <input checked="" type="radio"/> | WEEKNIGHT_WINDOW |               | TRUE    | Dec 8, 2003 10:00:00 PM  |          | 480            | FALSE  | Weeknight window for maintenance task |
| <input type="radio"/>            | WEEKEND_WINDOW   |               | TRUE    | Dec 13, 2003 12:00:00 AM |          | 2880           | FALSE  | Weekend window for maintenance task   |

Database | Setup | Preferences | Help | Logout

Copyright © 1996, 2003, Oracle. All rights reserved.  
About Oracle Enterprise Manager 10g Database Control

## Changing the GATHER\_STATS\_JOB Schedule

You can always customize the open times of the defined maintenance windows. For example, you can change their time interval and repeat frequency. You can also add resource plans to these windows to control the resources used by GATHER\_STATS\_JOB.

On the Database Control home page, click the Administration tab, and then click the Windows link in the Scheduler section. This brings you to the Scheduler Windows page, where you can select a window and click the Edit button to change its characteristics. On this page, you can also open and close a particular window. To do this, select the corresponding action in the drop-down list for a particular window, and then click Go.

**Note:** Although it is not recommended, you can disable Automatic Statistics Gathering from Database Control by going to the Jobs page in the Administration tab and then disabling GATHER\_STATS\_JOB.

# Statistics Collection Configuration

- **DML monitoring**
- **Sampling**
- **Degree of parallelism**
- **Histograms**
- **Cascade**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Statistics Collection Configuration

Statistics collection configuration consists of:

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

# 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 `STATISTICS_LEVEL` is set to `TYPICAL` or `ALL`
- **You can:**
  - View the information about DML changes in the `USER_TAB_MODIFICATIONS` view
  - Use `DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO` to update the view with current information
  - Use `GATHER_DATABASE_STATS` or `GATHER_SCHEMA_STATS` for manual statistics gathering for tables with stale statistics when `OPTIONS` is set to `GATHER STALE` or `GATHER AUTO`

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## DML Monitoring

Oracle Database uses the DML monitoring facility for automatically monitoring objects for stale or missing statistics. This monitoring is enabled by default when `STATISTICS_LEVEL` is set to `TYPICAL` or `ALL`. It tracks the approximate number of DML statements for that table as well as tracking whether the table has been truncated since the last time statistics were gathered. Objects are monitored for DML transactions. When significant changes occur, statistics are automatically gathered by `GATHER_STATS_JOB` to keep them current.

The information about changes to tables can be viewed in the `USER_TAB_MODIFICATIONS` view. Following data modification, there may be a few minutes' delay while the database propagates the information to this view. Use the `DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO` procedure to immediately reflect the outstanding monitored information that is kept in the memory.

For gathering statistics manually, the `DBMS_STATS.GATHER_DATABASE_STATS` or `GATHER_SCHEMA_STATS` procedures can gather new statistics for tables with stale statistics when the `OPTIONS` parameter is set to `GATHER STALE` or `GATHER AUTO`. If a monitored table has been modified more than 10%, then these statistics are considered stale and are gathered again.

## DML Monitoring (continued)

The values for the `OPTIONS` parameter are as follows:

- `GATHER STALE`: Gathers statistics on tables with stale statistics
- `GATHER`: Gathers statistics on all tables (default)
- `GATHER EMPTY`: Gathers statistics on only those tables without statistics
- `LIST STALE`: Creates a list of tables with stale statistics
- `LIST EMPTY`: Creates a list of tables that do not have statistics
- `GATHER AUTO`: Gathers all the statistics for the objects of a specific schema or database (with `DBMS_STATS.GATHER_DATABASE_STATS()`) that are not up-to-date

The `GATHER STALE` option gathers statistics for only those tables that have stale statistics and that you have enabled for monitoring. The `GATHER STALE` option maintains up-to-date statistics for the optimizer. Using this option at regular intervals also avoids the overhead associated with gathering statistics on all tables at one time.

The `GATHER` option can incur much more overhead because this option generally gathers statistics for a greater number of tables than does `GATHER STALE`.

# 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.**

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS  
( 'SH', DBMS_STATS.AUTO_SAMPLE_SIZE );
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Sampling

Sampling is an important technique for gathering statistics. Gathering statistics without sampling requires full table scans and sorts of entire tables. Sampling minimizes the resources necessary to gather statistics, especially on large objects. Sampling is recommended when there is a need to conserve the resources to scan an entire object to collect statistics and when the distribution of data is fairly uniform and is represented accurately by a sample.

## Sampling (continued)

Automatic Statistics Gathering performs sampling using default sample sizes that are set at the database level while gathering statistics. By default, autosampling is enabled. This allows the Oracle Database to determine the best sample size necessary for good statistics, based on the statistical properties of the object. Because each type of statistics has different requirements, the size of the actual sample taken may not be the same across the tables, columns, or indexes. You may choose to set different sample sizes using the `ESTIMATE_PERCENT` option provided in the `DBMS_STATS` package methods if you consider the default samples to be too small. The sampling percentage can be set to any value; however, setting `ESTIMATE_PERCENT` to the value `DBMS_STATS.AUTO_SAMPLE_SIZE` (which is the default) maximizes performance gains while achieving necessary statistical accuracy. When the `ESTIMATE_PERCENT` value is set to a constant percentage, the `DBMS_STATS` procedures may automatically increase the sampling percentage if the specified percentage did not produce a large enough sample. This ensures the stability of the estimated values by reducing fluctuations. The example in the slide shows how to collect table and column statistics for all tables in the SH schema with automatic sampling.



# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Degree of Parallelism

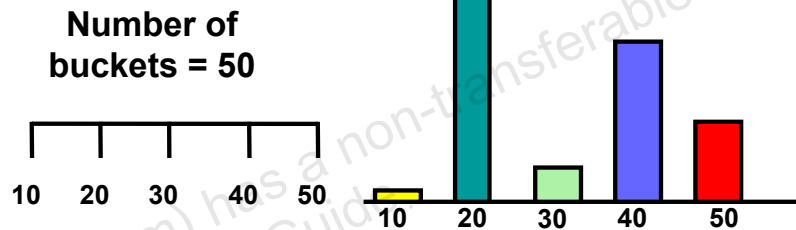
The statistics-gathering operations can run either serially or in parallel. The database makes a decision on whether statistics should be gathered in serial or parallel. If statistics gathering is performed in parallel, then the database also decides the degree of parallelism. Parallel statistics gathering can be used in conjunction with sampling.

By default, the appropriate degree of parallelism is determined automatically when collecting statistics. The default can be modified using the `DBMS_STATS` package's `DEGREE` parameter. When set to `DBMS_STATS.AUTO_DEGREE` (the recommended auto value), the database determines the degree of parallelism automatically. It will be either 1 (serial execution) or the value of `DEFAULT_DEGREE`, which is the system default value based on the number of CPUs and initialization parameters according to the size of the object.

# Histograms

- 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

| Column Value | Count of Rows |
|--------------|---------------|
| 10           | 10            |
| 20           | 1050          |
| 30           | 126           |
| 40           | 567           |
| 50           | 248           |



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Histograms

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

The optimizer must estimate the number of rows processed by a given query. This is achieved partly by estimating the selectivity of the query's predicates. The accuracy of these estimates depends on the optimizer's knowledge of the data distribution. Without histograms, an even distribution is assumed. If a histogram is available on a column, then the estimator uses it instead of the number of distinct 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:

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Creating Histograms

The Automatic Statistics Collection mechanism is configured by default to create histograms on appropriate columns as needed. Histograms are automatically created on skewed columns by the Oracle Database. It is recommended that you use this default setting and have the database automatically decide which columns need histograms. This default behavior can be modified by setting the METHOD\_OPT option of the DBMS\_STATS package. For example, the following statement sets the database default to automatic histogram creation:

```
Exec dbms_stats.set_param (method_opt => 'for all columns size
auto');
```

However, if you have changed the default configuration or have determined that additional histograms are needed, use the DBMS\_STATS package to generate them manually.

The slide example shows how to manually create a histogram with 50 buckets. The SIZE keyword declares the maximum number of buckets for the histogram. Alternatively, use the following statement to have the database determine the correct number of buckets for all columns.

```
Exec dbms_stats.gather_table_stats ('sh', 'products', method_opt
=> 'for columns size auto prod_list_price');
```

## Choosing the Number of Buckets

The default number of buckets is 75, a value with an appropriate level of detail for most data distributions. However, because the number of buckets in the histogram, the sample size, and the data distribution all affect the usefulness of a histogram, you may need to experiment with different numbers of buckets to obtain the best results.

If the number of frequently occurring distinct values in a column is relatively small, then you should set the number of buckets to be greater than the number of frequently occurring distinct values. Note that the Oracle Server never creates more buckets than the number of distinct column values, and the maximum is 254 buckets.

Several of the DBMS\_STATS package procedures that can be used for statistics collection have a parameter called `method_opt`. This parameter can take values of:

- REPEAT: The Oracle Database collects histograms on only those columns that already have histograms
- AUTO: The Oracle Database determines the columns to collect histograms based on data distribution and the workload of the columns.
- SKEWONLY: The Oracle Database determines the columns to collect histograms based on the data distribution of the columns.

## Choosing a Sample Size

Because statistics are being gathered automatically in Oracle Database 10g, columns with skewed data are appropriately sampled and have histograms created for them. However, if you decide to create histograms manually, the number of rows sampled should be at least 100 times the number of buckets in the histogram. In other words, the average number of sampled rows per bucket should be at least 100. You should not combine a very detailed histogram with a minimal sample size.

# Viewing Histogram Statistics

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

1

```
SELECT column_name, num_distinct,
       num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE column_name = 'PROD_LIST_PRICE';
```

2

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Viewing Histogram Statistics

Example 1 in the slide shows the columns with histograms, the type of histogram, and the number of buckets in the SH schema. You can also query by TABLE\_NAME and COLUMN\_NAME (as shown in example 2) as predicates to view table- or column-specific information.

The results are as follows:

1.

| COLUMN_NAME            | NUM_DISTINCT | NUM_BUCKETS | HISTOGRAM       |
|------------------------|--------------|-------------|-----------------|
| PROD_ID                | 72           | 72          | FREQUENCY       |
| CALENDAR_YEAR          | 4            | 4           | FREQUENCY       |
| CUST_MARITAL_STATUS    | 11           | 11          | FREQUENCY       |
| CUST_STATE_PROVINCE_ID | 146          | 75          | HEIGHT BALANCED |
| ...                    |              |             |                 |

2.

| COLUMN_NAME     | NUM_DISTINCT | NUM_BUCKETS | HISTOGRAM       |
|-----------------|--------------|-------------|-----------------|
| PROD_LIST_PRICE | 42           | 50          | HEIGHT BALANCED |

## 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
- **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.
  - Do not use histograms for queries with bind variables.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Histogram Tips

Histograms can affect performance and should be used only when they substantially improve query plans. For uniformly distributed data, the optimizer can make fairly accurate guesses about the cost of executing a particular statement without the use of histograms.

It is recommended that you specify the default `DBMS_STATS` histogram option so that the database determines when to create histograms. The default option is `FOR ALL COLUMNS SIZE AUTO`. If you choose to create them manually, you should create histograms on columns that are used frequently in `WHERE` clauses of queries, on columns that have a highly skewed data distribution, and on columns that are indexed.

Histograms, like all other optimizer statistics, are static. They are useful only when they reflect the current data distribution of a given column. (The data in the column can change as long as the distribution remains constant.) If the data distribution of a column changes frequently, you must recompute its histogram frequently.

## When Not to Use Histograms

Histogram data is persistent, so histograms can be expensive to store. The space required to save the data depends on the sample size.

Use histograms only when they substantially improve performance. Histograms should not be used when:

- The column is not used in the `WHERE` clause of queries
- The column is unique and is used only with equality queries
- All the predicates on the column use bind variables
- The column data is uniformly distributed

Histograms are not useful for columns with the following characteristics:

- All predicates on the column use bind variables.
- Column data is uniformly distributed.
- Column is unique and is used only with equality predicates.

Using histograms where bind variables are used may lead to unpredictable performance due to bind variable peeking.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Bind Variable Peeking

When a SQL statement is executed for the first time, the query optimizer “peeks” at the values of user-defined bind variables. This allows the optimizer to determine the selectivity of any WHERE clause condition, as well as to determine if literals have been used instead of bind variables. On subsequent invocations of the cursor, no peeking takes place; the cursor is shared based on the standard cursor-sharing criteria, even if subsequent invocations use different bind values.

When bind variables are used in a statement, it is assumed that cursor sharing is intended and that different invocations are supposed to use the same execution plan. If different invocations of the cursor would significantly benefit from different execution plans, then bind variables may have been used inappropriately in the SQL statement.



## 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Cascading to Indexes

By default, the database is configured to automatically decide whether to gather index statistics while gathering statistics on the parent tables. If you want to change this behavior, you set the CASCADE parameter in the DBMS\_STATS procedures.

When set to TRUE, this parameter gathers statistics on all the indexes for the tables when statistics are gathered. Index-statistics gathering is not parallelized. Using this option is equivalent to running the GATHER\_INDEX\_STATS procedure on each of the table's indexes.

Use the constant DBMS\_STATS.AUTO\_CASCADE to have the Oracle Database determine whether index statistics are collected or not. This is the default, which can be changed using the SET\_PARAM procedure.

#### Automatic Statistics Gathering During Index Creation/Rebuild

Oracle Database automatically collects index statistics whenever an index is created or rebuilt. Exceptions are domain indexes, or indexes created while the base table is empty.

## 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');
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Managing Statistics Collection: Example

The first example uses the DBMS\_STATS package to gather statistics on the CUSTOMERS table of the SH schema. This example uses some of the options discussed in the previous slides.

#### Setting Parameter Defaults

You can use the SET\_PARAM procedure in DBMS\_STATS to set default values for parameters of all DBMS\_STATS procedures. The second example in the slide shows this usage. You can also use the GET\_PARAM function to get the current default value of a parameter.

#### Parameters

CASCADE: The default value for CASCADE set by SET\_PARAM is not used by export/import procedures. It is used only by gather procedures.

DEGREE

ESTIMATE\_PERCENT

METHOD\_OPT

NO\_INVALIDATE

**Note:** The granularity option is provided only for backward compatibility.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## When to Gather Manual Statistics

As described in previous slides, the Automatics Statistics Gathering mechanism keeps all the statistics current. It is very important to determine when and how often to gather new statistics. The default gathering interval is nightly, but you can choose to change this interval to suit your business needs. The frequency of collection intervals should balance the task of providing accurate statistics for the optimizer against the processing overhead incurred by the statistics collection process.

Manual statistics gathering incurs an additional overhead that affects performance. However, manual statistics gathering may be required in certain cases. For example, the statistics on tables that are significantly modified during the day may become stale, because Automatic Statistics Gathering runs by default only during an overnight batch window. There are typically two types of such objects:

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

## Statistics Gathering: Manual Approaches

- **Dynamic sampling:**

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

- **Manual statistics collection:**

```
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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Statistics Gathering: Manual Approaches

For highly volatile tables, use the following approaches.

- The statistics on these tables can be set to NULL. When Oracle Database encounters a table with no statistics, Oracle Database dynamically gathers the necessary statistics as part of query optimization. This dynamic sampling feature is controlled by the `OPTIMIZER_DYNAMIC_SAMPLING` parameter, and this parameter should be set to a value of 2 or higher. The default value is 2. The statistics can be set to NULL by deleting and then locking the statistics.
- Analyze the pattern for the object and identify its maximum data state and gather statistics when the maximum data exists in the object. Then lock these statistics. Once locked, these statistics will be used even if the object shrinks. However, if you get good performance when an object is very large, then the performance will likely not be greatly affected when the object shrinks.

For tables that are being substantially modified in batch operations, such as with bulk loads, statistics should be gathered on those tables as part of the batch operation. The `DBMS_STATS` procedure should be called as soon as the load operation completes.

You also need to gather statistics manually after new object creation. The Automatic Statistics Gathering process should be used at all other times.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Dynamic Sampling

Dynamic sampling offers the opportunity to collect statistics for segments with either missing statistics or stale statistics. A sample is taken at run time from the segment and is used to estimate the statistics for that segment.

For example, if a new table has been created and no statistics exist for it, then in the interval until the next Automatic Statistics Gathering event, the optimizer generates statistics for the table dynamically. In the absence of these statistics, the optimizer may choose a less effective execution plan. Therefore, dynamic sampling can improve performance in a very volatile database where objects may be created or truncated frequently. However, dynamic sampling does incur an overhead, so it is recommended that you gather statistics manually on new and altered objects.

Using the `OPTIMIZER_DYNAMIC_SAMPLING` parameter enables dynamic sampling. The values accepted by this parameter are:

- 0: This disables dynamic sampling.
- 1: This value allows dynamic sampling to be performed when the optimizer determines that a full table scan is required due to nonexistent statistics.
- > 1: Values that are greater than 1 increase the times when dynamic sampling is an option. The maximum value allowed is 10.

# 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;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Locking Statistics

With Oracle Database 10g, you can lock statistics on a specified table with the new `LOCK_TABLE_STATS` procedure of the `DBMS_STATS` package to prevent automatic statistics gathering on that table. If a table's data is so volatile that the statistics become stale between statistics gathering intervals, you may choose to lock it when the table is empty. The optimizer then performs dynamic sampling on the table whenever required to get current statistics. You can also lock statistics on a volatile table at a point when it is fully populated so that the table statistics are more representative of the table population.

You can lock statistics at the schema level using the `LOCK_SCHEMA_STATS` procedure.

You can query the `STATTYPE_LOCKED` column in the `{USER | ALL | DBA}_TAB_STATISTICS` view to determine if the statistics on the table are locked.

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

**Note:** When you lock the statistics on a table, all of the dependent statistics are considered locked. This includes table statistics, column statistics, histograms, and dependent index statistics. Dynamic sampling involves an overhead and is repeated for the same objects unless statistics are gathered.

# Verifying Table Statistics

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

| ANALYZED  | SAMPLE_SIZE | MON | TABLE_NAME |
|-----------|-------------|-----|------------|
| 09-FEB-04 | 2000        | YES | EMPLOYEES  |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Verifying Table Statistics

To verify that the table statistics are available, query the USER\_TABLES data dictionary view using a statement like the one in the slide. Check for NULL values of the LAST\_ANALYZED column and also check for any updates after the date and time value indicated in this column. You can format this column to see the time information.

```
SQL> SELECT TO_TIMESTAMP(last_analyzed) analyzed  
2      FROM user_tables  
3      WHERE table_name = 'SALES';
```

ANALYZED

-----

09-FEB-04 12.00.00 AM

If performance on queries on specific tables or columns is poor, you should check that the optimizer has used the right sample size by querying the USER\_TABLES data dictionary view. Then, based on your knowledge of the data, you can choose to gather statistics manually. You can also decide whether a specific column requires a histogram based on the skew of the data.

# Verifying Column Statistics

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

| COLUMN_NAME      | NUM_DISTINCT | HISTOGRAM | NUM_BUCKETS | DENSITY    | ANALYZED  |
|------------------|--------------|-----------|-------------|------------|-----------|
| AMOUNT_SOLD      | 3586         | NONE      | 1           | .000278862 | 09-FEB-04 |
| CHANNEL_ID       | 4            | NONE      | 1           | .25        | 09-FEB-04 |
| CUST_ID          | 7059         | NONE      | 1           | .000141663 | 09-FEB-04 |
| PROD_ID          | 72           | FREQUENCY | 72          | 5.4416E-07 | 09-FEB-04 |
| PROMO_ID         | 4            | NONE      | 1           | .25        | 09-FEB-04 |
| QUANTITY_SOLD    | 1            | NONE      | 1           | 1          | 09-FEB-04 |
| TIME_ID          | 1460         | NONE      | 1           | .000684932 | 09-FEB-04 |
| 7 rows selected. |              |           |             |            |           |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Verifying Column Statistics

To verify that column statistics are available, query the USER\_TAB\_COL\_STATISTICS data dictionary view with a statement like the one in the slide.

Verifying column statistics is important in the following situations:

- Column statistics are useful to help determine the most efficient join method, which, in turn, is also based on the number of rows returned.
- The WHERE clause includes a column or columns with a bind variable (for example, column x = :variable\_y).
- Verify if data is skewed.

In these cases, you can use the stored column statistics to get a representative cardinality estimate for the given expression.

**NUM\_DISTINCT column statistic:** NUM\_DISTINCT indicates the number of distinct values for a column.

**HISTOGRAM column statistic:** The type of histogram for a column is stored in the HISTOGRAM column.

**DENSITY column statistic:** This statistic indicates the density of the values of the column, calculated as 1 divided by NUM\_DISTINCT. DENSITY is only relevant with histograms and is mainly used to estimate selectivity predicates for nonpopular values.



# Verifying Index Statistics

```
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;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Verifying Index Statistics

To verify that index statistics are available, query the USER\_INDEXES data dictionary view with a statement like the example in the slide. The output is as follows:

| NAME              | N_R   | L_A       | D_K  | L_B | A_L | J_I |
|-------------------|-------|-----------|------|-----|-----|-----|
| SALES_CHANNEL_BIX | 92    | 09-FEB-04 | 4    | 47  | 11  | NO  |
| SALES_CUST_BIX    | 35808 | 09-FEB-04 | 7059 | 475 | 1   | NO  |
| SALES_PROD_BIX    | 1074  | 09-FEB-04 | 72   | 32  | 1   | NO  |
| SALES_PROMO_BIX   | 54    | 09-FEB-04 | 4    | 30  | 7   | NO  |
| SALES_TIME_BIX    | 1460  | 09-FEB-04 | 1460 | 59  | 1   | NO  |

## Optimizer Index-Determination Criteria

The optimizer uses the following criteria when determining which index to use:

- Number of rows in the index (cardinality)
- Number of distinct keys (defining the selectivity of the index)
- Level or height of the index (indicating how deeply the data probe must search to find the data)

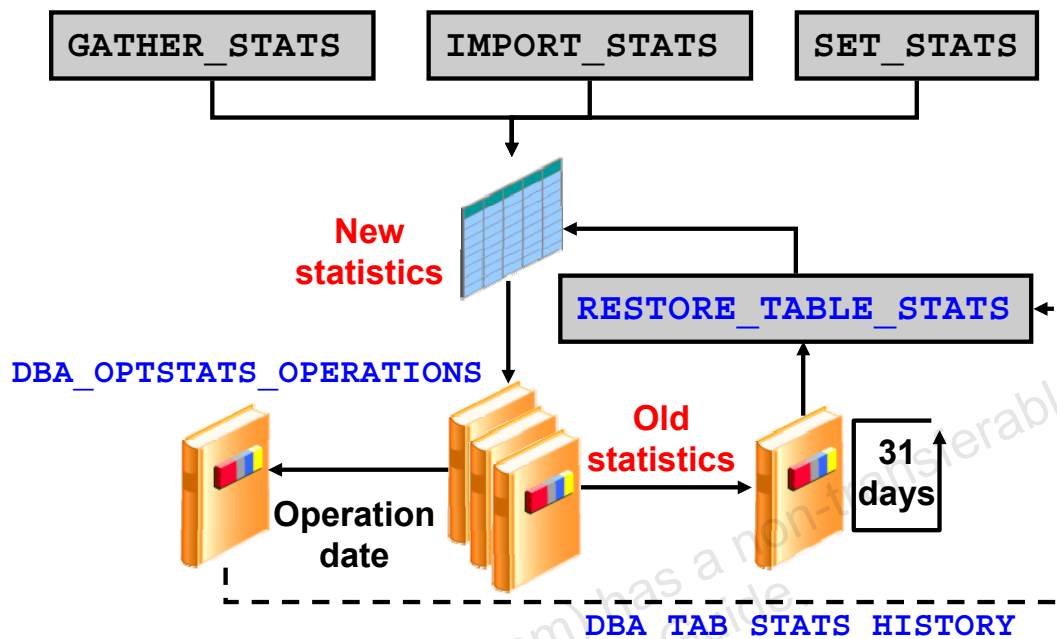
## Verifying Index Statistics

### Optimizer Index-Determination Criteria (continued)

- Number of leaf blocks in the index (number of I/Os needed to find the desired rows of data)
- Average leaf blocks for each key (ALFBKEY). This is the average number of leaf blocks in which each distinct value in the index appears, rounded to the nearest integer. For indexes that enforce UNIQUE and PRIMARY KEY constraints, this value is always 1.
- Average number of data blocks in the table that are pointed to by a distinct value in the index, rounded to the nearest integer. This statistic is the average number of data blocks that contain rows that contain a given value for the indexed columns.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# History of Optimizer Statistics



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## History of Optimizer Statistics

Whenever optimizer statistics are modified using the DBMS\_STATS package, old versions of the statistics are saved automatically for future restoration. Statistics can be restored using the RESTORE procedures of the DBMS\_STATS package. These procedures use a time stamp as an argument and restore statistics as of that time stamp. This is useful if newly collected statistics lead to some suboptimal execution plans and you want to revert to the previous set of statistics.

You can use the DBA\_OPTSTAT\_OPERATIONS view to determine the start and end times of all DBMS\_STATS operations executed at the schema and database level. By default, the DBA\_TAB\_STATS\_HISTORY view contains the history of table statistics modifications for the past 31 days. This means that you are able to restore the optimizer statistics to any time in the last 31 days. The old statistics are purged automatically at regular intervals based on the statistics history retention setting.

**Note:** Old versions of statistics are not stored when the ANALYZE command has been used for collecting statistics. You cannot use the RESTORE procedures to restore user-defined statistics.

# Managing Historical Optimizer Statistics

- `RESTORE_*_STATS()`
- `PURGE_STATS()`
- `ALTER_STATS_HISTORY_RETENTION()`

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Managing Historical Optimizer Statistics

- `RESTORE_TABLE_STATS` restores statistics of a table as of a specified time stamp. It also restores statistics of associated indexes and columns. If the table statistics were locked at the specified time stamp, the procedure locks the statistics.
- `RESTORE_SCHEMA_STATS` restores statistics of all tables of a schema as of a specified time stamp.
- `RESTORE_DATABASE_STATS` restores statistics of all tables in the database as of a specified time stamp.
- `RESTORE_FIXED_OBJECTS_STATS` restores statistics of all fixed tables as of a specified time stamp. You must have the `SYSDBA` or `ANALYZE ANY DICTIONARY` system privilege to execute this procedure.
- `RESTORE_DICTIONARY_STATS` restores statistics of all dictionary tables. You must have either the `SYSDBA` privilege or both the `ANALYZE ANY DICTIONARY` and `ANALYZE ANY` privileges to execute this procedure.
- `RESTORE_SYSTEM_STATS` restores system statistics as of a specified time stamp.

## Managing Historical Optimizer Statistics (continued)

- Retention is configurable using the ALTER\_STATS\_HISTORY\_RETENTION procedure. GET\_STATS\_HISTORY\_RETENTION gives you the current statistics history retention value. GET\_STATS\_HISTORY\_AVAILABILITY gives you the oldest time stamp for which statistics history is available.
- If automatic purging is disabled, you can manually purge the old versions of statistics using the PURGE\_STATS procedure.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# Generating System Statistics

- I/O
- CPU

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Generating System Statistics

System statistics enable the optimizer to consider a system's I/O and CPU performance and utilization. For each plan candidate, the optimizer computes estimates for I/O and CPU costs. It is important to know the system characteristics to pick the most efficient plan with optimal proportioning between I/O and CPU cost.

System I/O characteristics depend on many factors and do not always stay constant. Using system statistics management routines, database administrators can capture statistics in the interval of time when the system has the most common workload. For example, database applications can process OLTP transactions during the day and run OLAP reports at night. Administrators can gather statistics for both states and activate appropriate OLTP or OLAP statistics when needed. This enables the optimizer to generate relevant costs with respect to available system resource plans.

When the Oracle Database generates system statistics, it analyzes system activity in a specified period of time. Unlike table, index, or column statistics, the Oracle Database does not invalidate already parsed SQL statements when system statistics get updated. All new SQL statements are parsed using new statistics. It is highly recommended that you gather system statistics.

## Generating System Statistics (continued)

The `DBMS_STATS.GATHER_SYSTEM_STATS` routine collects system statistics in a user-defined time frame. You can also set system statistics values explicitly using `DBMS_STATS.SET_SYSTEM_STATS`. Use `DBMS_STATS.GET_SYSTEM_STATS` to verify system statistics

The preceding slide shows an example of gathering statistics by day for an OLTP application. Gathering ends after 720 minutes, and the statistics are stored in the `MYSTATS` table.

In the following example, you are gathering statistics during the night. Gathering ends after 720 minutes, and the statistics are stored in the `MYSTATS` table:

```
begin
dbms_stats.gather_system_stats(
    gathering_mode => 'interval',
    interval => 720,
   stattab => 'mystats',
    statid => 'olap');

end;
/
```

You can also switch between the two sets of statistics gathered. It is possible to automate this process by submitting a job to update the dictionary with appropriate statistics.

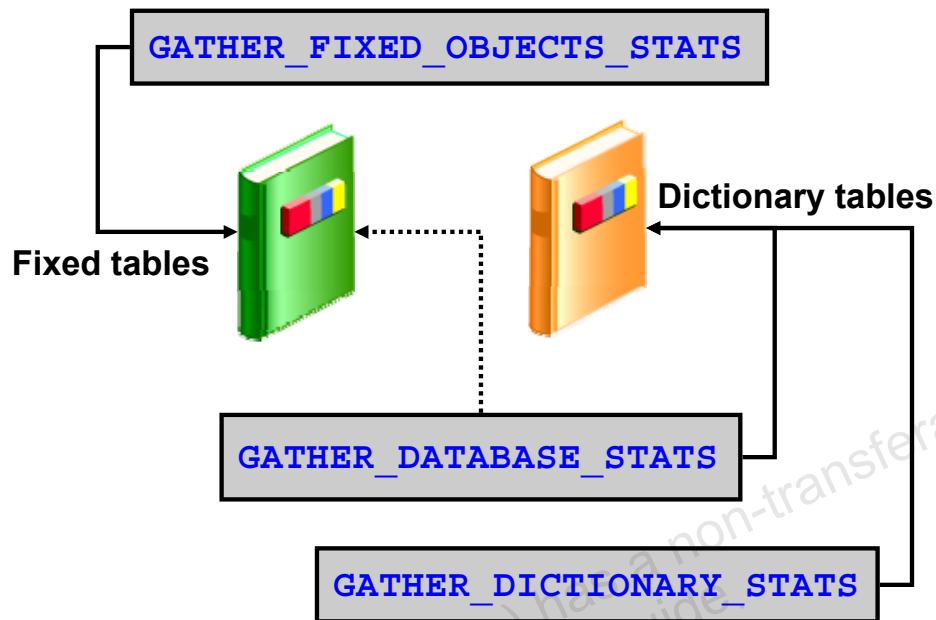
During the day, the following jobs import the OLTP statistics for the daytime run:

```
VARIABLE jobno number;
begin
dbms_job.submit(:jobno,
'dbms_stats.import_system_stats(''mystats'', ''oltp'');'
sysdate, 'sysdate + 1');
commit;
end;
/
```

During the night, the following jobs import the OLAP statistics for the night-time run:

```
begin
dbms_job.submit(:jobno,
'dbms_stats.import_system_stats(''mystats'', ''olap'');'
sysdate + 0.5, 'sysdate + 1');
commit; \
end;
/
```

# Statistics on Dictionary Objects



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Statistics on Dictionary Objects

With Oracle Database 10g, you should gather statistics on dictionary tables to get the best performance results.

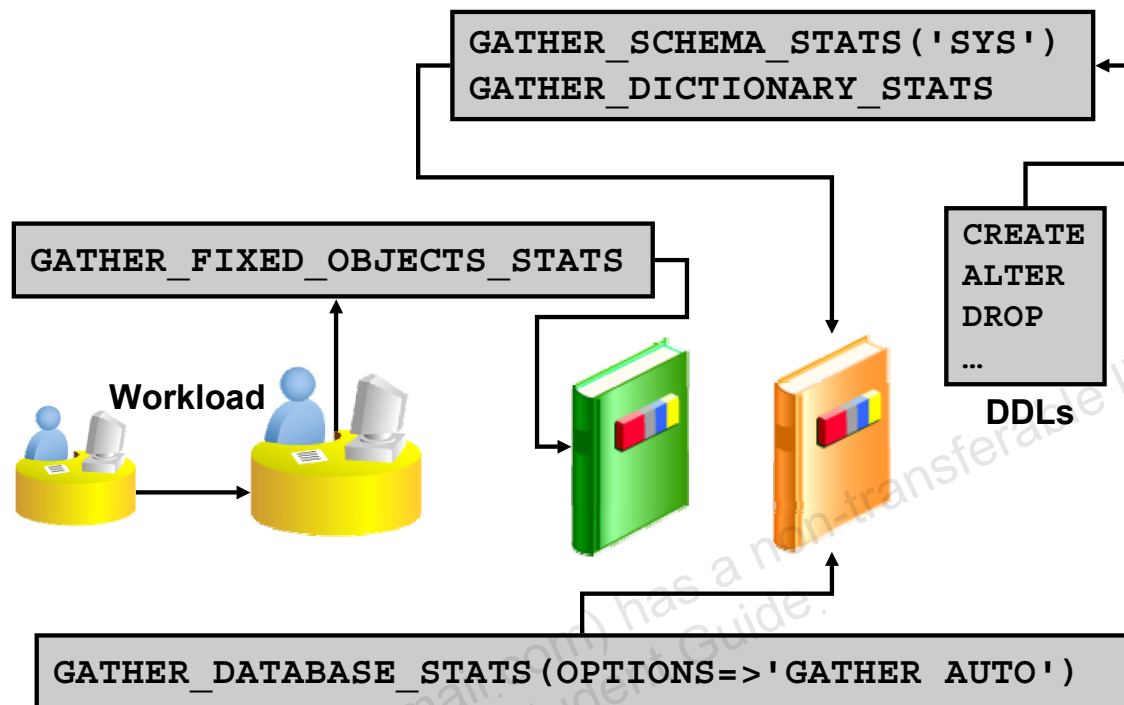
At any time, it is possible to collect statistics on dictionary tables by using the `DBMS_STATS.GATHER_{SCHEMA|DATABASE}_STATS` procedures with the `GATHER_SYS` argument set to `TRUE`. It is also possible to use the new procedure called `DBMS_STATS.GATHER_DICTIONARY_STATS`. You need the new `ANALYZE ANY DICTIONARY` system privilege for this purpose. This privilege is required to be able to analyze the dictionary objects and fixed objects, unless you are a user with the `SYSDBA` privilege.

The `GATHER_DATABASE_STATS` procedure has a new argument called `GATHER_FIXED`, which is set to `FALSE` by default and causes statistics not to be gathered for fixed tables (V\$). It should be sufficient to analyze fixed table statistics once during a typical system workload.

It is also possible to gather statistics on fixed tables by using the new `GATHER_FIXED_OBJECTS_STATS` procedure. Similarly, it is possible to delete statistics on all fixed tables and export or import statistics on fixed tables.



# Dictionary Statistics: Best Practices



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Dictionary Statistics: Best Practices

Here are some guidelines for collecting statistics on the data dictionary.

- **Dictionary objects:** Whatever the interval you use to analyze the objects in your schemas, the recommended method is to use either `GATHER_DATABASE_STATS` or `GATHER_SCHEMA_STATS`, with `OPTIONS` set to `GATHER AUTO`, assuming that the monitoring feature is enabled. In this way, only the objects that need to be reanalyzed are processed every time. You can also use the new `GATHER_DICTIONARY_STATS` procedure. Additionally, you may want to analyze the dictionary after a sufficient number of DDL operations have occurred.
- **Fixed objects:** It should be sufficient to analyze fixed objects once during a typical system workload. Subsequent collection is not necessary unless workload characteristics change dramatically. If you are connected as user `SYS`, for example, you can gather statistics of all fixed objects with the following command:  
`EXEC dbms_stats.gather_fixed_objects_stats('ALL');`

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

This lesson introduced you to using statistics with the Oracle query optimizer. In Oracle Database 10g, the task of statistics gathering has been fully automated using GATHER\_STATS\_JOB of the Scheduler. The predefined maintenance windows and automatic monitoring features ensure that all statistics are kept current.

Using the DBMS\_STATS package, you can also collect statistics manually on your objects.

Selectivity (the expected percentage of rows returned) depends on the type of operation performed in the WHERE clause. The optimizer is more likely to choose an index access path for a query with good selectivity. Selectivity is influenced by data distribution. When bind variables are used in a SQL statement, the optimizer must use a built-in default selectivity. These built-in values cannot be influenced.

## 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

swati bhatia (swati.june@gmail.com) has a non-transferable license to  
use this Student Guide.

# 8

## Application Tracing

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Objectives

You can use various analysis tools to evaluate the performance of SQL statements. After completing this lesson, you should be able to:

- Configure the SQL Trace facility to gather statistics
- Set up appropriate initialization parameters
- Format the trace statistics using the TKPROF utility
- Interpret the output of the TKPROF command

## Background Reading

For more information about SQL Trace and TKPROF, see the following documentation:

- *Oracle Database 10g Performance Guide and Reference*
- *Oracle Database 10g Performance Methods*
- *Oracle Database 10g Application Developer's Guide*
- *Oracle Database 10g Reference*

# Overview of Application Tracing

- **End to End Application Tracing**
  - Enterprise Manager
  - DBMS\_MONITOR
- **trcsess utility**
- **SQL Trace and TKPROF**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Overview of Application Tracing

Oracle Database 10g provides several tracing tools that can help you monitor and analyze applications running against an Oracle database.

- End to End Application Tracing can identify the source of an excessive workload, (such as a high-load SQL statement) by client identifier, service, module, or action. This isolates the problem to a specific user, service, or application component. This can be done through two interfaces:
  - Enterprise Manager
  - DBMS\_MONITOR package
- The Oracle-provided `trcsess` command-line utility consolidates tracing information based on specific criteria.
- The SQL Trace facility and TKPROF are two basic performance diagnostic tools that can help you monitor applications running against the Oracle Server.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## End to End Application Tracing

End to End Application Tracing simplifies the process of diagnosing performance problems in multitier environments. In multitier environments, a request from an end client is routed to different database sessions by the middle tier, making it difficult to track a client across different database sessions. End to End Application Tracing uses a client identifier to uniquely trace a specific end client through all tiers to the database server.

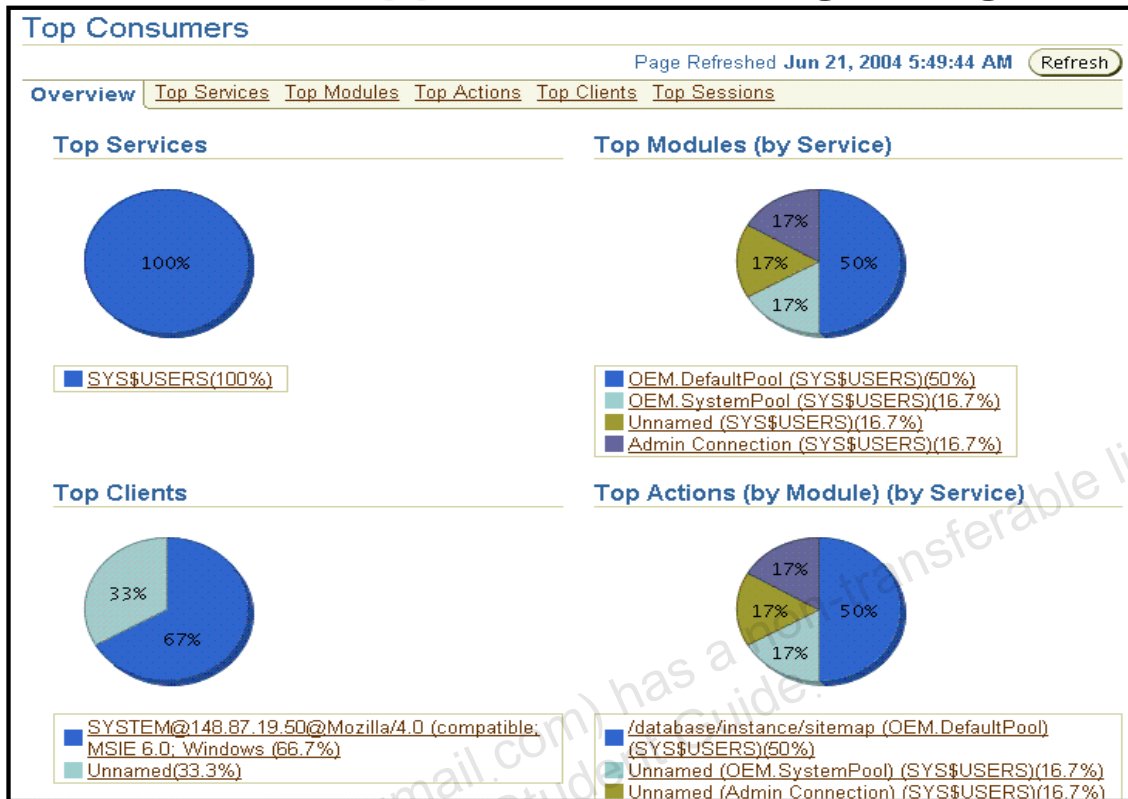
You can use End to End Application Tracing to identify the source of an excessive workload, such as a high-load SQL statement. Also, you can identify what a user's session is doing at the database level to resolve user performance problems.

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

- **Client identifier:** Specifies an end user based on the log-on ID, such as HR . HR
- **Service:** Specifies a group of applications with common attributes, service-level thresholds, and priorities, or specifies a single application, such as ACCTG (for an accounting application)
- **Module:** Specifies a functional block, such as Accounts Receivable or General Ledger, in an application
- **Action:** Specifies an action (such as an INSERT or UPDATE operation) in a module



# End to End Application Tracing Using EM



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## End to End Application Tracing Using Enterprise Manager

The primary interface for End to End Application Tracing is Oracle Enterprise Manager Database Control. To manage End to End Application Tracing through Oracle Enterprise Manager Database Control:

- On the **Performance** page, select the **Top Consumers** link under **Additional Monitoring Links**.
- Click the **Top Services**, **Top Modules**, **Top Actions**, **Top Clients**, or **Top Sessions** links to display the top consumers.
- On the individual **Top Consumers** pages, you can enable and disable statistics gathering and tracing for specific consumers.

After tracing information is written to files, the information can be consolidated by the `trcsess` utility and diagnosed with an analysis utility such as `TKPROF`.

End to End Application Tracing can also be managed with `DBMS_MONITOR` package APIs.

## Using DBMS\_MONITOR

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

1

```
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');
```

2

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Using DBMS\_MONITOR

### Enabling and Disabling for End to End Tracing

To enable tracing for client identifier, service, module, or action, you need to execute the appropriate procedures in the DBMS\_MONITOR package. You can enable tracing for specific diagnosis and workload management by the following criteria:

Client identifier for specific clients, combination of service name, module, and action name and session. With the criteria that you provide, specific trace information is captured in a set of trace files and combined into a single output trace file.

### Statistic Gathering for Client Identifier

1. The CLIENT\_ID\_STAT\_ENABLE procedure enables statistics gathering for a given client identifier. In the first example, OE.OE is the client identifier for which SQL tracing is to be enabled. The TRUE argument specifies that wait information will be present in the trace. The FALSE argument specifies that bind information will not be present in the trace.

You can view client identifiers in the CLIENT\_IDENTIFIER column in V\$SESSION. The CLIENT\_ID\_STAT\_DISABLE procedure disables statistics gathering for a given client identifier, as in the following example:

```
EXECUTE DBMS_MONITOR.CLIENT_ID_STAT_DISABLE(  
    client_id => 'OE.OE');
```

## Using DBMS\_MONITOR (continued)

### Statistic Gathering for Service, Module, and Action

2. The `SERV_MOD_ACT_STAT_ENABLE` procedure enables statistics gathering for a combination of service, module, and action. If both of the commands shown in the second example are executed, statistics are gathered as follows:

- For the ACCTG service, because accumulation for each service name is the default
- For all actions in the PAYROLL module
- For the INSERT ITEM action within the GLEDGER module

The `SERV_MOD_ACT_STAT_DISABLE` procedure disables statistics gathering for a combination of service, module, and action, as in the following example:

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_STAT_DISABLE(  
    service_name => 'ACCTG',  
    module_name => 'GLEDGER',  
    action_name => 'INSERT ITEM');
```

**To enable tracing for all actions for a given combination of service and module in an instance:**

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE(  
    service_name => 'ACCTG', module_name => 'PAYROLL',  
    waits => TRUE, binds => FALSE, instance_name => 'inst1');
```

The `SERV_MOD_ACT_TRACE_DISABLE` procedure disables the trace at all enabled instances for a given combination of service name, module, and action name globally.

The `SESSION_TRACE_ENABLE` procedure enables the trace for a given database session identifier (SID) on the local instance.

To enable tracing for a specific session ID and serial number, determine the values for the session that you want to trace:

```
SELECT SID, SERIAL#, USERNAME FROM V$SESSION; SID SERIAL#  
USERNAME;
```

```
  SID          SERIAL#  USERNAME  
-----  
    238          28841  SH
```

...

Use the appropriate values to enable tracing for a specific session:

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

The `TRUE` argument specifies that wait information will be present in the trace. The `FALSE` argument specifies that bind information will not be present in the trace.

The `SESSION_TRACE_DISABLE` procedure disables the trace for a given database session identifier (SID) and serial number:

```
EXECUTE DBMS_MONITOR.SESSION_TRACE_DISABLE(  
    session_id => 27, serial_num => 60);
```

## 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.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

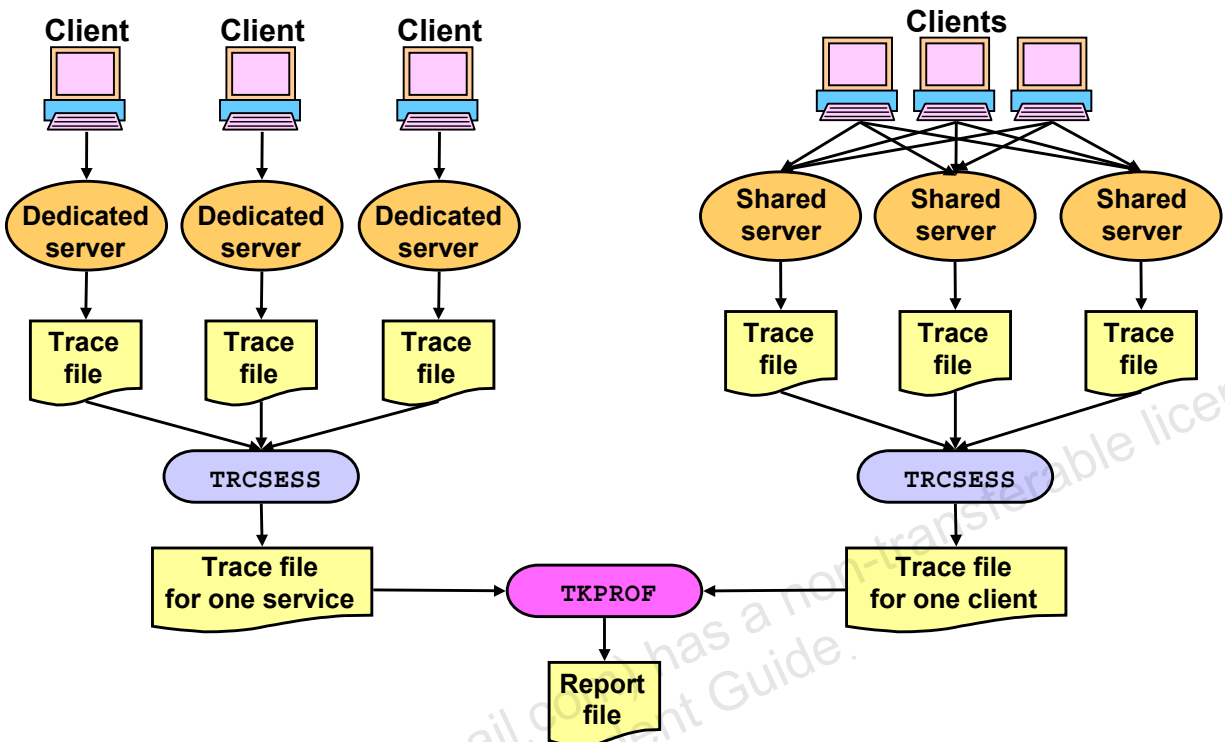
### Viewing Gathered Statistics for End to End Application Tracing

The statistics that have been gathered can be displayed with a number of dynamic views. The accumulated global statistics for the currently enabled statistics can be displayed with the `DBA_ENABLED_AGGREGATIONS` view. The accumulated statistics for a specified client identifier can be displayed in the `V$CLIENT_STATS` view. The accumulated statistics for a specified service can be displayed in the `V$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.

### Viewing Enabled Traces for End to End Tracing

All outstanding traces can be displayed in an Oracle Enterprise Manager report or with the `DBA_ENABLED_TRACES` view. In the `DBA_ENABLED_TRACES` view, you can determine detailed information about how a trace was enabled, including the trace type. The trace type specifies whether the trace is enabled for client identifier, session, service, or a combination of service, module, and action.

## trcsess Utility



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## trcsess Utility

**trcsess** is useful for consolidating the tracing of a particular session for performance or debugging purposes. Tracing a specific session is usually not a problem in the dedicated server model because a single dedicated process serves a session during its lifetime. All the trace information for the session can be seen from the trace file belonging to the dedicated server serving it. However, in a shared server configuration, a user session is serviced by different processes from time to time. The trace pertaining to the user session is scattered across different trace files belonging to different processes. This makes it difficult to get a complete picture of the life cycle of a session. **trcsess** can be used to process all the files generated by End to End Tracing:

```
trcsess [output=output_file_name]
        [session=session_id]
        [clientid=client_id]
        [service=service_name]
        [action=action_name]
        [module=module_name]
        [trace_files]
```

## trcsess Utility

```
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
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### trcsess Utility (continued)

The `trcsess` utility consolidates trace output from selected trace files based on several criteria:

- `output` specifies the file where the output is generated. If this option is not specified, then standard output is used for the output.
- `session` consolidates the trace information for the session specified. The session identifier is a combination of session index and session serial number, such as 21.2371. You can locate these values in the `V$SESSION` view.
- `clientid` consolidates the trace information for the given client ID.
- `service` consolidates the trace information for the given service name.
- `action` consolidates the trace information for the given action name.
- `module` consolidates the trace information for the given module name.
- `trace_files` is a list of all the trace file names, separated by spaces, in which `trcsess` should look for trace information. The wild card character `*` can be used to specify the trace file names. If trace files are not specified, all the files in the current directory are taken as input to `trcsess`.

After `trcsess` merges the trace information into a single output file, the output file can be processed by `TKPROF`.

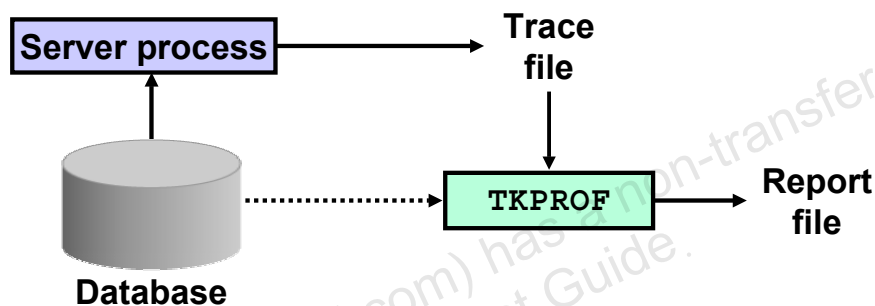
The example shows generating trace for the HR user session.

## Sample Output of trcsess Command

```
*** [ Unix process pid: 11155 ]
*** 2004-06-21 06:23:52.002
=====
PARSING IN CURSOR #2 len=34 dep=0 uid=58 oct=42 lid=58
tim=1062328351564737 hv=2069488880 ad='57dcdcb0'
alter session set sql_trace = true
END OF STMT
EXEC
#2:c=0,e=18102,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=106232
8351564695
*** 2004-06-21 06:24:13.233
=====
PARSING IN CURSOR #1 len=23 dep=0 uid=58 oct=3 lid=58
tim=1062328372298810 hv=4069246757 ad='55c754ec'
select * from employees
END OF STMT
PARSE
#1:c=30000,e=27470,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=10
62328372298794
EXEC
#1:c=0,e=135,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=10623283
72299158
FETCH
#1:c=0,e=275,p=0,cr=3,cu=0,mis=0,r=1,dep=0,og=1,tim=10623283
72299583
FETCH
#1:c=0,e=154,p=0,cr=1,cu=0,mis=0,r=15,dep=0,og=1,tim=1062328
372301134
...
=====
PARSING IN CURSOR #2 len=35 dep=0 uid=58 oct=42 lid=58
tim=1062328398305099 hv=310044142 ad='572b96ac'
alter session set sql_trace = false
END OF STMT
PARSE
#2:c=0,e=997,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=10623283
98305081
EXEC
#2:c=0,e=502,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=10623283
98305797
```

# SQL Trace Facility

- Is usually enabled at the session level
- Gathers session statistics for SQL statements grouped by session
- Produces output that can be formatted by TKPROF



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Trace Facility

The SQL Trace facility and TKPROF let you accurately assess the efficiency of the SQL statements an application runs. A good way to compare two execution plans is to execute the statements and compare the statistics to see which one performs better. SQL Trace writes its session statistics output to a file, and you use TKPROF to format it. You can use these tools along with EXPLAIN PLAN to get the best results.

SQL Trace facility:

- Can be enabled for a session or for an instance
- Reports on volume and time statistics for the parse, execute, and fetch phases
- Produces output that can be formatted by TKPROF

When the SQL Trace facility is enabled for a session, the Oracle Database generates a trace file containing session statistics for traced SQL statements for that session. When the SQL Trace facility is enabled for an instance, the Oracle Database creates trace files for all sessions.

**Note:** SQL Trace involves some overhead, so you usually do not want to enable SQL Trace at the instance level.



# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Information Provided by SQL Trace

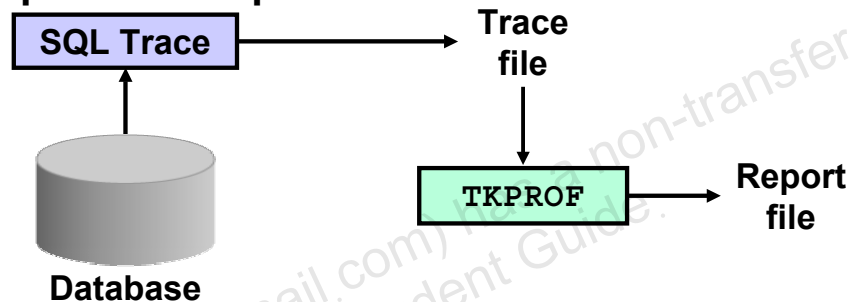
The SQL Trace facility provides performance information about individual SQL statements. In addition to the information listed in the slide, SQL Trace also provides row source information including:

- Row operations showing the actual execution plan of each SQL statement
- Number of rows, number of consistent reads, number of physical reads, number of physical writes, and time elapsed for each operation on a row

**Note:** A summary for each trace file can be obtained using the TKPROF utility

# 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.



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## How to Use the SQL Trace Facility

You must complete these steps to use SQL Trace:

1. Set the appropriate initialization parameters.
2. Enable SQL Trace.
3. Run the application (and disable tracing when done).
4. Disable Trace.
5. Close the session.
6. Format the trace file produced by SQL Trace with TKPROF.
7. Interpret the output and tune the SQL statements when needed.

Running SQL Trace increases system overhead. Use SQL Trace only when required, and use it at the session level rather than at the instance level.

# Initialization Parameters

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Initialization Parameters

Several initialization parameters relate to SQL Trace.

### **TIMED\_STATISTICS**

The SQL Trace facility provides a variety of information about SQL execution within a process, optionally including timing information. If you want timing information, you must set this parameter to TRUE. You can do so for the entire database by setting the following initialization parameter in the parameter file before starting up and opening the database:

```
TIMED_STATISTICS = TRUE
```

The parameter also can be set dynamically for a particular session with the following command:

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

The timing statistics have a resolution measured in microseconds.

**MAX\_DUMP\_FILE\_SIZE:** When the SQL Trace facility is enabled at the instance level, every call to the server produces a text line in a file in the operating system's file format. The maximum size of these files (in operating system blocks) is limited by this initialization parameter. This is a dynamic parameter as well as a session parameter.

## Initialization Parameters (continued)

**USER\_DUMP\_DEST** is the location of the background dump destination. This must fully specify the destination for the trace file according to the conventions of the operating system. The default value is the default destination for system dumps on the operating system. This value can be modified with the ALTER SYSTEM command. This is a dynamic parameter as well as a session parameter.

### **STATISTICS\_LEVEL**

The Oracle-provided STATISTICS\_LEVEL initialization parameter controls all major statistics collections or advisories in the database. This parameter sets the statistics collection level for the database.

Depending on the setting of STATISTICS\_LEVEL, certain advisories or statistics are collected:

- **BASIC:** No advisories or statistics are collected. Monitoring and many automatic features are disabled. Oracle does not recommend this setting because it disables important Oracle features.
- **TYPICAL:** This is the default value and ensures collection for all major statistics while providing best overall database performance. This setting should be adequate for most environments.
- **ALL:** All of the advisories or statistics that are collected with the TYPICAL setting are included, plus timed operating system statistics and row source execution statistics.

This view lists the status of the statistics or advisories controlled by STATISTICS\_LEVEL.

## Obtain Information About Parameter Settings

You can display current parameter values by querying the V\$PARAMETER view:

```
SQL> SELECT name, value
2      FROM v$parameter
3      WHERE name LIKE '%dump%';
```

Or you can use the following:

```
SQL> SHOW PARAMETER dump
```

# 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
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Enabling SQL Trace for a Session

You can use a SQL command to enable SQL Trace for your session:

```
SQL> ALTER SESSION SET sql_trace = true;
```

You can also use a supplied package to enable SQL Trace for your session. This is useful when you want to enable or disable SQL Trace from within a PL/SQL unit.

```
SQL> EXECUTE dbms_session.set_sql_trace(true);
```

You can also enable SQL Trace for another user's session with a supplied package.

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

In this procedure call, `session_id` and `serial_id` are the values in the `SID` and `SERIAL#` columns of `V$SESSION`, a data dictionary view commonly used by database administrators.

To enable SQL Trace for an entire instance use an `ALTER SYSTEM` command or set the initialization parameter `SQL_TRACE = TRUE`.

To ensure that wait events information is written to the trace file for the session, run the following SQL statement:

```
ALTER SESSION SET EVENTS '10046 trace name context forever,level 8';
```

## Disabling SQL Trace

When you have finished tuning, disable SQL Trace by using any of the preceding methods, substituting the word `FALSE` for `TRUE`. If you enable SQL Trace for a single session, exiting that session also disables SQL Trace.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Formatting Your Trace Files

Use the TKPROF command to format your trace files into readable output. This is the TKPROF syntax:

```
OS> tkprof tracefile outputfile [options]
tracefile          Name of the trace output file (input for TKPROF)
outputfile         Name of the file to store the formatted results
```

When the TKPROF command is entered without any arguments, it generates a usage message together with a description of all TKPROF options. See the next slide for a full listing. This is the output that you get when you enter the TKPROF command without any arguments:

```
Usage: tkprof tracefile outputfile [explain= ] [table= ]
        [print= ] [insert= ] [sys= ] [sort= ]
```

By default, the .trc file is named after the SPID. You can find the SPID in V\$PROCESS. An easier way of finding the file is the following:

```
SQL> ALTER SESSION SET TRACEFILE_IDENTIFIER = 'MY_FILE';
```

Then the trace file in TKPROF will include the 'MY\_FILE' string.

# TKPROF Command Options

```
SORT = option
PRINT = n
EXPLAIN = username/password
INSERT = filename
SYS = NO
AGGREGATE = NO
RECORD = filename
TABLE = schema.tablename
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## TKPROF Command Options

|           |  |
|-----------|--|
| INSERT    | Creates a SQL script to load the TKPROF results into a database table  |
| SORT      | Order in which to sort the statements in the report (see previous page for a list of values)   |
| PRINT     | Produces a report on this (sorted) number of statements only (This option is especially useful in combination with the SORT option.)   |
| EXPLAIN   | Logs on and executes EXPLAIN PLAN in the specified schema  |
| SYS       | Disables the listing of recursive SQL statements, issued by the user SYS   |
| AGGREGATE | Disables or enables the (default) behavior of TKPROF, aggregating identical SQL statements into one record   |
| WAITS     | Specifies whether to record summaries for any wait events found in the trace files   |
| TABLE     | Specifies the table to temporarily store execution plans before writing them to the output file (This parameter is ignored if EXPLAIN is not specified. It is useful when several individuals use TKPROF to tune the same schema concurrently, to avoid destructive interference.) |
| RECORD    | Creates a SQL script with all the nonrecursive SQL statements found in the trace file (This script can be used to replay the tuning session later.)  |



## Output of the TKPROF Command

Other options are:

|        |  |
|--------|--|
| prscnt | Number of times parse was called                     |
| prscpu | CPU time parsing                                     |
| prsela | Elapsed time parsing                                 |
| prsdsk | Number of disk reads during parse                    |
| prsqry | Number of buffers for consistent read during parse   |
| prscu  | Number of buffers for current read during parse      |
| prsmis | Number of misses in library cache during parse       |
| execnt | Number of executes that were called                  |
| execpu | CPU time spent executing                             |
| exeela | Elapsed time executing                               |
| exedsk | Number of disk reads during execute                  |
| exeqry | Number of buffers for consistent read during execute |
| execu  | Number of buffers for current read during execute    |
| exerow | Number of rows processed during execute              |
| exemis | Number of library cache misses during execute        |
| fchcnt | Number of times fetch was called                     |
| fchcpu | CPU time spent fetching                              |
| fchela | Elapsed time fetching                                |
| fchdsk | Number of disk reads during fetch                    |
| fchqry | Number of buffers for consistent read during fetch   |
| fchcu  | Number of buffers for current read during fetch      |
| fchrow | Number of rows fetched                               |
| userid | ID of user who parsed the cursor                     |

## Output of the TKPROF Command

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

|                |  |
|----------------|--|
| <b>PARSE</b>   | <b>Translates the SQL statement into an execution plan</b>   |
| <b>EXECUTE</b> | <b>Executes the statement<br/>(This step modifies the data for INSERT, UPDATE,<br/>and DELETE statements.)</b> |
| <b>FETCH</b>   | <b>Retrieves the rows returned by a query<br/>(Fetches are performed only for SELECT statements.)</b>          |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Output of the TKPROF Command

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

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

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

# Output of the TKPROF Command

There are seven categories of trace statistics:

|         |  |
|---------|--|
| Count   | Number of times the procedure was executed         |
| CPU     | Number of seconds to process                       |
| Elapsed | Total number of seconds to execute                 |
| Disk    | Number of physical blocks read                     |
| Query   | Number of logical buffers read for consistent read |
| Current | Number of logical buffers read in current mode     |
| Rows    | Number of rows processed by the fetch or execute   |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Output of the TKPROF Command

The output is explained on the following page.

Sample output is as follows:

| call    | count | cpu  | elapsed | disk | query | current | rows |
|---------|-------|------|---------|------|-------|---------|------|
| Parse   | 1     | 0.03 | 0.06    | 0    | 0     | 0       | 0    |
| Execute | 1     | 0.06 | 0.30    | 1    | 3     | 0       | 0    |
| Fetch   | 2     | 0.00 | 0.46    | 0    | 0     | 0       | 1    |
| total   | 4     | 0.09 | 0.83    | 1    | 3     | 0       | 1    |

## Output of the TKPROF Command (continued)

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

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

### Note:

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

# 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

Rows      Row Source Operation
-----
24  TABLE ACCESS BY INDEX ROWID EMPLOYEES (cr=9 pr=0 pw=0 time=129 us)
24  INDEX RANGE SCAN SAL_IDX (cr=3 pr=0 pw=0 time=1554 us)(object id ...
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Output of the TKPROF Command

### Recursive Calls

To execute a SQL statement issued by a user, the Oracle server must occasionally issue additional statements. Such statements are called *recursive SQL statements*. For example, if you insert a row in a table that does not have enough space to hold that row, the Oracle server makes recursive calls to allocate the space dynamically. Recursive calls are also generated when data dictionary information is not available in the data dictionary cache and must be retrieved from disk.

If recursive calls occur while the SQL Trace facility is enabled, TKPROF marks them clearly as recursive SQL statements in the output file. You can suppress the listing of recursive calls in the output file by setting the SYS=NO command-line parameter. Note that the statistics for recursive SQL statements are always included in the listing for the SQL statement that caused the recursive call.

### Library Cache Misses

TKPROF also lists the number of library cache misses resulting from parse and execute steps for each SQL statement. These statistics appear on separate lines following the tabular statistics.

## Output of the TKPROF Command (continued)

### Row Source Operations

These provide the number of rows processed for each operation executed on the rows and additional row source information, such as physical reads and writes; cr = consistent reads, w = physical writes, r = physical reads, time = time (in microseconds).

### Parsing User ID

This is the ID of the last user to parse the statement.

### Row Source Operation

The row source operation shows the data sources for execution of the SQL statement. This is included only if the cursor has been closed during tracing. If the row source operation does not appear in the trace file, you may then want to see the EXPLAIN PLAN.

### Execution Plan

If you specify the EXPLAIN parameter on the TKPROF command line, TKPROF uses the EXPLAIN PLAN command to generate the execution plan of each SQL statement traced. TKPROF also displays the number of rows processed by each step of the execution plan.

**Note:** Be aware that the execution plan is generated *at the time that the TKPROF command is run* and not at the time the trace file was produced. This could make a difference if, for example, an index has been created or dropped since tracing the statements.

### Optimizer Mode or Hint

This indicates the optimizer hint that is used during the execution of the statement. If there is no hint, then it shows the optimizer mode that is used.

## TKPROF Output with No Index: Example

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

| call    | count | cpu  | elapsed | disk | query | current | rows |
|---------|-------|------|---------|------|-------|---------|------|
| Parse   | 1     | 0.02 | 0.02    | 0    | 0     | 0       | 0    |
| Execute | 1     | 0.00 | 0.00    | 0    | 0     | 0       | 0    |
| Fetch   | 2     | 0.10 | 0.09    | 1408 | 1459  | 0       | 1    |
| total   | 4     | 0.12 | 0.11    | 1408 | 1459  | 0       | 1    |

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) |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### TKPROF Output with No Index: Example

The example in the slide shows that the aggregation of results across several executions (rows) is being fetched from the CUSTOMERS table. It requires .12 seconds of CPU fetch time. The statement is executed through a full table scan of the CUSTOMERS table, as you can see in the row source operation of the output.

The statement must be optimized.

**Note:** If CPU or elapsed values are 0, then `timed_statistics` is not set.

## TKPROF Output with Index: Example

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

| call    | count | cpu  | elapsed | disk | query | current |   |
|---------|-------|------|---------|------|-------|---------|---|
| rows    |       |      |         |      |       |         |   |
| Parse   | 1     | 0.01 | 0.00    | 0    | 0     | 0       | 0 |
| Execute | 1     | 0.00 | 0.00    | 0    | 0     | 0       | 0 |
| Fetch   | 2     | 0.00 | 0.00    | 0    | 77    | 0       | 1 |
| total   | 4     | 0.01 | 0.00    | 0    | 77    | 0       | 1 |

```
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)
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### TKPROF Output with Index: Example

The results shown in the slide indicate that CPU time was reduced to .01 second when an index was created on the CUST\_CITY column. These results may have been achieved because the statement uses the index to retrieve the data. Additionally, because this example is reexecuting the same statement, most of the data blocks are already in memory. You can achieve significant improvements in performance by indexing sensibly. Identify areas for potential improvement using the SQL Trace facility.

**Note:** Indexes should not be built unless required. Indexes do slow down the processing of INSERT, UPDATE, and DELETE commands because references to rows must be added, changed, or removed. Unused indexes should be removed. However, instead of processing all of the application SQL through EXPLAIN PLAN, you can use index monitoring to identify and remove any indexes that are not used.



# 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**

```
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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

This lesson introduced you to the SQL Trace facility. By using SQL Trace, you can evaluate the performance of SQL statements. After SQL Trace is enabled, you view execution plans and statistics on SQL statements that are generated in a trace file.

To use SQL Trace, follow these steps:

- Identify the appropriate initialization parameters that you want to use.
- Enable SQL Trace.
- Use TKPROF to format the trace file that is generated.
- Interpret the output of TKPROF.

## Practice 8: Overview

**This practice covers the following topics:**

- **Using TKPROF**
- **Using DBMS\_MONITOR**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 9

## Identifying High-Load SQL

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# Objectives

**After completing this lesson, you should understand the different methods to identify high-load SQL:**

- **ADDM**
- **Top SQL**
- **Dynamic performance views**
- **Statspack**

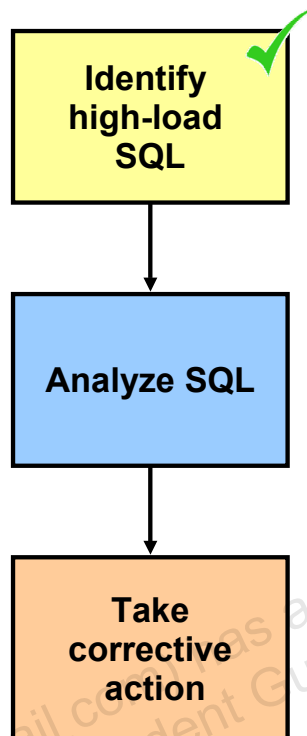
**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Objectives

Identifying high-load SQL is the first step in the SQL Tuning process. In this lesson, you learn the different tools that are available for identifying high-load SQL.

# SQL Tuning Process: Overview



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Tuning Process: Overview

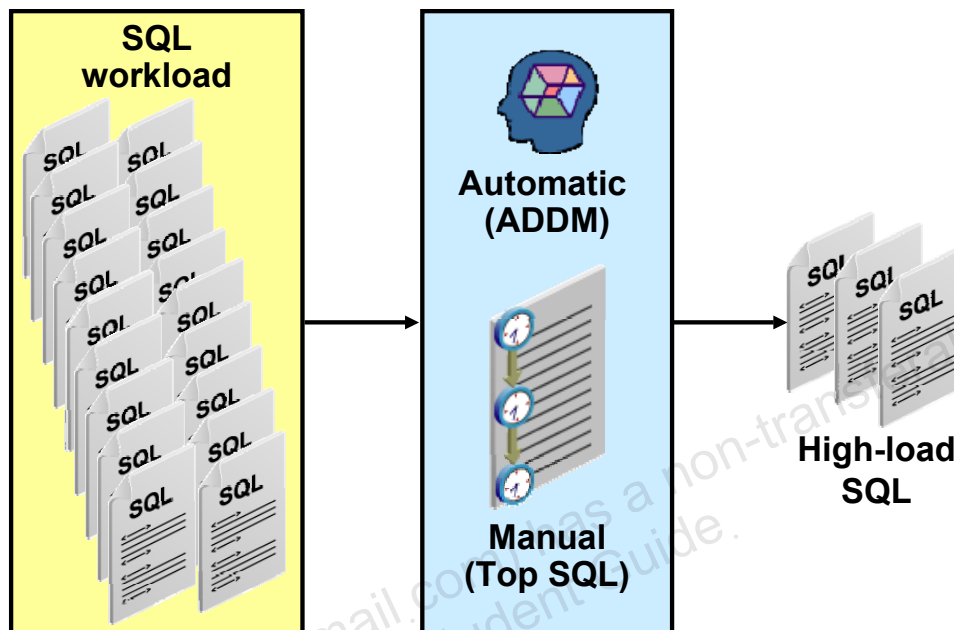
SQL tuning involves three basic steps:

- Identify high-load or top SQL statements that are responsible for a large share of the application workload and system resources, by looking at the past SQL execution history available in the system (for example, the ADDM report or the Top SQL page on EM, or the cursor cache statistics stored in the V\$SQL dynamic view).
- Analyze SQL to verify that the execution plans produced by the query optimizer for these statements perform reasonably well.
- Take corrective action to generate better execution plans for poorly performing SQL statements.

The three steps are repeated until the system performance reaches a satisfactory level or until no more statements can be tuned.

This lesson covers the first step by showing you how to identify high-load SQL.

# Identifying High-Load SQL



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Identifying High-Load SQL

This is the first step in the SQL tuning process. SQL statements that have a slow response time or are consuming high resources (such as CPU, I/O, and temporary space) are considered to be high-load SQL. The process of identifying such SQL statements and tuning them can be very challenging.

With Oracle Database 10g, the task of identifying high-load SQL statements has been automated by Automatic Database Diagnostic Monitor (ADDM). ADDM is the proactive performance diagnostic tool that automatically identifies bottlenecks within the Oracle Database, including slow-running or high-load SQL, and makes recommendations about the options available for fixing these bottlenecks. In Oracle Database 10g, all that is required on the part of the user is to look at the ADDM findings report to identify high-load SQL.

Other ways of identifying high-load SQL are also available in Oracle Database 10g. The Top SQL page in EM contains information about the most resource-consuming SQL in the system. It has information about two types of SQL:

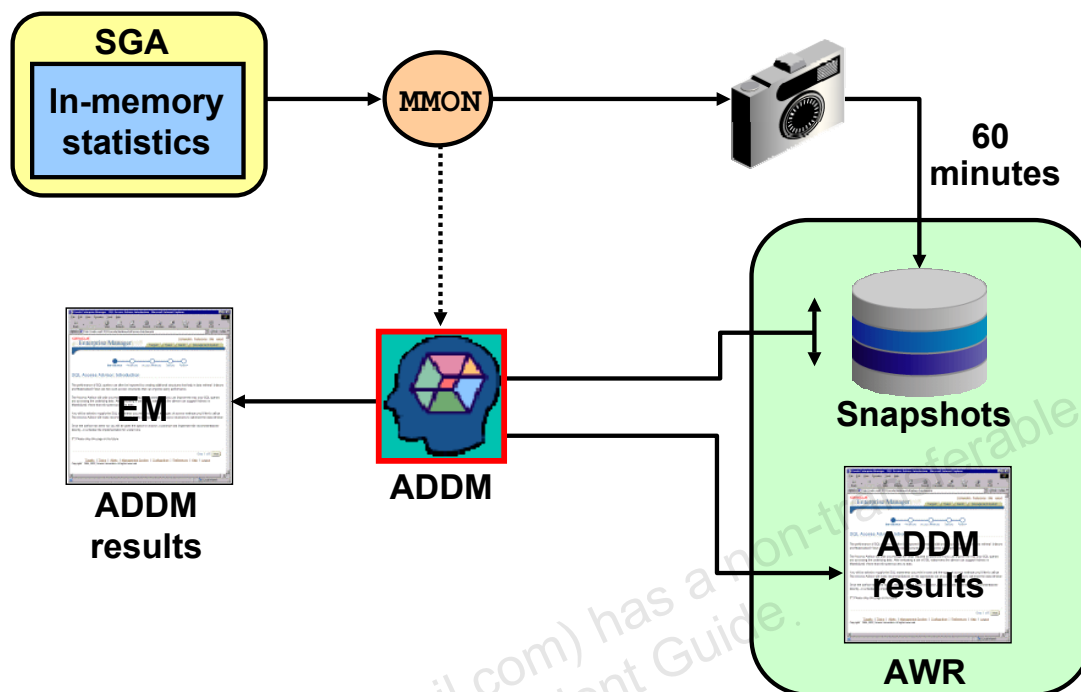
- Spot SQL: Current SQL from the cursor cache and active sessions
- Period SQL: Historical SQL stored in the Automatic Workload Repository (AWR) based on the retention period specified

## Identifying High-Load SQL (continued)

In addition, you can obtain information about high-load SQL by studying the dynamic performance views such as V\$SQL\_AREA and V\$ACTIVE\_SESSION\_HISTORY, and the workload repository views such as DBA\_HIST\_ACTIVE\_SESS\_HISTORY and DBA\_HIST\_SQL\_PLAN.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# Automatic Database Diagnostic Monitor



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Automatic Database Diagnostic Monitor

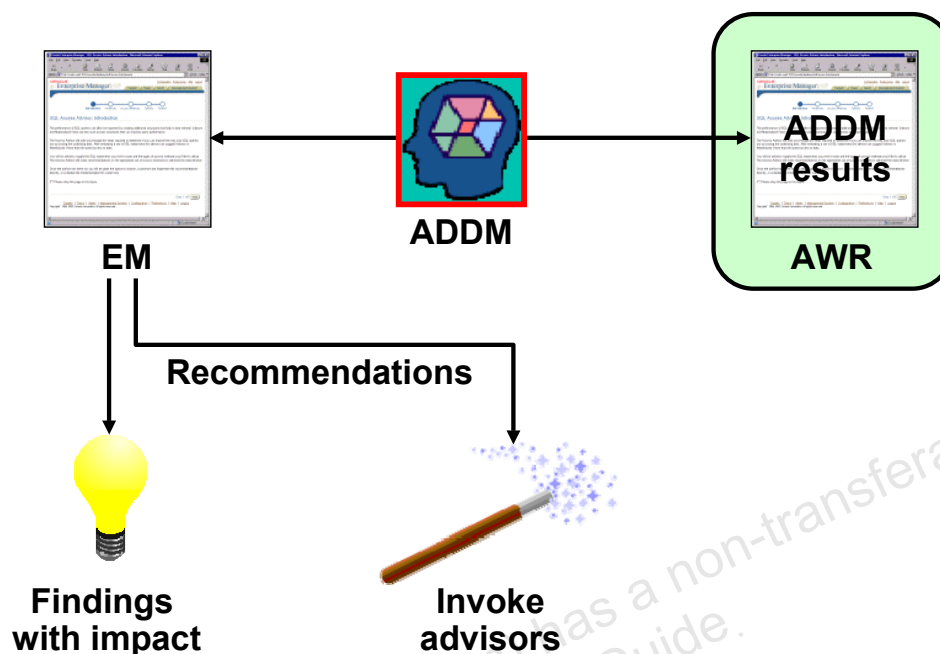
ADDM is a server-based performance expert in a box. It is automatically used for proactive and effective tuning of your database. ADDM runs automatically at predefined intervals and analyzes system performance. It identifies all kinds of potential performance problems including poorly performing SQL. Thus, the task of high-load SQL identification is performed automatically by ADDM and you simply need to look at the ADDM report to see its recommendations.

By default, the database automatically captures statistical information every 60 minutes from the SGA and stores it in the AWR in the form of snapshots. These snapshots are stored on disk. Additionally, ADDM is scheduled to run automatically by MMON, a new background process introduced in Oracle Database 10g to detect problems proactively. Each time a snapshot is taken, ADDM is triggered to do an analysis of the period corresponding to the last two snapshots. Such a capability proactively monitors the instance and detects all types of performance bottlenecks, such as poor memory configuration or poorly performing SQL statements, before they become a significant problem.

The results of each ADDM analysis are stored inside the AWR and are accessible as well through the EM console and command-line interface.



## ADDM Output



ORACLE

Copyright © 2007, Oracle. All rights reserved.

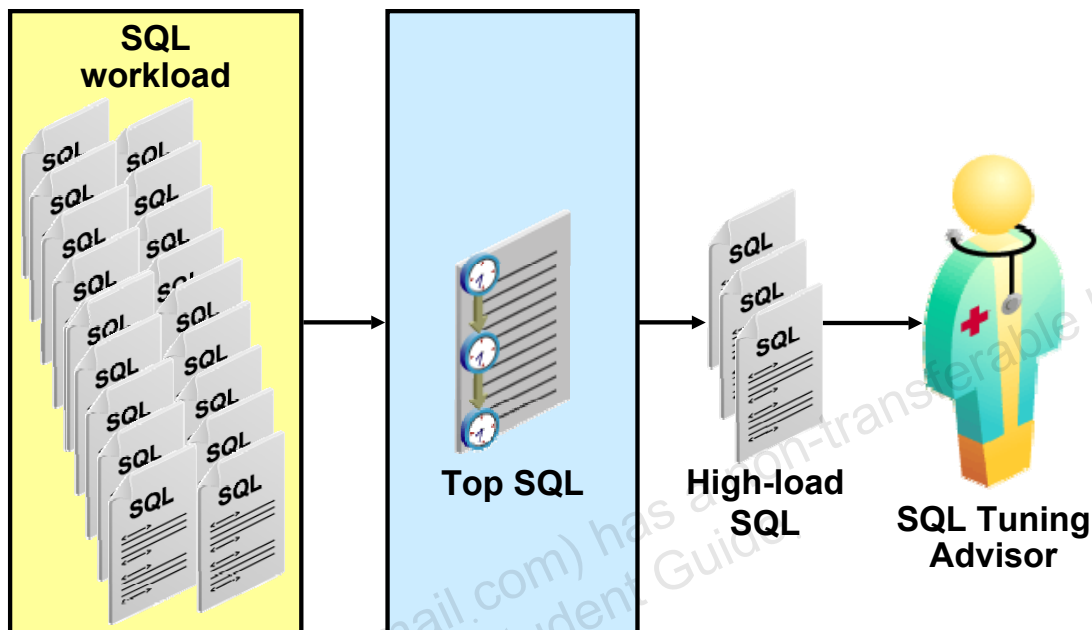
### ADDM Output

ADDM output is best viewed using the Enterprise Manager console. After each analysis, ADDM reports information about findings with corresponding impact on the database. The impact is measured by the percentage of database time spent: Had the problem not existed, the database time would have been reduced by that percentage. The ADDM findings include symptoms and root causes.

ADDM also produces a set of recommendations with associated benefits and rationales. The recommendations can also include information to further call server-based advisors for more detailed analysis of a particular problem.

ADDM output is also stored in the AWR for history purposes.

## Manual Identification: Top SQL



ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Manual Identification: Top SQL

Another source for identifying high-load SQL is the list of top SQL statements shown on the EM screen. These top SQL statements are captured by the AWR based on their cumulative execution statistics within the selected time window. The top SQL statements can be ranked on associated statistics such as CPU consumption, elapsed time, buffer gets, and so on. The user can select one or more top SQL statements identified by their SQL ID and launch the SQL Tuning Advisor to tune them.

#### SQL Tuning Set (STS)

A SQL Tuning Set is a new object for tuning multiple SQL statements and managing SQL workloads. An STS is a database object that stores one or more SQL statements along with their execution statistics and execution context, and possibly with user priority ranking. The SQL statements can be loaded into an STS from different SQL sources. The SQL sources include a list of top SQL statements, the Automatic Workload Repository (AWR), the cursor cache, and specific SQL that you need analyzed. You can select the SQL statements of interest from the Top SQL page and either tune them right away or create an STS that stores the selected SQL statements along with their execution context for later tuning. EM also lets you look at various STSs created by different users. STSs are used as a source for custom SQL tuning by the SQL Tuning and Access Advisors.

# Spot SQL

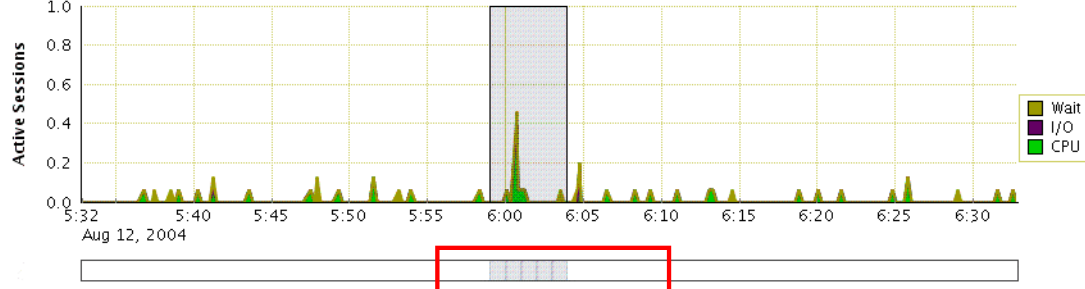
Spot SQL [Period SQL](#)

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

View Data [Real Time: 15 Second Refresh](#)

## 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

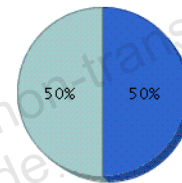
Top SQL (ordered by Activity)

Run SQL Tuning Advisor

Create SQL Tuning Set

Select All | Select None

| Select                   | SQL ID        | SQL Type       | Activity (%) ▾ | CPU (%) | Wait (%) |
|--------------------------|---------------|----------------|----------------|---------|----------|
| <input type="checkbox"/> | 6gvch1xu9ca3g | PL/SQL EXECUTE | 50.00          | 50.00   | 0.00     |
| <input type="checkbox"/> | 2b064ybkwl1y  | PL/SQL EXECUTE | 50.00          | 50.00   | 0.00     |



6gvch1xu9ca3g(50%) 2b064ybkwf1y(50%)

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Spot SQL

This slide shows how SQL statements can be selected from the Top SQL page to be given as input to the SQL Tuning Advisor. The Top SQL page has two tabs: Spot SQL and Period SQL. The illustration shows the contents of the Spot SQL page.

Spot SQL shows all the SQL statements that have been active in a recent five-minute interval. This screen has a drag tool for selecting the five-minute interval for which you want to view details. This page is used for identifying current or recently executed SQL statements.

If you are most concerned with CPU, then examine the top SQL statements that have highest CPU\_TIME during that interval. Otherwise, start with the SQL statement that performed the most DISK\_READS.

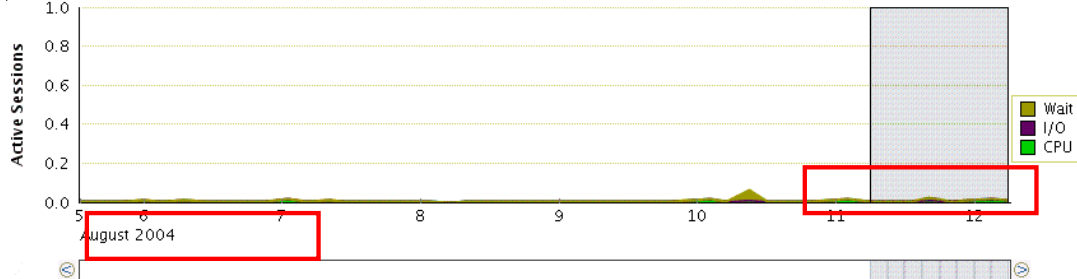
To access these pages, select EM home page > Performance page > Additional Monitoring Links (TOP SQL) > Period SQL or SPOT SQL.

# Period SQL

Spot SQL | **Period SQL**

## Historical Interval Selection

Click on the band below the chart to select the historical 24 hour interval for which you want to view data in the graphs below. Use the active sessions data to help with your selection.



## Detail for Selected 24 Hour Interval

Start Time **Aug 11, 2004 5:00:12 AM**

Run SQL Tuning Advisor

Create SQL Tuning Set

Select All | Select None

Previous

1-25 of 365

Next 25

| Select                   | SQL Text                         | % of Total Elapsed Time | CPU Time (seconds) | Wait Time (seconds) | Elapsed Time Per Execution (seconds) | Module         |
|--------------------------|----------------------------------|-------------------------|--------------------|---------------------|--------------------------------------|----------------|
| <input type="checkbox"/> | BEGIN EMD_NOTIFICATION.QUEUE_R   | 20.76                   | 277.09             | 4.01                | 0.10                                 | OEM.SystemPool |
| <input type="checkbox"/> | call dbms_stats.gather_databases | 10.68                   | 89.23              | 55.44               | 144.67                               |                |
| <input type="checkbox"/> | begin MGMT_JOB_ENGINE.get_sche   | 7.98                    | 106.71             | 1.42                | 0.00                                 | OEM.SystemPool |
| <input type="checkbox"/> | INSERT INTO MGMT_METRICS_RAW (C  | 7.55                    | 74.45              | 27.83               | 0.00                                 | OEM.SystemPool |
| <input type="checkbox"/> | select cust_first_name -- ws     | 6.18                    | 0.50               | 83.25               | 83.76                                | SQL*Plus       |
| <input type="checkbox"/> | DECLARE job BINARY_INTEGER :=    | 4.89                    | 60.02              | 6.17                | 0.05                                 |                |
| <input type="checkbox"/> | select c.CUST_GENDER, c.CUST_M   | 4.01                    | 15.43              | 38.92               | 18.12                                | SQL*Plus       |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Period SQL

The graphic shows the Period SQL page of Top SQL. The information shown is stored in the AWR. It is used for identifying historical high-load SQL. You can use the drag tool to select the historical 24-hour interval for which you want to view data in the graphs. You can use the active sessions data to help with your selection.

Information in the AWR is retained for seven days by default. This means that you can view SQL for the past seven days and identify high-load SQL for any period within those seven days by sliding the drag tool that is provided.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Statspack

Statspack is a set of SQL, PL/SQL, and SQL\*Plus scripts that allow the collection, automation, storage, and viewing of performance data. Statspack offers yet another way of identifying high-load SQL statements. Statspack collects performance data and produces reports, which in addition to high-load SQL statements include an instance health and load summary page, and the traditional wait events and initialization parameters. Statspack can be used with the Standard Edition of Oracle Database if ADDM is not available. For more information, see the Statspack appendix in this course.

The main features of Statspack are that it:

- Collects data about high-load SQL.
- Precalculates many ratios that are useful for performance tuning
- Uses permanent tables owned by the PERFSTAT user to store performance statistics. You can take snapshots at different times to compare performance statistics.
- Separates data collection from report generation. Data is collected when a snapshot is taken. The performance engineer then runs the performance report and views the data collected.
- Makes data collection easy to automate using either DBMS\_JOB or an operating system utility to schedule collection tasks

**Note:** For more information about Statspack, refer to Appendix E.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Using Dynamic Performance Views

Dynamic performance views can also be used in identifying high-load or top SQL statements that are responsible for a large share of the application workload and system resources. This is done by reviewing past SQL execution history that is available in the system.

Using dynamic performance views involves the following steps:

1. Determine which period in the day you would like to examine; typically this is an application's peak processing time when the application is performing slowly.
2. Gather operating system and Oracle statistics at the beginning and end of that period.  
The minimum statistics gathered should be:
  - System statistics from the V\$SYSSTAT view
  - SQL statistics from the V\$SQLAREA view
3. Using the data collected in step 2, identify the SQL statements that use the most resources. A good way to identify candidate SQL statements is to query V\$SQLAREA. V\$SQLAREA contains resource-usage information for all SQL statements in the shared pool. The data in V\$SQLAREA should be ordered by resource usage.

## V\$SQLAREA View

| Column       | Description   |
|--------------|---|
| SQL_TEXT     | First thousand characters of the SQL text   |
| SORTS        | Sum of the number of sorts that were done for all the child cursors                     |
| EXECUTIONS   | Total number of executions, totaled over all the child cursors                          |
| DISK_READS   | Sum of the number of disk reads over all child cursors                                  |
| CPU_TIME     | CPU time (in microseconds) used by this cursor for parsing/executing/fetching           |
| ELAPSED_TIME | Elapsed time (in microseconds) used by this cursor for parsing, executing, and fetching |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### V\$SQLAREA View

The V\$SQLAREA view can also be used to identify high-load SQL. You can query this view to find the SQL statements that most require optimization. The ratio of DISK\_READS/EXECUTIONS identifies the SQL statements that have the most disk hits per execution. High values of CPU\_TIME and ELAPSED\_TIME are indications that the statement is slow to execute and uses significant resources. You should also watch for a high number of sorts.

The most common resources are:

- CPU time (V\$SQLAREA.CPU\_TIME, for statements that use high CPU)
- Disk reads (V\$SQLAREA.DISK\_READS, for high-I/O statements)
- Sorts (V\$SQLAREA.SORTS, for many sorts)

One method to identify the SQL statements that create the highest load is to compare the resources used by a SQL statement to the total amount of that resource used in the period. For CPU\_TIME, divide each SQL statement's CPU\_TIME by the total CPU\_TIME for the period.

## 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;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Querying the V\$SQLAREA View

The slide shows an example of querying the statistics for a specific query from V\$SQLAREA. You can also query for a specific user by joining PARSING\_USER\_ID with the USER\_ID column of the DBA\_USERS view. Additionally, you can query for large values for DISK\_READS (excess of 100,000).

The output for this query is as follows:

| SQL_TEXT    | DISK_READS | BUFFER_GETS | SORTS | CPU_TIME | ELAPSED_TIME |
|-------------|------------|-------------|-------|----------|--------------|
| select      | 23078      | 16166       | 5     | 17225551 | 20001832     |
| c.CUST_     |            |             |       |          |              |
| GENDER, ... |            |             |       |          |              |



# Investigating Full Table Scan Operations

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

| NAME                           | VALUE    |
|--------------------------------|----------|
| -----                          | -----    |
| table scans (short tables)     | 217842   |
| table scans (long tables)      | 3040     |
| table scans (rowid ranges)     | 254      |
| table scans (cache partitions) | 7        |
| table scans (direct read)      | 213      |
| table scan rows gotten         | 40068909 |
| table scan blocks gotten       | 1128681  |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Investigating Full Table Scan Operations

A full table scan (FTS) operation on a large table is a very resource-intensive operation that should be minimally used. To find out if you have too many FTS operations in the database, you can query the V\$SYSSTAT view during an application's peak usage time (as shown).

The results of the query provide an overview of how many full table scans are taking place.

The values for table scans (long tables) and table scans (short tables) relate to full table scans. A long table is defined as having blocks higher than 2% of the buffer cache and a short table has blocks fewer or equal to 2% of the buffer cache.

If the value of table scans (long tables) is high, then a large percentage of the tables accessed were not indexed lookups. Your application may need tuning, or you should add indexes. Make sure that the appropriate indexes are in place and valid. As you create indexes on your large tables, the FTS number should decrease. If it does not, the wrong indexes are being created.

# Summary

**In this lesson, you should have learned about the different methods to identify high-load SQL:**

- **ADDM**
- **Top SQL**
- **Statspack**
- **Dynamic performance views**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Summary

Identifying high-load (problematic) SQL is the first step in the SQL Tuning process. In this lesson, you should have learned about the different tools available for identifying high-load SQL.

# 10

## Automatic SQL Tuning

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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**

**ORACLE**

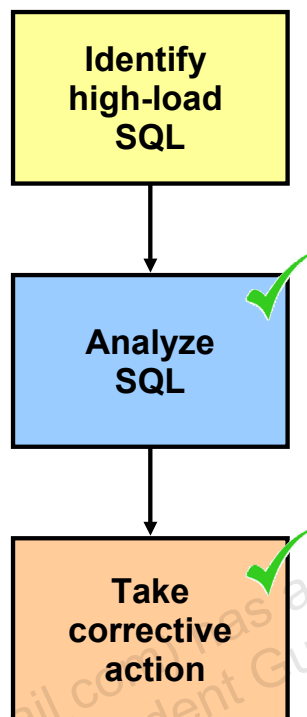
Copyright © 2007, Oracle. All rights reserved.

## Objectives

Oracle Enterprise Manager (EM) is a graphical application to administer Oracle databases. This lesson concentrates on the automatic SQL tuning features of Oracle Database 10g using EM, which is part of the Oracle Tuning Pack.

Because the automatic SQL tuning solution is part of the database kernel and is not an external tool, it can also be accomplished through the command-line interface using PL/SQL APIs.

# SQL Tuning Process: Overview



ORACLE

Copyright © 2007, Oracle. All rights reserved.

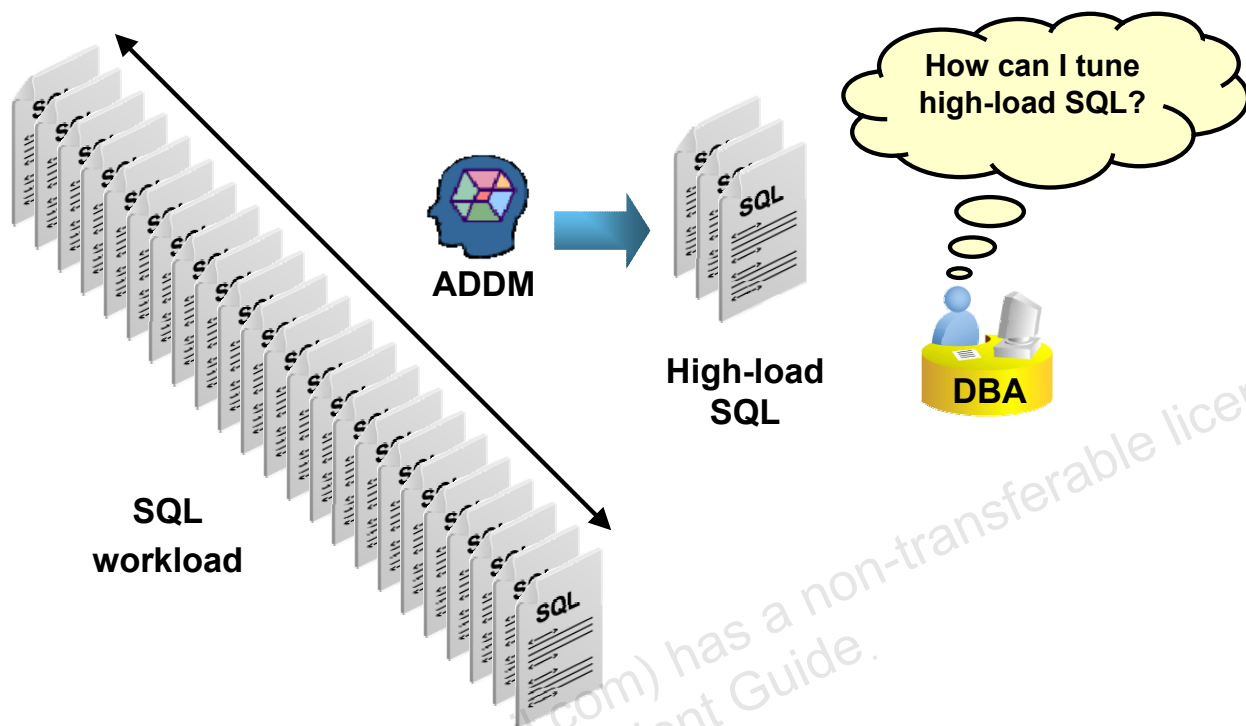
## SQL Tuning Process: Overview

SQL tuning involves three basic steps:

- Identify high-load or top SQL statements that are responsible for a large share of the application workload and system resources, by looking at the past SQL execution history available in the system (for example, the ADDM report or the Top SQL page on EM, or the cursor cache statistics stored in the V\$SQL dynamic view).
- Analyze SQL to verify that the execution plans produced by the query optimizer for these statements perform reasonably well.
- Take corrective action to generate better execution plans for poorly performing SQL statements.

In this lesson, you learn how to perform the last two steps by using the automatic SQL tuning mechanism.

# Automatic SQL Tuning



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Automatic SQL Tuning

Manual SQL tuning consists of steps such as the following:

- Gather information about the referenced objects.
- Verify optimizer statistics.
- Review execution plans.
- Restructure SQL statements.
- Create indexes and materialized views.

All of these tasks have been automated in Oracle Database 10g by using the automatic SQL tuning feature. Using the interface provided, you can easily improve performance of high-load SQL by following the recommendations of the SQL Tuning and Access advisors that are included in the Enterprise Edition of Oracle Database.

### What Is Automatic SQL Tuning?

Automatic SQL tuning is a new solution in Oracle Database 10g that automates the entire SQL tuning process by comprehensively exploring all the possible ways of tuning a SQL statement. It is performed by the database engine's query optimizer, which has been significantly enhanced in Oracle Database 10g. Automatic SQL tuning offers a comprehensive SQL tuning solution that includes the Automatic Workload Repository (AWR), the SQL Tuning Advisor (STA), the SQL Access Advisor (SAA), and SQL Tuning Sets.

# Automatic Tuning Optimizer

- **Is the query optimizer running in tuning mode**
- **Performs verification steps**
- **Performs exploratory steps**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Automatic Tuning Optimizer

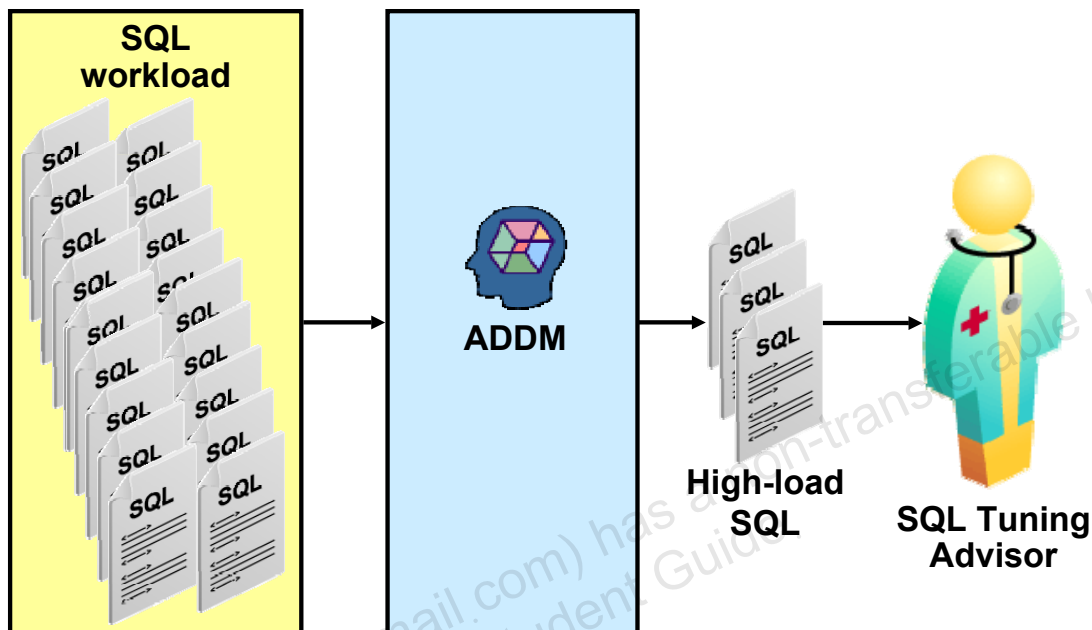
Automatic SQL tuning is performed by the newly enhanced optimizer called the Automatic Tuning Optimizer (ATO). Having the optimizer perform tuning gives several advantages:

- The tuning is done by the system component that is ultimately responsible for the execution plans and hence the SQL performance.
- The tuning process is fully cost based, and it naturally accounts for any changes and enhancements done to the query optimizer.
- The tuning process takes into account the past execution statistics of a SQL statement and customizes the optimizer settings for that statement.
- The ATO collects auxiliary information in conjunction with the regular statistics based on what is considered useful by the query optimizer.

The ATO, when performing SQL tuning, uses dynamic sampling and partial execution techniques to verify its own estimates of cost, selectivity, and cardinality. It also uses past execution history of the SQL statement to determine optimal settings such as `OPTIMIZER_MODE`. This enables the generation of better execution plans. The functionality of the ATO is exposed via two advisors, the SQL Tuning Advisor and the SQL Access Advisor.

These advisors accept one or more SQL statements as input and pass the input to the ATO along with other input parameters. The ATO then tunes these statements.

# SQL Tuning Advisor



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Tuning Advisor

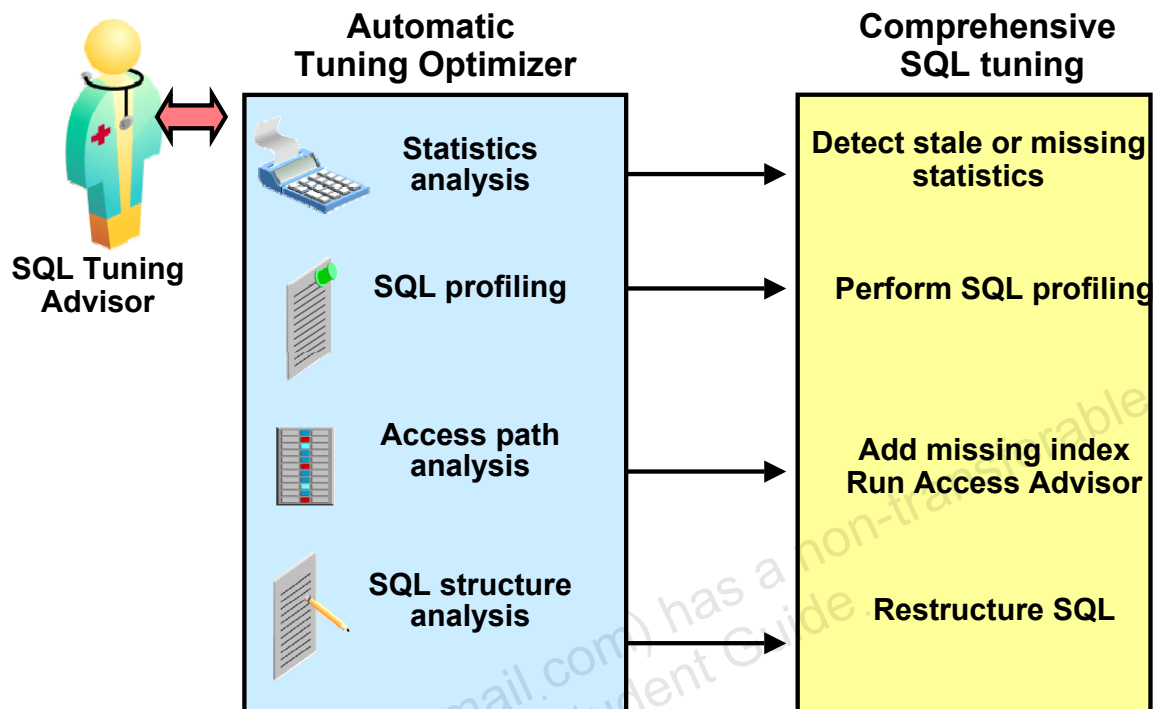
The next step in the SQL tuning process, following the identification of high-load SQL, is to analyze and tune the SQL. Even though the number of high-load SQL statements identified by Automatic Database Diagnostic Monitor (ADDM) may represent a very small percentage of the total SQL workload, the task of tuning them is still highly complex and requires a high level of expertise.

The SQL Tuning Advisor (STA) is a new feature of Oracle Database 10g that users interact with to perform automatic SQL tuning. The advisor receives one or more SQL statements as input and provides advice on how to optimize their execution plan, rationale for the advice, the estimated performance benefit, and the actual command to implement the advice. The user can simply choose to accept the advice, thus tuning the SQL statements.

As mentioned earlier, automatic SQL tuning is performed by the ATO. The STA is just the interface through which you invoke the query optimizer in this special mode. The actual tuning is performed by the optimizer itself.



# SQL Tuning Advisor Analysis



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Tuning Advisor Analysis

The ATO performs four types of analysis during the plan-generation process when invoked by the SQL Tuning Advisor, resulting in comprehensive SQL tuning.

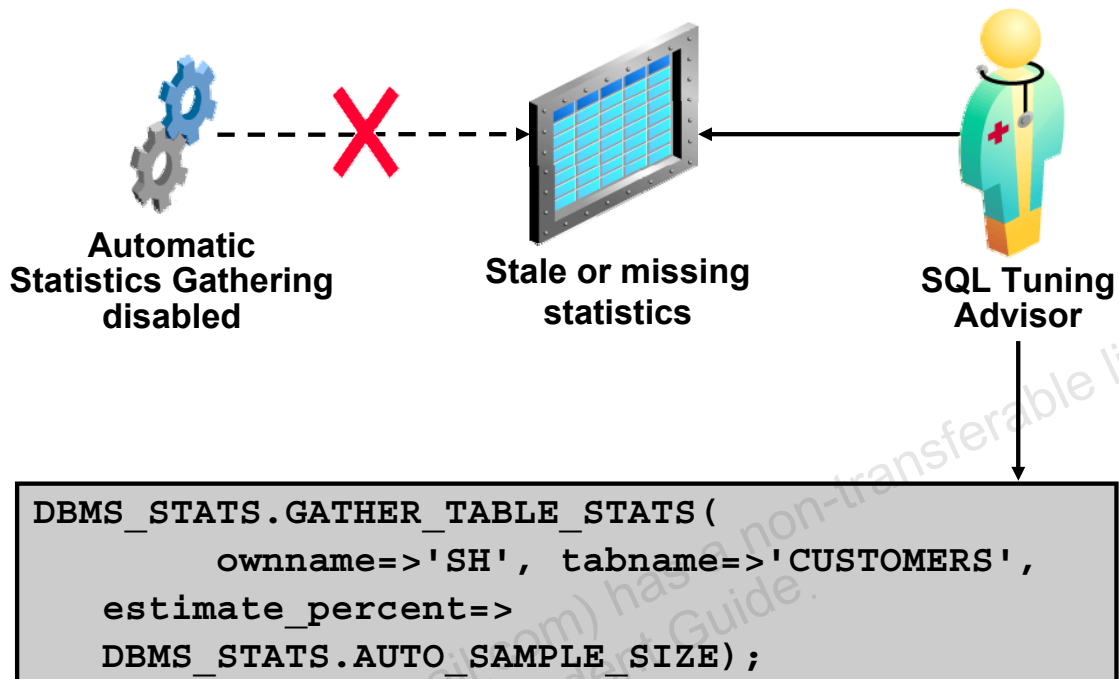
**Statistics analysis:** The ATO checks each query object for missing or stale statistics, and makes a recommendation to gather relevant statistics.

**SQL profiling:** The ATO verifies its own estimates and collects auxiliary information to remove estimation errors. It also collects auxiliary information in the form of customized optimizer settings (for example, first rows versus all rows) based on past execution history of the SQL statement. It designs a SQL Profile using the auxiliary information and makes a recommendation to create it. When a SQL Profile is created, it enables the ATO (under normal mode) to generate a well-tuned plan.

**Access path analysis:** The ATO explores whether a new index can be used to significantly improve access to each table in the query, and when appropriate makes recommendations to create such indexes.

**SQL structure analysis:** Here the ATO tries to identify SQL statements that lend themselves to bad plans, and makes relevant suggestions to restructure them. The suggested restructurings can be syntactic as well as semantic changes to the SQL code.

# Statistics Analysis



ORACLE

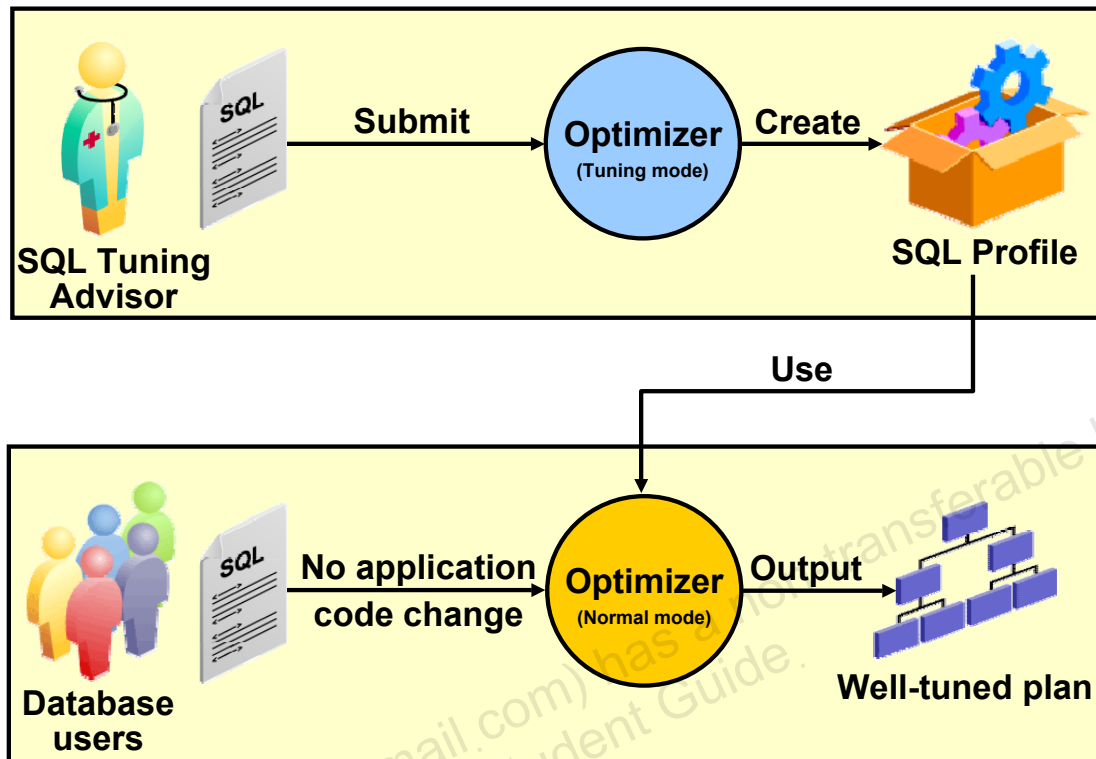
Copyright © 2007, Oracle. All rights reserved.

## Statistics Analysis

Accurate statistics are a key input if the optimizer is to generate optimal execution plans. Oracle Database 10g introduces Automatic Statistics Gathering to maintain up-to-date optimizer statistics. Generally, the optimizer statistics should not be stale or missing in Oracle Database 10g as a result of this new feature. However, if this feature is disabled for some reason, the statistics may become stale or (what is worse) may be completely missing and may thus require updating.

In the check statistics mode, the ATO runs extra checks to verify the statistics that it uses, and makes recommendations to gather statistics where appropriate. In this analysis, the ATO also generates additional information to compensate for missing or stale statistics using very efficient sampling techniques. This information is stored in a new object called *SQL Profile*, which is discussed later in more detail. If you choose not to collect statistics immediately on the specific objects as recommended, you have the option to accept the SQL Profile. The contents of this profile provide the optimizer with sufficient statistical information to enable the generation of a better execution plan. However, this information is not as comprehensive as that generated by the DBMS\_STATS package. The recommended practice, therefore, is to use the DBMS\_STATS package to manually gather statistics on objects identified as having missing or stale statistics in this analysis phase.

# SQL Profiling



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Profiling

The query optimizer can sometimes produce inaccurate estimates about an attribute of a statement due to lack of information, leading to poor execution plans. Traditionally, users have corrected this problem by manually adding hints to the application code to guide the optimizer into making correct decisions. For packaged applications, changing application code is not an option and the only alternative available is to log a bug with the application vendor and wait for a fix.

Automatic SQL tuning deals with this problem with its SQL profiling capability. In this phase, the ATO collects more information about the specific SQL statement being tuned. It performs verification steps to validate its own cost estimates of the SQL statement. The validation consists of taking a sample of data and applying appropriate predicates to the sample. A new estimate is produced from the data sample result.

Another method of estimate validation involves the execution of a fragment of the SQL statement called partial execution. The partial execution method is more efficient than the sampling method when respective predicates provide efficient access paths. The ATO picks the appropriate estimate validation method and also uses the past execution history of the SQL statement to determine correct OPTIMIZER\_MODE settings.

## SQL Profiling (continued)

For example, if the execution history shows that only a few rows are fetched even though the overall result set of the SQL statement is large, then the appropriate `OPTIMIZER_MODE` setting will be `FIRST_ROWS` instead of the default `ALL_ROWS`. This will be a customized setting for the SQL statement being tuned.

The output of this analysis phase is a SQL Profile, which is a container of additional information specific to the SQL statement collected here. This information includes data skew adjustment factors, predicate selectivity information, cost adjustments, and so on. Thus, a SQL Profile is to a SQL statement what optimizer statistics are to an object. This additional information is then used in conjunction with the optimizer statistics to produce a well-tuned execution plan for the corresponding SQL statement without requiring any change to the application code. This makes SQL profiling an ideal solution for tuning packaged applications as well.

A SQL Profile, after it is created, is stored persistently in the data dictionary. If a SQL Profile exists for a SQL statement, every time the corresponding SQL statement is compiled (that is, optimized) the query optimizer will use the contents of the SQL Profile along with the optimizer statistics to produce a well-tuned plan. A full set of functions is provided for the management of SQL Profiles.

Because the estimate validation using the method of data sampling or partial execution can be an expensive and time-consuming process, SQL profiling is not performed when the SQL Tuning Advisor is run in Limited mode. You need to use Comprehensive mode to let the ATO generate a SQL Profile.

It is important to note that the SQL Profile does not freeze the execution plan of a SQL statement, as is the case with stored outlines. As tables grow or indexes are created or dropped, the execution plan can change with the same SQL Profile. The information stored in it continues to be relevant even as the data distribution or access path of the corresponding statement changes. However, over a long period of time, its content can become outdated and must be regenerated. This can be done by running automatic SQL tuning again on the same statement to regenerate the SQL Profile.

### Managing SQL Profiles with APIs

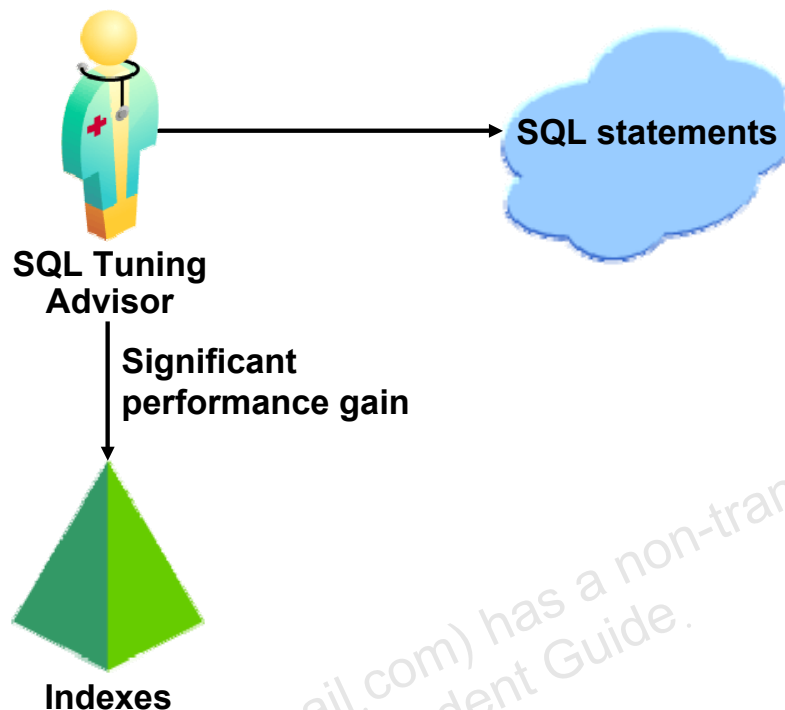
While SQL Profiles are usually handled by Oracle Enterprise Manager as part of the automatic SQL tuning process, SQL Profiles can be managed through the `DBMS_SQLTUNE` package. To use the SQL Profiles APIs, you need the `CREATE ANY SQL_PROFILE`, `DROP ANY SQL_PROFILE`, and `ALTER ANY SQL_PROFILE` system privileges.

You can use the `DBMS_SQLTUNE.ACCEPT_SQL_PROFILE` procedure to accept a SQL Profile recommended by the SQL Tuning Advisor. This creates and stores a SQL Profile in the database. You can view information about a SQL Profile in the `DBA_SQL_PROFILES` view.

You can alter the attributes of an existing SQL Profile with the `ALTER_SQL_PROFILE` procedure.

You can drop a SQL Profile with the `DROP_SQL_PROFILE` procedure.

# SQL Access Path Analysis



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Access Path Analysis

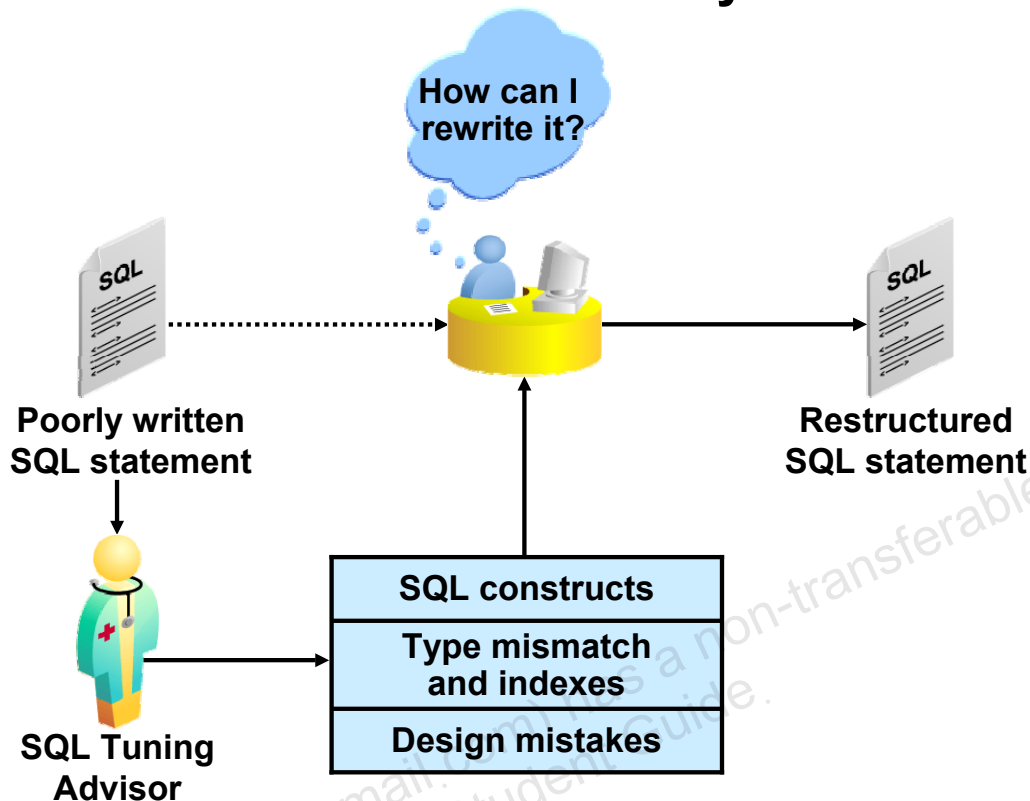
During the access path analysis, the ATO determines whether additional indexes are needed to enhance performance of the SQL statement being tuned. Because an index may be dropped by mistake, may become unusable, or may never be created, adding an index may significantly improve performance of the SQL statement.

The output of this analysis consists of the following recommended steps:

- Create new indexes if they provide significantly superior performance.
- Run the SQL Access Advisor (SAA) to perform a comprehensive index analysis based on application workload.

The SAA is a new solution featured in Oracle Database 10g for comprehensive access path analysis. (It is discussed in more detail later in the lesson.) Because the SQL Tuning Advisor tunes one SQL statement at a time, it does not analyze how its index recommendation can affect the entire SQL workload. This is why it also recommends running the SSA on the SQL statement along with a representative SQL workload, which will look at the impact of creating an index on the entire SQL workload before making any recommendations.

# SQL Structure Analysis



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Structure Analysis

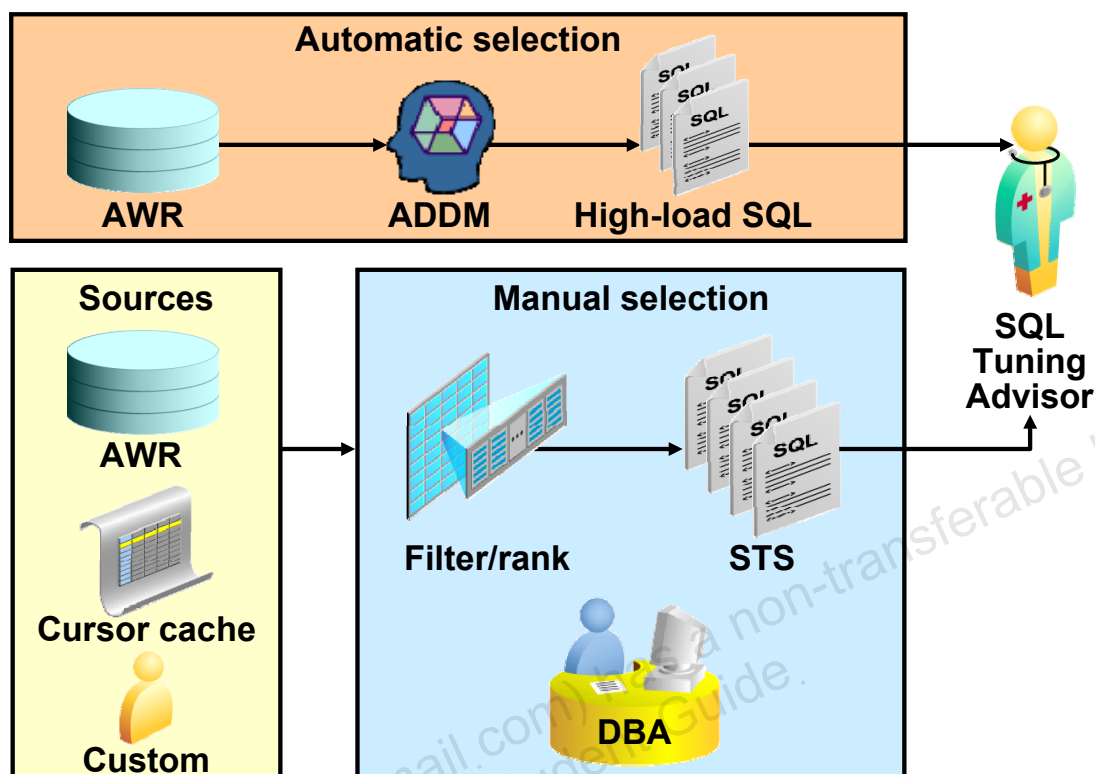
The goal of SQL Structure Analysis is to help you to identify poorly written SQL statements as well as to give you advice on how to restructure them.

There are certain syntax variations that are known by Oracle experts to have a negative impact on performance. In SQL Analysis mode, the optimizer evaluates statements against a set of rules, identifying less efficient coding techniques and providing recommendations for an alternative statement where possible. The recommendation is very similar (but not precisely equivalent) to the original query (for example, the recommendation to use UNION ALL instead of the UNION constructor). The constructs are similar but not identical. Hence, you have to decide whether the recommendation is applicable to you. For this reason, the optimizer does not automatically rewrite the query but gives advice instead.

The categories of problems detected by SQL Structure Analysis are:

- Use of SQL constructors such as NOT IN instead of NOT EXISTS (preventing unnesting)
- Use of predicates involving indexed columns with data type mismatch that prevents the use of the index
- Design mistakes (like Cartesian products)

# SQL Tuning Advisor: Usage Model



Copyright © 2007, Oracle. All rights reserved.

## SQL Tuning Advisor: Usage Model

The SQL Tuning Advisor takes one or more SQL statements as the input, which can come from many different sources:

- High-load SQL statements identified by ADDM
- SQL statements that are currently in the cursor cache
- SQL statements from the AWR: A user can select any set of SQL statements captured by the AWR. By default, the AWR maintains data for up to seven days. Hence, a user can go back to any time during the last seven days and tune statements selectively.
- Custom workload: A user can create a custom workload consisting of statements of interest to that user. These may be statements that are not in the cursor cache and are not high-load to be captured by ADDM or the AWR. For such statements, a user can create a custom workload and tune it using the advisor.

SQL statements from the cursor cache, the AWR, and a custom workload can be filtered and/or ranked before they are input to the SQL Tuning Advisor.

For multiple statement input, a new object called the *SQL Tuning Set* (STS) is provided. An STS stores multiple SQL statements along with their execution information:

- Execution context: Parsing schema name, bind values, and so on
- Execution statistics: Execution count, average elapsed time, CPU time, and so on



## Using SQL Tuning Advisor APIs

Although the primary interface for the SQL Tuning Advisor is the Oracle Enterprise Manager Database Control, the advisor can be administered with procedures in the DBMS\_SQLTUNE package. To use the APIs, the user must have been granted the DBA role and the ADVISOR privilege.

Running SQL Tuning Advisor with the DBMS\_SQLTUNE package is a two-step process:

1. Create a SQL tuning task.
2. Execute a SQL tuning task.

### Creating a SQL Tuning Task

You can create tuning tasks from the text of a single SQL statement, a SQL tuning set containing multiple statements, a SQL statement selected by a SQL identifier from the cursor cache, or a SQL statement selected by a SQL identifier from the Automatic Workload Repository. You can use the DBMS\_SQLTUNE.CREATE\_TUNING\_TASK function to create a SQL tuning task. The CREATE\_TUNING\_TASK function either returns the task name that you provide or generates a unique task name. You can use the task name to specify this task when using other APIs. To view the task names that are associated with a specific owner, use the following query

```
SELECT task_name FROM DBA_ADVISOR_LOG WHERE owner = 'HR';
```

### Executing a Tuning Task

After you have created a tuning task, you need to execute the task and start the tuning process. You can use DBMS\_SQLTUNE.EXECUTE\_TUNING\_TASK to execute a previously created SQL tuning task:

```
Exec DBMS_SQLTUNE.EXECUTE_TUNING_TASK( task_name =>
    'my_sql_tuning_task' );
```

You can check the status of the task by reviewing the information in the DBA\_ADVISOR\_LOG view, or you can check execution progress of the task in the V\$SESSION\_LONGOPS view. Here is an example:

```
SELECT status FROM DBA_ADVISOR_LOG WHERE task_name =
    'my_sql_tuning_task';
```

### Displaying the Results of a Tuning Task

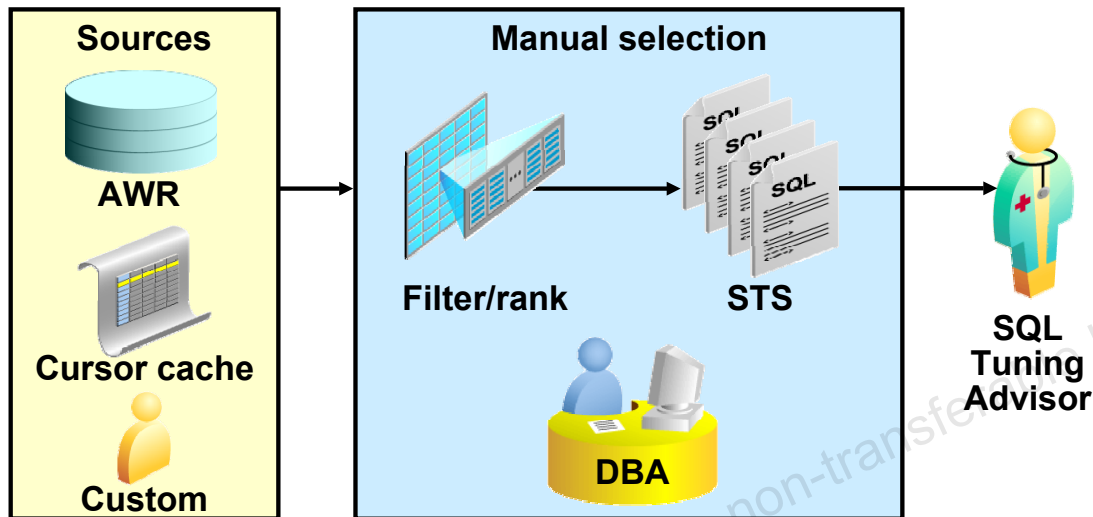
After a task has been executed, you display a report of the results with the REPORT\_TUNING\_TASK function, as in the following example:

```
SELECT DBMS_SQLTUNE.REPORT_TUNING_TASK( 'my_sql_tuning_task')
FROM DUAL;
```

The report contains all the findings and recommendations of automatic SQL tuning. For each proposed recommendation, the rationale and benefit are provided along with the SQL commands that are needed to implement the recommendation.



# SQL Tuning Set



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Tuning Set

The ADDM enables automatic identification of high-load SQL statements for the user to choose from and tune. The AWR enables the selection of top SQL statements over a time interval. However, the user may want to tune a set of custom SQL statements in a user-specified priority order. A good example of this situation is when a developer is in the process of developing new SQL statements and testing them. Since this SQL is not yet deployed into the system, the usual SQL sources (for example, the AWR) cannot be used. So there needs to be a mechanism for the user to create his or her own custom SQL workload and perform automatic SQL tuning on it.

The answer is the STS. Support for the STS has been introduced in Oracle Database 10g to make it easy for the user to manage a related set of SQL statements as a unit. An STS is a database object that stores one or more SQL statements along with their execution statistics and execution context, and possibly with user priority ranking. The SQL statements can be loaded into an STS from different SQL sources. The SQL sources include the Automatic Workload Repository (AWR), the cursor cache, and the custom SQL provided by the user.

The figure shows an STS usage model. An STS is created and then loaded with SQL statements from one of the SQL sources. As illustrated, the SQL statements are selected, ranked, and filtered before they are loaded into an STS.

## SQL Tuning Set (continued)

### Managing SQL Tuning Sets

The SQL Tuning Set APIs enable you to manage SQL Tuning Sets to determine performance information about SQL statements running on your system. You typically use the STS operations in the following sequence:

1. Create a new STS.
2. Load the STS.
3. Select the STS to review the contents.
4. Update the STS if necessary.
5. Create a tuning task with the STS as input.
6. Drop the STS when finished.

To use the APIs, you need the ADMINISTER ANY SQL TUNING SET system privilege.

### Creating a SQL Tuning Set

The `CREATE_SQLSET` procedure is used to create an empty STS object in the database. For example, the following procedure creates an STS object that could be used to tune I/O-intensive SQL statements during a specific period of time:

```
exec DBMS_SQLTUNE.CREATE_SQLSET  
( sqlset_name => 'my_sql_tuning_set', description => 'I/O  
intensive workload');
```

In this procedure, `my_sql_tuning_set` is the name of the STS in the database and `I/O intensive workload` is the description that is assigned to the STS.

### Loading a SQL Tuning Set

The `LOAD_SQLSET` procedure populates the STS with selected SQL statements. The standard sources for populating an STS are the workload repository, another STS, or the cursor cache. For both the workload repository and the STS, there are predefined table functions that can be used to select columns from the source to populate a new STS.

In the following example, procedure calls are used to load `MY_SQL_TUNING_SET` from an AWR baseline called `peak baseline`. The data has been filtered to include only those SQL statements that have been executed at least 10 times and that have a ratio of disk-reads divided by buffer gets greater 50% during the baseline period. The SQL statements are ordered by the ratio, with only the top 30 SQL statements selected. First, a REF cursor is opened to select from the specified baseline. Then the statements and their statistics are loaded from the baseline to the STS.

## SQL Tuning Set (continued)

```
DECLARE
baseline_cursor DBMS_SQLTUNE.SQLSET_CURSOR;
BEGIN
OPEN baseline_cursor FOR SELECT VALUE(p) FROM TABLE
(DBMS_SQLTUNE.SELECT_WORKLOAD_REPOSITORY( 'peak baseline',
'executions >= 10 AND disk_reads/buffer_gets >= 0.5', NULL,
'disk_reads/buffer_gets', NULL, NULL, NULL, 30)) p;
DBMS_SQLTUNE.LOAD_SQLSET( sqlset_name => 'my_sql_tuning_set',
populate_cursor => baseline_cursor);
END;
/
```

### Displaying the Contents of a SQL Tuning Set

The `SELECT_SQLSET` table function reads the contents of the STS. After an STS has been created and populated, you can browse through the SQL in the STS using the `SELECT_SQLSET` procedure.

In the following example, the SQL statements in the STS are displayed for statements that have a ratio of disk-reads to buffer-gets that is greater than or equal to 75%.

```
SELECT * FROM TABLE(DBMS_SQLTUNE.SELECT_SQLSET(
'my_sql_tuning_set', '(disk_reads/buffer_gets) >= 0.75')); Additional
details about the SQL Tuning Sets that have been created and loaded can also be displayed
with DBA views such as DBA_SQLSET, DBA_SQLSET_STATEMENTS, and
DBA_SQLSET_BINDS.
```

### Modifying a SQL Tuning Set

SQL statements can be updated and deleted from a SQL Tuning Set based on a search condition. In the following example, the `DELETE_SQLSET` procedure deletes SQL statements from `my_sql_tuning_set` that have been executed fewer than 50 times.

```
exec DBMS_SQLTUNE.DELETE_SQLSET( sqlset_name =>
'my_sql_tuning_set', basic_filter => 'executions < 50');
```

### Dropping a SQL Tuning Set

The `DROP_SQLSET` procedure is used to drop an STS that is no longer needed, as in the following example:

```
Exec DBMS_SQLTUNE.DROP_SQLSET(sqlset_name => my_sql_tuning_set');
```

### Additional Operations on SQL Tuning Sets

The `UPDATE_SQLSET` procedure updates the attribute values of an existing STS that is identified by STS name and SQL identifier.

The `SELECT_WORKLOAD_REPOSITORY` function enables the creation of an STS by returning an STS from a snapshot or baseline.

The `ADD_SQLSET_REFERENCE` function adds a new reference to an existing STS to indicate its use by a client. The function returns the identifier of the added reference. The `REMOVE_SQLSET_REFERENCE` procedure deactivates an STS to indicate that it is no longer used by the client.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Tuning Advisor Views

You can query the data dictionary to find the information displayed by ADDM and used by STA. You need DBA privileges to access these views.

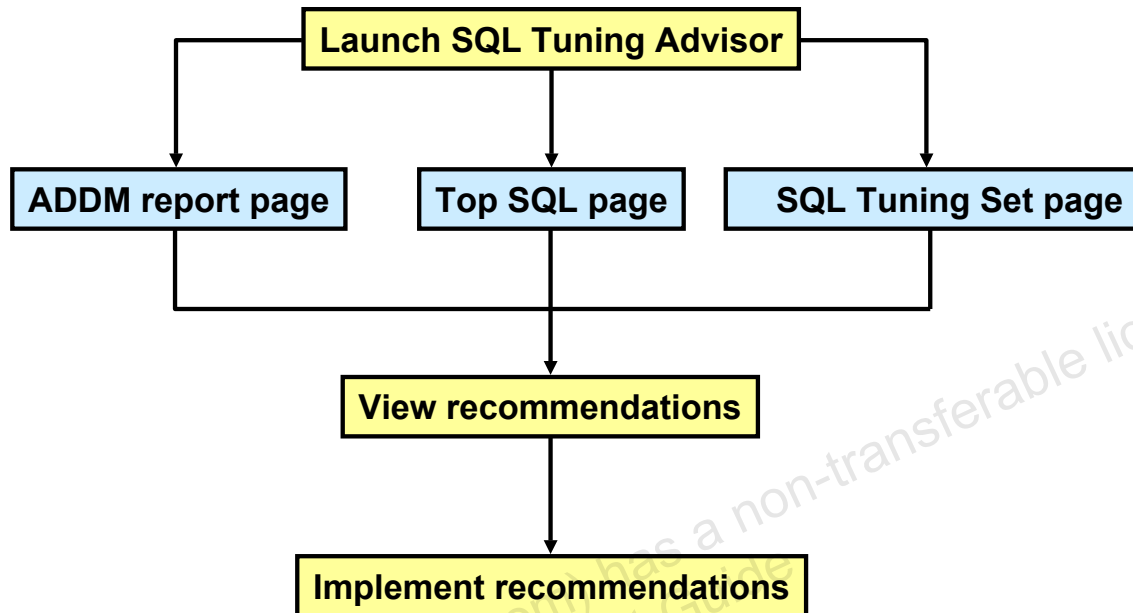
The Advisor information is available in the DBA\_ADVISOR\_TASKS, DBA\_ADVISOR\_FINDINGS, DBA\_ADVISOR\_RECOMMENDATIONS, and DBA\_ADVISOR\_RATIONALE views.

SQL tuning information is stored in the DBA\_SQLTUNE\_STATISTICS, DBA\_SQLTUNE\_BINDS, and DBA\_SQLTUNE\_PLANS views.

SQL Tuning Set information is available in the DBA\_SQLSET, DBA\_SQLSET\_BINDS, DBA\_SQLSET\_STATEMENTS, and DBA\_SQLSET\_REFERENCES views.

SQL Profile information is displayed in the DBA\_SQL\_PROFILES view.

# Enterprise Manager: Usage Model



ORACLE

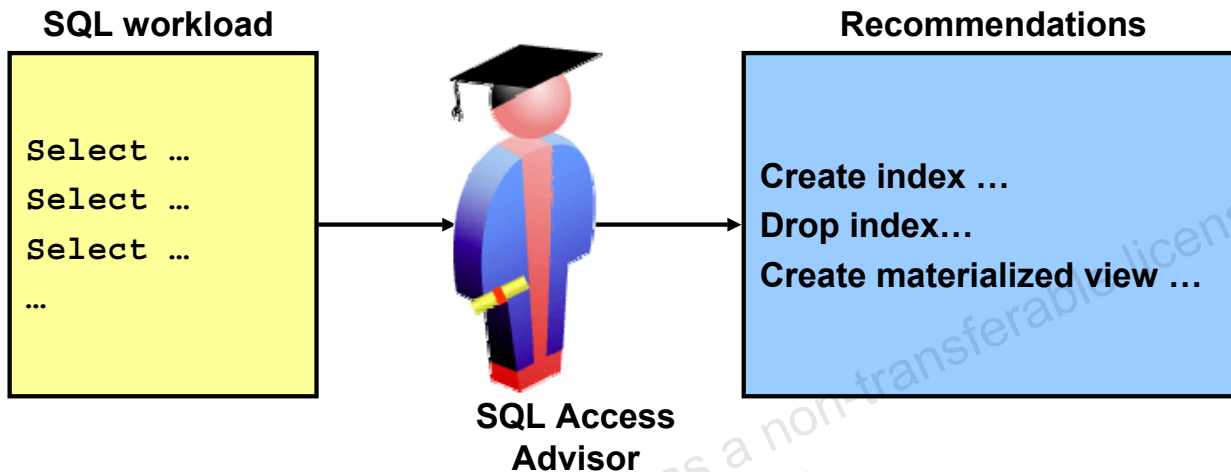
Copyright © 2007, Oracle. All rights reserved.

## Enterprise Manager: Usage Model

The SQL Tuning Advisor has both GUI and command-line interfaces. The preferred way to use it is with the Enterprise Manager graphical interface, as indicated in the following steps:

1. Launch the SQL Tuning Advisor from SQL source pages:
  - ADDM report page
  - Top SQL page
  - SQL Tuning Set (STS) page
2. View the SQL Tuning Advisor recommendations.
3. Implement the recommendations.

# SQL Access Advisor



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Access Advisor (SAA)

Together with the SQL Tuning Advisor, the SAA completes the automatic SQL tuning solution. It is the primary source for advice about the access structure. It recommends indexes and materialized views to create, retain, or drop based on the supplied workload.

The SAA is supplied with several default templates; the template defines all the rules for generating the recommendations. For example, the rules might include which mode to use, whether only indexes are to be created, how the new objects should be named, and so on.

## SQL Access Advisor (continued)

Just like the SQL Tuning Advisor, the SAA first makes its recommendations and then lets you decide whether to implement all of its recommendations, only some of them, or none of them. As with the SQL Tuning Advisor, it gives an expected performance benefit for each recommendation and also generates the SQL script for the implementation of its recommendations. You can implement one or all of its recommendations by a simple click of a button.

There are two main differences between the SAA and the Access Path analysis of the SQL Tuning Advisor. First, the SAA takes the entire workload into consideration, and its recommendations improve the performance of the entire workload and not just an individual SQL statement. This means that if the SQL workload contains DML statements like inserts, updates, and deletes, the SAA will take into account the impact of any new indexes or materialized views on the performance of these statements before making any recommendations. In contrast, the SQL Tuning Advisor tunes one SQL statement in isolation from the others, and its index recommendations therefore do not take into account the impact that the new indexes will have on DML operations. For this reason, any time the SQL Tuning Advisor determines that an index will improve the performance of a statement, it always recommends the running of the SAA as well. In this way, the user can ensure that DML operations are not impacted too adversely by the new index.

The second main difference between the two advisors is that the SQL Tuning Advisor recommends only B\*-tree indexes, while the SAA recommends different types of indexes as well as materialized views and materialized view logs. This difference arises because the SQL Tuning Advisor does not take the entire SQL workload into account, but the SAA does.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Access Advisor: Features

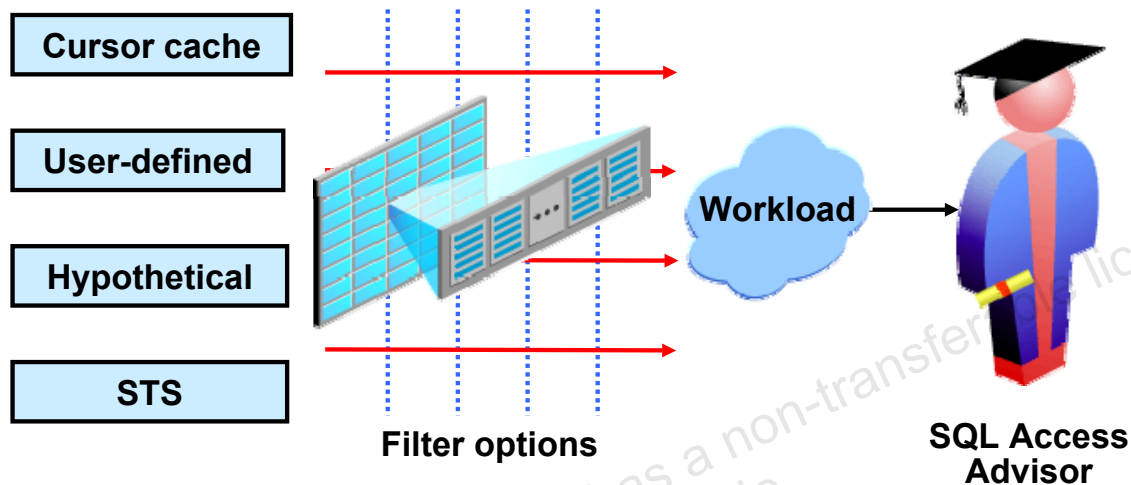
The SAA simplifies access structure design for optimal application performance. Materialized views and indexes are essential when tuning a database to achieve optimum performance for complex, data-intensive queries. The SAA helps you achieve your performance goals by recommending the proper set of materialized views, materialized view logs, and indexes for a given workload.

The SAA recommends both bitmap indexes and B\*-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B\*-tree indexes are useful for high-selectivity columns.

Another component of the SAA recommends how to optimize materialized views so that they can be fast refreshable and can take advantage of general query rewrite.



# SQL Access Advisor: Usage Model



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Workload Sources

The SAA uses the common workload repository used by other advisors. It can take a workload derived from many sources:

- Cursor cache:
  - Takes current contents of V\$SQL.
- User-defined:
  - Specify your workload in an input table
- Hypothetical:
  - SAA generates a likely workload from your dimensional model.
- STS from the workload repository

## Filter Options

SAA provides powerful workload filters to enable a user to target the tuning. For example, a user can specify that the advisor should look at only the 30 most resource-intensive statements in the workload based on optimizer cost.

You can use the DBMS\_ADVISOR package to manually perform the following tasks:

- Create a task.
- Define the workload.
- Generate the recommendations.
- View and implement the recommendations.

# SQL Access Advisor: User Interface

**Oracle Enterprise Manager**

Host: nedc-smp2.us.oracle.com > Database: demoDB > Advisor Central

Logged in As grocery

Collected From Target July 23, 2003 7:06:12 AM EDT

**Advisors**

ADDM, SQL Tuning Advisor, SQL Access Advisor, Memory Advisor, MTTR Advisor, Segment Advisor, Undo Management

**Advisor Tasks**

**Search**

Select an advisory type and optionally enter a task name to filter the data that is displayed in your results set.

Advisory Type: All Types, Task Name: , Advisor Runs: Last Run, Go

**Results**

| Select                           | Advisory Type      | Name             | Description  | User   | Status    | Start Time           | End Time             | Expires In (days) |
|----------------------------------|--------------------|------------------|--|--------|-----------|----------------------|----------------------|-------------------|
| <input checked="" type="radio"/> | ADDM               | ADDM_1_36_37     | ADDM run between snapshots 36 and 37 in database with id 3310336926 and instance 1 | SYS    | COMPLETED | 23-Jul-2003 00:00:00 | 23-Jul-2003 00:00:00 | 30                |
| <input type="radio"/>            | SQL Access Advisor | SQLACCESS2394833 | SQL Access Advisor   | SYSTEM | COMPLETED | 22-Jul-2003 00:00:00 | 22-Jul-2003 00:00:00 | 29                |

Copyright © 1996, 2003, Oracle. All rights reserved.

## SQL Access Advisor: User Interface

The SAA has both graphical and command-line user interfaces. The easiest way to access it is to invoke it from Enterprise Manager with the following steps:

1. Launch SQL Access Advisor from Advisor Central.
2. Select the workload source.
3. Set the options:
  1. Workload
  2. Recommendation
4. Schedule the job.
5. Review the job and submit.
6. Monitor the job.
7. View the recommendations.
8. Implement the recommendations.

# SQL Tuning Advisor and SQL Access Advisor

| Analysis Types                      | Advisor                   |
|-------------------------------------|---------------------------|
| Statistics                          | SQL Tuning Advisor        |
| SQL Profile                         | SQL Tuning Advisor        |
| SQL Structure                       | SQL Tuning Advisor        |
| Access Path: Indexes                | SQL Tuning/Access Advisor |
| Access Path: Materialized Views     | SQL Access Advisor        |
| Access Path: Materialized View Logs | SQL Access Advisor        |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## SQL Tuning Advisor and SQL Access Advisor

The SQL Tuning Advisor and the SQL Access Advisor together completely automate SQL tuning. The table shows the types of analysis performed by the two advisors.

# 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**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Summary

Manual tuning is a very tedious process. The automatic tuning features in Enterprise Manager facilitate the tuning process through automatic monitoring and interactive advisors.

# 11

## Index Usage

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# Objectives

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

- **Identify index types**
- **Identify basic access methods**
- **Monitor index usage**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Objectives

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

- Identify index types
- Create indexes manually
- Identify basic access methods
- Use skip-scanning of indexes
- 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Indexing Guidelines

Although the optimizer avoids nonselective indexes in query execution, the Oracle Server must maintain all indexes defined against a table regardless of whether they are used. Index maintenance can present a significant CPU and I/O resource demand in any write-intensive application. In other words, you should build indexes only if necessary. To maintain optimal performance, drop indexes that an application is not using by using the index monitoring facility. However, an index that is used to enforce primary keys and unique keys cannot be dropped.

If you are deciding whether to create new indexes to tune statements, then you can also use the EXPLAIN PLAN statement to determine if the optimizer will choose to use these indexes when the application is run. If you create new indexes to tune a statement that is currently parsed, then the Oracle Server invalidates the statement. When the statement is next executed, the optimizer automatically chooses a new execution plan that could potentially use the new index.

You should also keep in mind that the way you tune one statement can affect the optimizer's choice of execution plans for other statements. For example, if you create an index to be used by one statement, then the optimizer can choose to use that index for other statements in the application as well. For this reason, you should reexamine the application's performance and rerun the SQL trace facility after you have tuned the statements that you initially identified for tuning.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Indexes

An index is a database object that is logically and physically independent of the table data. The Oracle Server may use an index to access data that is required by a SQL statement, or it may use indexes to enforce integrity constraints. You can create and drop indexes at any time. The Oracle Server automatically maintains indexes when the related data changes.

### Index Types

Indexes can be unique or nonunique. Unique indexes guarantee that no two index entries have the same value. A composite index (also called a *concatenated index*) is an index that you create on multiple columns (up to 32) in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.

### Index Storage Techniques

For standard indexes, Oracle uses **B\*-tree indexes** that are balanced to equalize access times. B\*-tree indexes can be normal, reverse key, descending, or function based.

**Bitmap indexes** use a series of bitmaps to store information about which rows have specific values.

**Function-based indexes** are indexes based on frequently used functional operations on a column or columns.

**Bitmap-join indexes** are indexes built on frequently joined columns to facilitate joins.



## Indexes (continued)

**Domain indexes** are application-specific indexes. They are created, managed, and accessed by routines supplied by an index type. They are not covered in this course.

**Key compression** allows elimination of repeated occurrences of key-column prefixes in index-organized tables and indexes.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# When to Index

| Index   | Do Not Index  |
|---|---|
| Keys frequently used in search or query expressions | Keys and expressions with few distinct values except bitmap indexes in data warehousing |
| Keys used to join tables                            | Frequently updated columns  |
| High-selectivity keys                               | Columns used only with functions or expressions unless creating function-based indexes  |
| Foreign keys  | Columns based only on query performance   |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## When to Index

- Index keys that are used frequently in WHERE clauses.
- Index keys that are used frequently to join tables in SQL statements.
- Index keys that have high selectivity. The selectivity of an index is the percentage of rows in a table having the same value for the indexed key. An index's selectivity is optimal if few rows have the same value.
- Do not use standard B\*-tree indexes on keys or expressions with few distinct values. Such keys or expressions usually have poor selectivity and therefore do not optimize performance unless the frequently selected key values appear less frequently than the other key values. You can use bitmap indexes effectively in such cases, unless a high-concurrency OLTP application is involved in which the index is modified frequently.
- Do not index columns that are modified frequently. DML statements that modify take longer on indexed tables than if there were no index. Such SQL statements must modify data in indexes as well as data in tables.
- Do not index keys that appear only in WHERE clauses with functions or operators. A WHERE clause that uses a function other than MIN or MAX, or an operator with an indexed key, does not make available the access path that uses the index except with function-based indexes.

**Note:** Bitmap-indexes do not support row-level locking and must be used with caution in OLTP systems.

## When to Index (continued)

- Consider indexing foreign keys of referential integrity constraints in cases in which a large number of concurrent INSERT, UPDATE, and DELETE statements access the parent and child tables. Indexing foreign key columns helps avoid full table scans while searching for matching rows in the child when DML is performed on the parent. As an example, suppose that you want to delete a parent row or change its value. In this case, in order to delete the parent table rows, the child table must not have any matching keys. If there is an ON DELETE CASCADE clause or ON DELETE SET NULL, then the child rows affected by the parent DML must be located. Without an index, this would require a full table scan; if the child table is large, this can be expensive.
- When choosing to index a key, consider whether the performance gain for queries is worth the performance loss for INSERTs, UPDATEs, and DELETEs and the use of the space required to store the index. You might want to experiment by comparing the processing times of the SQL statements with and without indexes. You can measure processing time with the SQL Trace facility.
- Even after you create an index, the optimizer cannot use an access path that uses the index simply because the index exists. The optimizer can choose such an access path for a SQL statement only if it contains a construct that makes the access path available. To allow the query optimizer the option of using an index access path, ensure that the statement contains a construct that makes such an access path available.

## Effect of DML Operations on Indexes

- **Inserts result in the insertion of an index entry in the appropriate block (Block splits might occur.)**
- **Deletes 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Effect of DML Operations on Indexes

The Oracle Server maintains all the indexes when DML operations are carried out on the table. Here is an explanation of the effect of a DML command on an index:

- Insert operations result in the insertion of an index entry in the appropriate block. If there is no room for the entry in the block, then it is split into two blocks and an entry is added in the parent block to point to the new block.
- Deleting a row results in deletion of the index entry.
- Updates to the key columns result in a logical deletion and an insertion to the index. The PCTFREE setting has no effect on the index except at the time of creation. A new entry can be added to an index block even if it has less space than that specified by PCTFREE.

After periods of high DML activity, you should verify the index statistics and (if required) reorganize your B\*-tree indexes.

# Indexes and Constraints

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

- Primary key constraints
- Unique key constraints

```
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)
);
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Indexes and Constraints

When you define a primary or unique key constraint on a table, the Oracle Database automatically generates an index (or uses an existing index) to support it. Indexes that are created for this purpose are physically but not logically dependent on the table structure.

By default, the Oracle Server gives the index the same name as the constraint, but a different name for the index may be specified when creating the key.

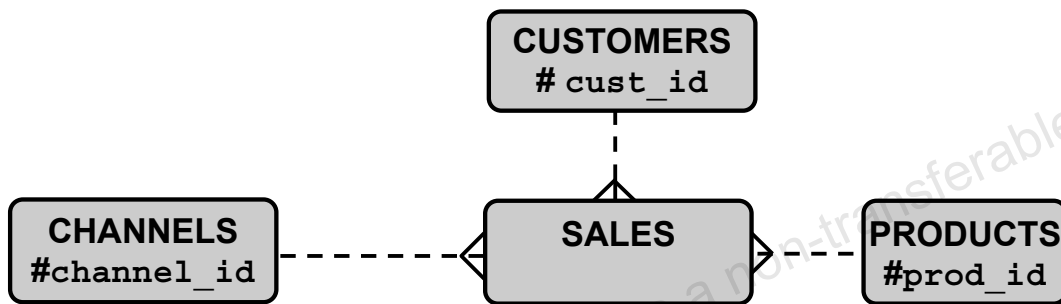
You cannot drop an index that enforces a constraint. Instead, you must disable the constraint, in most cases causing the Oracle Database to automatically drop the index.

Be aware that the Oracle Database supports deferrable constraints. Unique deferrable constraints are always enforced using nonunique indexes. Furthermore, when disabling such a constraint, the Oracle Database does not drop the associated index, thus offering an easy check when the constraint is enabled again.

Indexes serve at least two purposes: to make queries faster and to enforce unique and primary keys. The Oracle optimizer uses existing indexes when new constraints are defined. Note that the Oracle Database can use nonunique indexes to check unique constraints.

# Indexes and Foreign Keys

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



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Indexes and Foreign Keys

The Oracle Database does not automatically create an index for a foreign key constraint.

If the foreign key columns are often used in join conditions, then you should create an index on them to enhance the join process.

DML operations benefit from foreign key indexes as well. Deleting or updating parent rows requires that all matching child rows be located to make sure there are no dependents. Without an index on the foreign key column, this would cause a full scan of the child table.

# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Basic Access Methods

### Full Table Scans

A full table scan may involve reading many sequential data blocks from database files on disk into the buffer cache. However, the Oracle Server can read several sequential blocks in a single I/O. This is referred to as *multiblock I/O*. You can influence multiblock I/O by setting the `DB_FILE_MULTIBLOCK_READ_COUNT` parameter. Using the parallel query capability can help speed up full table scans by allocating several processes to the work.

### Index Scans

Indexes improve the performance of many SQL statements when the optimizer uses the index as an access path.

### Fast-Full Index Scans

The fast-full index scan is an alternative to a full table scan when there is an index that contains all the columns that are needed for a query. It is faster than a normal index scan because it can use multiblock I/O and can be parallelized in the same way as a table scan. Unlike regular index scans, however, the rows do not necessarily come back in sorted order. If there is a predicate that narrows the index search to a range of values, then the optimizer does not consider a fast-full index scan. For fast-full index scans to occur the index cannot have NULL values.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Identifying Unused Indexes

The Oracle Database provides the capability to gather statistics in the database about the usage of an index.

### Benefits

- If you find an index that is never used, you can drop the index and free the space it uses. However, indexes created on foreign keys speed up certain DML operations and should be retained even if they do not appear to be frequently used.
- Indexes must be maintained during most inserts and deletes and some updates. Eliminating an unused index eliminates this overhead.
- Parsing is simplified when the optimizer has fewer indexes to consider.



# Enabling and Disabling the Monitoring of Index Usage

- To start monitoring the usage of an index:

```
ALTER INDEX customers_pk MONITORING USAGE;
```

- To stop monitoring the usage of an index:

```
ALTER INDEX customers_pk NOMONITORING USAGE;
```

- V\$OBJECT\_USAGE contains information about the usage of an index.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Enabling and Disabling the Monitoring of Index Usage

The V\$OBJECT\_USAGE view displays information about the usage of an index. Each time the index is altered with the MONITORING USAGE clause, V\$OBJECT\_USAGE is reset for the specified index. Previous information is lost, and a new start time is recorded. The V\$OBJECT\_USAGE view supports the identification of unused indexes with the following columns:

- INDEX\_NAME: Index name
- TABLE\_NAME: Corresponding table
- MONITORING: Indicates whether monitoring is enabled or disabled
- USED: Indicates whether an index has been used during the monitoring time
- START\_MONITORING: The time that monitoring began on an index
- STOP\_MONITORING: The time that monitoring stopped on 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Index Tuning Using the SQL Access Advisor

The SQL Access Advisor is an alternative to manually determining which indexes are required. This advisor recommends a set of indexes when invoked from Advisor Central in Oracle Enterprise Manager or when run through the DBMS\_ADVISOR package APIs. The SQL Access Advisor either recommends using a workload or generates a hypothetical workload for a specified schema. Various workload sources are available, such as the current contents of the SQL cache, a user-defined set of SQL statements, or a SQL tuning set.

Given a workload, the SQL Access Advisor generates a set of recommendations from which you can select the indexes that are to be implemented. An implementation script is provided that can be executed manually or automatically through Oracle Enterprise Manager.

**Note:** The SQL Access Advisor was covered in the lesson titled “Automatic SQL Tuning.”

# 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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

This lesson introduced you to indexes.

- An index is a database object that is logically and physically independent of the table data.
- The Oracle Server may use an index to access data that is required by a SQL statement, or it may use indexes to enforce integrity constraints.
- There are different index types. For standard indexes, Oracle uses B\*-tree indexes that are balanced to equalize access times.
- Indexes can affect performance.
- There are different methods to scan a table: full table scans, index scans, and fast-full index scans. Index scans can improve the performance of many SQL statements.

swati bhatia (swati.june@gmail.com) has a non-transferable license to  
use this Student Guide.

# 12

## Using Different Indexes

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# 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**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Objectives

After completing this lesson, you should be able to:

- Create bitmap indexes, identify bitmap index operations, and use bitmap index hints
- Create and use function-based indexes
- View data dictionary information about indexes

# 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);
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Composite Indexes

A composite index contains more than one key column. Composite indexes can provide additional advantages over single-column indexes.

- **Improved selectivity:** Two or more columns or expressions, each with poor selectivity, can sometimes be combined to form a composite index with higher selectivity.
- **Reduced I/O:** If all columns selected by a query are in a composite index, then Oracle Database can return these values from the index without accessing the table.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Composite Index Guidelines

- Consider creating a composite index on keys that are used together frequently in WHERE clause conditions combined with AND operators, especially if their combined selectivity is better than the selectivity of either key individually.
- If several queries select the same set of keys based on one or more key values, then consider creating a composite index containing all of these keys.
- Consider the guidelines associated with the general performance advantages and trade-offs of indexes described in the previous slides.

## Ordering Keys for Composite Indexes

- Create the index so that the keys used in WHERE clauses make up a leading portion.
- If some keys are used in WHERE clauses more frequently, then be sure to create the index so that the more frequently selected keys make up a leading portion to allow the statements that use only these keys to use the index.
- If all keys are used in the WHERE clauses equally often but the data is physically ordered on one of the keys, then place that key first in the composite 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Skip Scanning of Indexes

The Oracle optimizer can use a composite index even if the prefix column value is not known. The optimizer uses an algorithm called *skip scanning* to retrieve row IDs for values that do not use the prefix column.

Skip scans reduce the need to add an index to support occasional queries that do not reference the prefix column of an existing index. Skip scanning can be useful when high levels of DML activity are degraded by the existence of too many indexes used to support infrequent queries.

The algorithm is also valuable in cases where there are no clear advantages as to which column to use as the prefix column in a composite index. The prefix column should be the most discriminating but also the most frequently referenced in queries. Two different columns in a composite index may meet these two requirements by forcing a compromise or the use of multiple 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.

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Bitmap Index

Bitmap indexing provides both substantial performance improvements and space savings over the usual (B\*-tree) indexes when there are few distinct values in the indexed columns.

A bitmap index may provide the best performance for queries based on columns with low cardinality, that is, columns that have few distinct values.

The maximum number of columns in a single bitmap index is 30.

The Oracle Server compresses bitmap storage, making bitmap indexes very storage efficient. The bitmaps are stored in a B\*-tree structure internally to maximize access performance.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## When to Use Bitmap Indexes

Use bitmap indexes in the following circumstances:

- The column has a low cardinality, meaning that there are few possible values for the column. A slightly higher cardinality would also be appropriate if the column is frequently used in complex conditions in the WHERE clauses of queries.
- The WHERE clauses of queries contain multiple predicates of these low- or medium-cardinality columns. Bitmap indexes are especially helpful for complex ad hoc queries with lengthy WHERE clauses or aggregate queries (containing SUM, COUNT, or other aggregate functions).
- The table has many rows. On a table with 1 million rows, even 10,000 distinct possible values would be acceptable.
- There are frequent (possibly ad hoc) queries on the table.
- The environment is oriented toward data warehousing (decision support systems).

Bitmap indexes are not ideal for OLTP environments because of their locking behavior. It is not possible to lock a single bitmap position. The smallest amount of a bitmap that can be locked is a bitmap segment, which can be up to half a data block in size. Changing the value of a row results in a bitmap segment becoming locked, in effect blocking changes on a number of rows. This is a serious disadvantage when there are many UPDATE, INSERT, or DELETE statements being issued by users. It is not a problem when data is loaded or updated in bulk actions, as in data warehousing environments.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Advantages of Bitmap Indexes

When used appropriately, bitmap indexes can provide substantial advantages:

- They provide reduced response time for many ad hoc queries. The optimizer is able to consider the use of bitmap indexes and decide when their use would be the most efficient way to execute a given query, incorporating parallelism if appropriate.
- They provide a substantial reduction of space usage compared to regular (B\*-tree) indexes. If only regular indexes are used, then the index designer must predict which columns will be used together and create a multicolumn (concatenated) index for these columns. This regular index would require a large amount of space and be sorted, making it useless for many queries that involve only a subset of the columns. The index designer could also create indexes on the other permutations of these columns; this would use a great deal of storage space. Bitmap indexes, however, can be created individually and then combined efficiently at run time. If the column to be indexed is a unique key column, then the space required for the bitmap index exceeds that of the B\*-tree index. Use bitmap indexes on low- or medium-cardinality columns only.
- You can obtain dramatic performance gains, even on low-end hardware. You do not need parallel processing capability to achieve performance gains with bitmap indexes. Bitmap indexes are created faster than B\*-tree indexes due to less sorting.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

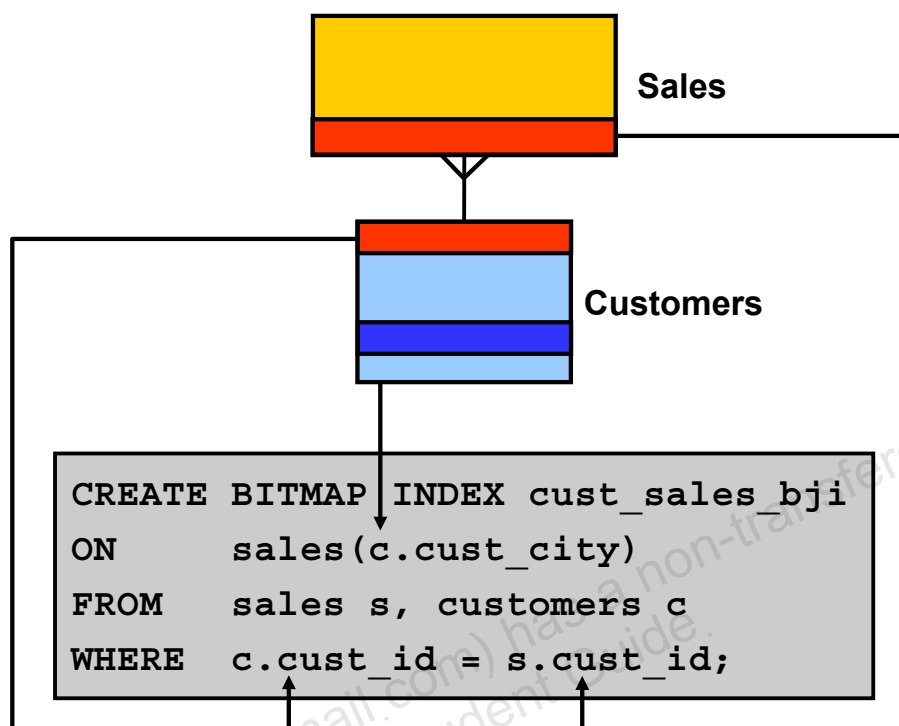
## Bitmap Index Guidelines

You can do several things to improve performance and storage of bitmap indexes:

- Declare NOT NULL constraints on all possible columns.
- Use fixed-length data types.
- Use the `ALTER TABLE ... MINIMIZE RECORDS_PER_BLOCK` command.

By using this command before creating a bitmap index, you can reduce the size of the bitmap segment. When this command is used, Oracle Database scans the table to see which block in the table has the most rows. It then uses that value to determine how many bits are used to represent each table block in the index. This results in a smaller bitmap index segment because there are fewer bits representing nonexistent rows.

# Bitmap Join Index



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Bitmap Join Index

In addition to a bitmap index on a single table, you can create a bitmap join index in the Oracle Database. A bitmap join index is a bitmap index for the join of two or more tables. A bitmap join index is a space-efficient way of reducing the volume of data that must be joined by performing restrictions in advance.

As you can see in the slide, the Oracle Database introduces a `CREATE BITMAP INDEX` syntax that you can use to specify a `FROM` and a `WHERE` clause. Here, you create a bitmap join index named `CUST_SALES_BJI` on the `SALES` table. The key of this index is the `CUST_CITY` column of the `CUSTOMERS` table.

This example assumes that there is an enforced primary key–foreign key relationship between `SALES` and `CUSTOMERS` to ensure that what is stored in the bitmap reflects the reality of the data in the tables. The `CUST_ID` column is the primary key of `CUSTOMERS` and also the foreign key from the `SALES` table to the `CUSTOMERS` table. The `FROM` and `WHERE` clauses in the `CREATE` statement enable the Oracle Database to make the link between the two tables. They represent the natural join condition between the two tables.

**Note:** This kind of index is not suitable for tables undergoing frequent updates due to locking of the index values.

## 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';
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Bitmap Join Index (continued)

The graphic for the previous slide shows you a theoretical implementation of this bitmap join index. Each entry or key in the index represents a possible city found in the CUSTOMERS table. A bitmap is then associated to one particular key. The meaning of the bitmap is the same representation as traditional bitmap indexes. Each bit in a bitmap corresponds to one row in the SALES table.

The interesting aspect of this structure becomes clear with the query in the slide. When the user tries to find the total cost of all sales for the Sully customers, the Oracle Database can use the bitmap join index and the SALES table to answer the question. In this case, there is no need to compute the join between the two tables explicitly. Using the bitmap join index here is much faster than computing the join at query time.

## 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Bitmap Join Index: Advantages and Disadvantages

A bitmap join index is an index on one table that involves columns of one or more different tables through a join.

The volume of data that must be joined can be reduced if bitmap join indexes are used as joins that have been precalculated. In addition, bitmap join indexes that can contain multiple dimension tables can eliminate bitwise operations that are necessary in the star transformation's use of bitmap indexes.

An alternative to a bitmap join index is a materialized join view, which is the materialization of a join in a table. Compared to a materialized join view, a bitmap join index is much more space efficient because it compresses ROWIDs of the fact tables.

In addition, queries using bitmap join indexes can be sped up by means of bitwise operations.

On the other hand, you may need to create more bitmap join indexes on the fact table to satisfy the maximum number of different queries. This means that you may have to create one bitmap join index for each column of the corresponding dimension tables. Of course, the result of having many indexes on one table is that maintenance costs are higher, especially when the fact table is updated.



## Bitmap Join Index: Advantages and Disadvantages (continued)

Because of the necessity of storing the results of joins, bitmap join indexes have the following restrictions:

- With the bitmap join index, only one table can be updated concurrently by different transactions.
- No table can appear twice in the join.
- You cannot create a bitmap join index on an index-organized table or on a temporary table.
- The columns in the index must all be columns of the dimension tables.
- The dimension-table join columns must either be primary key columns or have unique constraints.
- If a dimension table has a composite primary key, each column in the primary key must be part of the join.
- Bitmap join indexes cannot be built or rebuilt online.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

# 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';
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Function-Based Index

You can use the Oracle Database to create indexes based on column expressions (virtual columns). Function-based indexes provide an efficient mechanism for evaluating statements that contain expressions in their WHERE clauses. For example, you can use function-based indexes to create case-insensitive indexes, as the example in the slide shows.

You can create a function-based index to materialize computational-intensive expressions in the index, so that the Oracle Server does not need to compute the value of the expression when processing SQL statements.

Function-based index expressions can use any function that is deterministic; that is, the returned value must not change for a given set of input values. PL/SQL functions that are used in defining function-based indexes must be deterministic. The index owner needs the EXECUTE privilege on the defining function. If the EXECUTE privilege is revoked, then the function-based index is marked DISABLED. A function-based index can be created as a bitmap index.

### Enabling Function-Based Indexes

In order for function-based indexes to be used, query rewrite must be enabled. You can enable it at the session level as shown in the example or at the instance level by setting the initialization parameter QUERY\_REWRITE\_ENABLED to be TRUE;

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Data Dictionary Information

You can query the data dictionary to see specific information about function-based indexes:

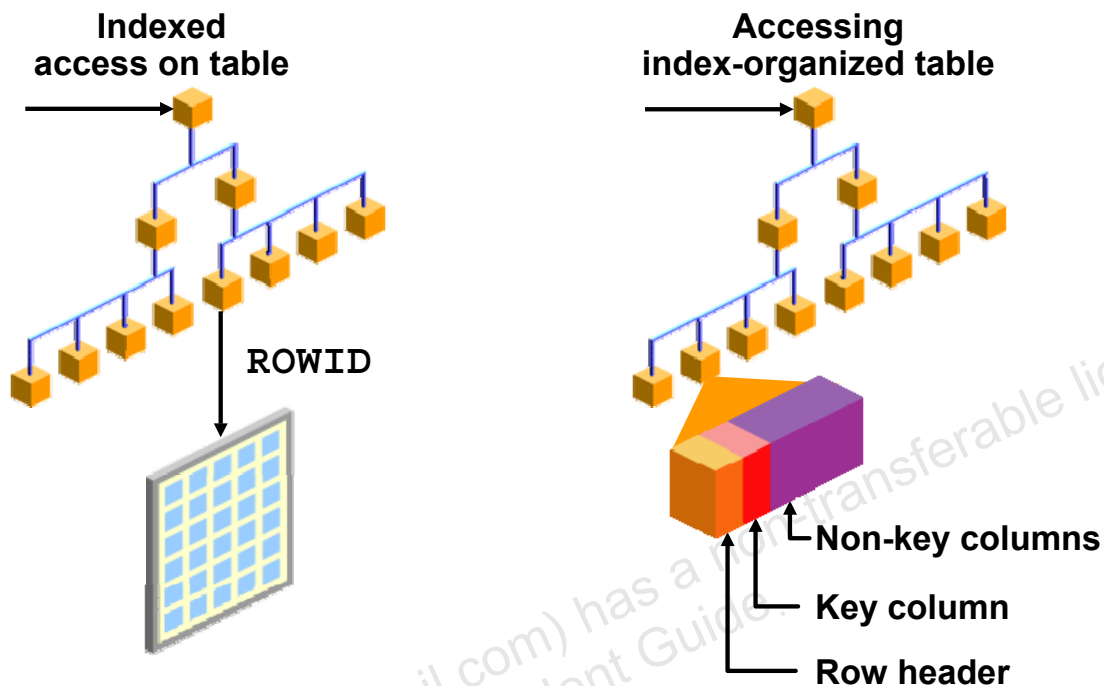
```
select table_name, index_name, index_type
from   user_indexes
where  index_name = 'FBI_UPPER_LASTNAME';
```

| TABLE_NAME | INDEX_NAME         | INDEX_TYPE            |
|------------|--------------------|-----------------------|
| CUSTOMERS  | FBI_UPPER_LASTNAME | FUNCTION-BASED NORMAL |

```
select column_expression
from   user_ind_expressions
where  index_name = 'FBI_UPPER_LASTNAME';
```

| COLUMN_EXPRESSION        |
|--------------------------|
| UPPER ("CUST_LAST_NAME") |

# Index-Organized Tables: Overview



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Index-Organized Tables: Overview

An index-organized table (IOT) is like a regular table with a concatenated index on all of its columns. However, the key values (for the table and the B\*-tree index) are stored in the same segment. An IOT contains:

- Primary key values
- Other (non-key) column values for the row

The B\*-tree structure, which is based on the primary key of the table, is organized like an index. The leaf blocks in this structure contain the rows instead of the row IDs. This means that the rows in the IOT are always maintained in the order of the primary key.

You can create additional indexes on IOTs. The primary key can be composite of many columns. Because large rows of an IOT can destroy the dense and efficient storage of the B\*-tree structure, you can store part of the row in another segment, which is called an *overflow area*. This is discussed in the following pages.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Index-Organized Tables: Characteristics

### Index-organized tables:

- Must have a primary key. This is the unique identifier and is used as the basis for ordering; there is no row ID to act as a unique identifier in the B\*-tree structure.
- Cannot include LONG columns but can contain LOB columns

IOTs are B\*-tree structures and are subject to fragmentation as a result of incremental updating. You can use the ALTER TABLE ... MOVE command to rebuild the IOT:

```
ALTER TABLE iot_tablename MOVE [OVERFLOW...];
```

Specifying the optional OVERFLOW clause causes the overflow segment to be rebuilt as well. Overflow segments are explained in the following slides.

You can access an IOT by using either the primary key or a combination of columns that constitute the leading part of the primary key. Because the rows are ordered in the IOT, full scans on an IOT return rows in a primary key sequence.

# 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
- **Disadvantage:**
  - Not suitable for queries that do not use the primary key in a predicate

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Benefits of Index-Organized Tables

Index-organized tables provide fast key-based access to table data for queries involving exact match and range searches. Changes to the table data result only in updating the index structure. Also, storage requirements are reduced because key columns are not duplicated in the table and index. The remaining non-key columns are stored in the index structure.

IOTS are particularly useful when you are using applications that must retrieve data based on a primary key. IOTs are also suitable for modeling application-specific index structures. For example, content-based information retrieval applications containing text, image, and audio data require inverted indexes that can be effectively modeled using IOTs.

No special considerations exist for using most SQL statements against an IOT. The structure of the table should be completely transparent to the user.

IOTs provide fast, key-based access for queries involving exact match (equality operator) or range searches on the primary key.

Because there is no duplication of primary key values, IOTs use less storage. Index organization is useful for a table that is frequently accessed using the primary key and has only a few, relatively short non-key columns.

# Summary

**In this lesson, you should have learned about:**

- **Composite indexes**
- **Bitmap indexes**
- **Bitmap join indexes**
- **Function-based indexes**
- **Index-organized tables**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

## Summary

This lesson introduced you to bitmap indexes, bitmap join indexes, composite indexes, and function-based indexes.

- Bitmap indexing provides both substantial performance improvements and space savings over usual (B\*-tree) indexes when there are few distinct values in the indexed columns.
- With function-based indexes, you can create indexes that are based on column expressions (virtual columns). Function-based indexes provide an efficient mechanism for evaluating statements that contain expressions in their WHERE clauses or for supporting linguistic sorting.
- You can use the data dictionary views to retrieve information about your indexes.

swati bhatia (swati.june@gmail.com) has a non-transferable license to  
use this Student Guide.



# 13

## Optimizer Hints

ORACLE

Copyright © 2007, Oracle. All rights reserved.

# Objectives

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

- **Optimizer mode**
- **Query transformation**
- **Access path**
- **Join orders**
- **Join methods**

**ORACLE**

Copyright © 2007, Oracle. All rights reserved.

# 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**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Optimizer Hints: Overview

Optimizer hints are used with SQL statements to alter execution plans. Hints enable you to influence decisions made by the optimizer. Hints provide a mechanism to direct the optimizer to choose a certain query execution plan based on the specific criteria.

For example, you might know that a certain index is more selective for certain queries. Based on this information, you might be able to choose a more efficient execution plan than the plan that the optimizer recommends. In such a case, use hints to force the optimizer to use the optimal execution plan.

# Types of Hints

|                    |                                     |
|--------------------|-------------------------------------|
| Single-table hints | Specified on one table or view      |
| Multitable hints   | Specify more than one table or view |
| Query block hints  | Operate on a single query block     |
| Statement hints    | Apply to the entire SQL statement   |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Types of Hints

**Single-table:** Single-table hints are specified on one table or view. INDEX and USE\_NL are examples of single-table hints.

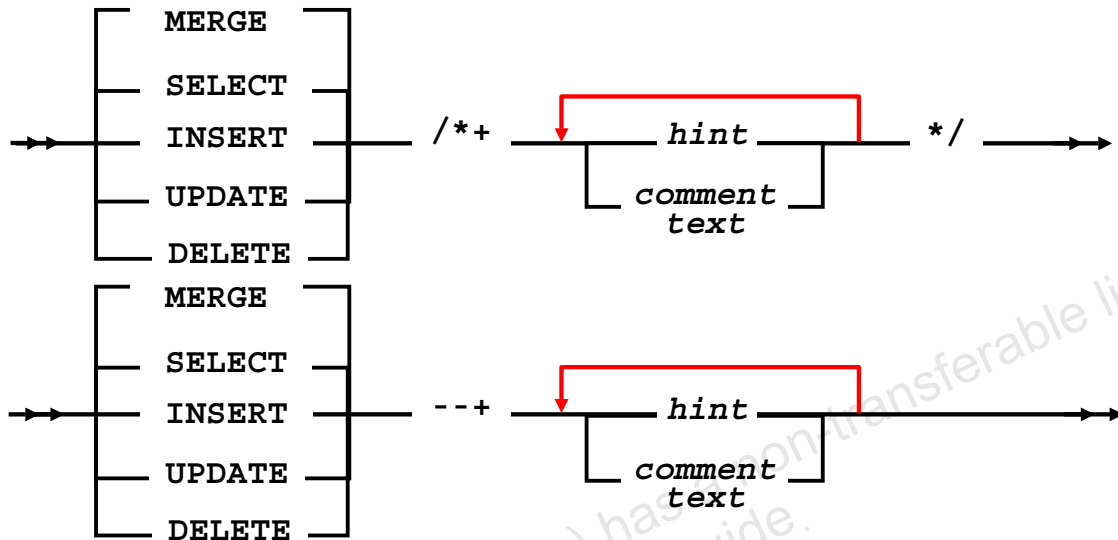
**Multitable:** Multitable hints are like single-table hints, except that the hint can specify one or more tables or views. LEADING is an example of a multitable hint.

**Note:** USE\_NL(table1 table2) is not considered a multitable hint because it is actually a shortcut for USE\_NL(table1) and USE\_NL(table2).

**Query block:** Query block hints operate on single query blocks. STAR\_TRANSFORMATION and UNNEST are examples of query block hints.

**Statement:** Statement hints apply to the entire SQL statement. ALL\_ROWS is an example of a statement hint.

# Specifying Hints



ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Specifying Hints

Hints apply to the optimization of only the block of the statement in which they appear. A statement block is:

- A simple MERGE, SELECT, INSERT, UPDATE, or DELETE statement
- A parent statement or a subquery of a complex statement
- A part of a compound query using set operators (UNION, MINUS, INTERSECT)

For example, a compound query consisting of two component queries combined by the UNION operator has two blocks, one for each component query. For this reason, hints in the first component query apply only to its optimization, not to the optimization of the second component query.

## Optimizer Hint Syntax

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

# Rules for Hints

- **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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Rules for Hints

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

The Oracle Server ignores incorrectly specified hints. However, be aware of the following situations:

- You never get an error message.
- Other (correctly) specified hints in the same comment are considered.
- The Oracle Server also ignores combinations of conflicting hints.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hint Recommendations

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

## Optimizer Hint Syntax: Example

```
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'
/
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Optimizer Hint Syntax: Example

The slide shows an example with a hint that advises the cost-based optimizer to use the index. The execution plan is the following:

Execution Plan

```
-----
0      UPDATE STATEMENT Optimizer=ALL_ROWS (Cost=3 ...)
1      0      UPDATE OF 'PRODUCTS'
2      1      TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS' (TABLE) (Cost...)
3      2      INDEX (RANGE SCAN) OF 'PRODUCTS_PROD_CAT_IX' (INDEX)
              (cost...)
4      1      TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS' (TABLE) (Cost...)
5      4      INDEX (UNIQUE SCAN) OF 'PRODUCTS_PK' (INDEX (UNIQUE))
              (Cost=0 ...)
```

The hint shown in the example works only if an index called PRODUCTS\_PROD\_CAT\_IX exists on the PRODUCTS table on the PROD\_CATEGORY column.



# Hint Categories

There are hints for:

- Optimization approaches and goals
- Access paths
- Query transformations
- Join orders
- Join operation
- Parallel execution

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hint Categories

Most of these hints are discussed in the following slides. Many of these hints accept the table and index names as arguments.

**Note:** The topic of hints for parallel execution is not covered in this course.

# Optimization Goals and Approaches

|                              |  |
|------------------------------|--|
| <b>ALL_ROWS</b>              | <b>Chooses cost-based approach with a goal of best throughput</b>                            |
| <b>FIRST_ROWS (<i>n</i>)</b> | <b>Instructs the Oracle Server to optimize an individual SQL statement for fast response</b> |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Optimization Goals and Approaches

**ALL\_ROWS:** The ALL\_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).

**FIRST\_ROWS (*n*):** The hints FIRST\_ROWS (*n*) (where *n* is any positive integer) and FIRST\_ROWS instruct the Oracle Server to optimize an individual SQL statement for fast response. FIRST\_ROWS (*n*) affords greater precision because it instructs the server to choose the plan that returns the first *n* rows most efficiently. The FIRST\_ROWS hint, which optimizes for the best plan to return the first single row, is retained for backward compatibility and plan stability.

If you specify either the ALL\_ROWS or the FIRST\_ROWS (*n*) hint in a SQL statement, and if the data dictionary does not have statistics about tables accessed by the statement, then the optimizer uses default statistical values, such as allocated storage for such tables, to estimate the missing statistics and to subsequently choose an execution plan. These estimates might not be as accurate as those gathered by the DBMS\_STATS package, so you should use the DBMS\_STATS package to gather statistics. If you specify hints for access paths or join operations along with either the ALL\_ROWS or FIRST\_ROWS (*n*) hint, then the optimizer gives precedence to the access paths and join operations specified by the hints.

## Hints for Access Paths

|                      |  |
|----------------------|--|
| <b>FULL</b>          | <b>Performs a full table scan</b>              |
| <b>ROWID</b>         | <b>Accesses a table by ROWID</b>               |
| <b>INDEX</b>         | <b>Scans an index in ascending order</b>       |
| <b>INDEX_ASC</b>     | <b>Scans an index in ascending order</b>       |
| <b>INDEX_COMBINE</b> | <b>Explicitly chooses a bitmap access path</b> |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Hints for Access Paths

Specifying one of these hints causes the optimizer to choose the specified access path only if the access path is available based on the existence of an index or on the syntactic constructs of the SQL statement. If a hint specifies an unavailable access path, then the optimizer ignores it.

You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, then use the alias rather than the table name in the hint. The table name in the hint should not include the schema name if the schema name is present in the statement.

#### **FULL**

The FULL hint explicitly chooses a full table scan for the specified table.

For example:

```
SELECT /*+ FULL(e) */ employee_id, last_name
FROM hr.employees e WHERE last_name LIKE 'K%';
```

The Oracle Server performs a full table scan on the employees table to execute this statement, even if there is an index on the last\_name column that is made available by the condition in the WHERE clause.

#### **ROWID**

The ROWID hint explicitly chooses a table scan by ROWID for the specified table.

## Hints for Access Paths (continued)

### INDEX

The INDEX hint explicitly chooses an index scan for the specified table. You can use the INDEX hint for domain, B\*-tree, bitmap, and bitmap join indexes. However, it is suggested that you use INDEX\_COMBINE rather than INDEX for bitmap indexes because it is a more versatile hint.

This hint can optionally specify one or more indexes.

If this hint specifies a single available index, then the optimizer performs a scan on this index. The optimizer does not consider a full table scan or a scan on another index on the table.

- If this hint specifies a list of available indexes, then the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The optimizer can also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan or a scan on an index not listed in the hint.
- If this hint specifies no indexes, then the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost. The optimizer can also choose to scan multiple indexes and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan.

### INDEX\_ASC

The INDEX\_ASC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then the Oracle Server scans the index entries in the ascending order of their indexed values.

Because the server's default behavior for a range scan is to scan index entries in the ascending order of their indexed values, this hint does not specify anything more than the INDEX hint. However, you might want to use the INDEX\_ASC hint to specify ascending range scans explicitly should the default behavior change.

### INDEX\_COMBINE

The INDEX\_COMBINE hint explicitly chooses a bitmap access path for the table. If no indexes are given as arguments for the INDEX\_COMBINE hint, then the optimizer uses whatever Boolean combination of bitmap indexes has the best cost estimate for the table. If certain indexes are given as arguments, then the optimizer tries to use some Boolean combination of those particular bitmap indexes.

For example:

```
SELECT /*+INDEX_COMBINE(customers cust_gender_bix cust_yob_bix)*/  
*  
FROM customers WHERE cust_year_of_birth < 70 AND cust_gender =  
'M';
```

# Hints for Access Paths

|                   |   |
|-------------------|---|
| <b>INDEX_JOIN</b> | <b>Instructs the optimizer to use an index join as an access path</b> |
| <b>INDEX_DESC</b> | <b>Chooses an index scan for the specified table</b>                  |
| <b>INDEX_FFS</b>  | <b>Performs a fast-full index scan</b>                                |
| <b>NO_INDEX</b>   | <b>Disallows using a set of indexes</b>                               |
| <b>AND_EQUAL</b>  | <b>Merges single-column indexes</b>                                   |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hints for Access Paths (continued)

### **INDEX\_JOIN**

The **INDEX\_JOIN** hint explicitly instructs the optimizer to use an index join as an access path. For the hint to have a positive effect, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query.

For example, the following query uses an index join to access the `employee_id` and `department_id` columns, both of which are indexed in the `employees` table:

```
SELECT /*+index_join(employees emp_emp_id_pk  
    emp_department_ix)*/ employee_id, department_id  
FROM hr.employees WHERE department_id > 50;
```

### **INDEX\_DESC**

The **INDEX\_DESC** hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then the Oracle Server scans the index entries in the descending order of their indexed values. In a partitioned index, the results are in the descending order within each partition.

For example:

```
SELECT /*+ INDEX_DESC(a ord_order_date_ix) */ a.order_date,  
    a.promotion_id, a.order_id  
FROM oe.orders a WHERE a.order_date < '01-jan-1985';
```

## Hints for Access Paths (continued)

### INDEX\_FFS

The `INDEX_FFS` hint causes a fast-full index scan to be performed rather than a full table scan.

For example:

```
SELECT /*+ INDEX_FFS ( o order_pk ) */ COUNT(*)
FROM order_items l, orders o
WHERE l.order_id > 50 AND l.order_id = o.order_id;
```

### NO\_INDEX

The `NO_INDEX` hint explicitly disallows a set of indexes for the specified table.

- If this hint specifies a single available index, then the optimizer does not consider a scan on this index. Other indexes that are not specified are still considered.
- If this hint specifies a list of available indexes, then the optimizer does not consider a scan on any of the specified indexes. Other indexes that are not specified in the list are still considered.
- If this hint specifies no indexes, then the optimizer does not consider a scan on any index on the table. This behavior is the same as a `NO_INDEX` hint that specifies a list of all available indexes for the table.

The `NO_INDEX` hint applies to function-based, B\*-tree, bitmap, or domain indexes. If a `NO_INDEX` hint and an index hint (`INDEX`, `INDEX_ASC`, `INDEX_DESC`, `INDEX_COMBINE`, or `INDEX_FFS`) both specify the same indexes, then both the `NO_INDEX` hint and the index hint are ignored for the specified indexes and the optimizer considers the specified indexes.

For example:

```
SELECT /*+NO_INDEX(employees emp_empid)*/ employee_id
FROM employees WHERE employee_id > 200;
AND_EQUAL
```

The `AND_EQUAL` hint explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes, where you can specify:

- The name or alias of the table that is associated with the indexes to be merged
- The indexes on which an index scan is to be performed. You must specify at least two indexes but no more than five.

## 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';
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### INDEX\_COMBINE Hint

The INDEX\_COMBINE hint is designed for bitmap index operations. Remember the following:

- If certain indexes are given as arguments for the hint, the optimizer tries to use some combination of those particular bitmap indexes.
- If no indexes are named in the hint, all indexes are considered to be hinted.
- The optimizer always tries to use hinted indexes, whether or not it considers them to be cost effective.

### INDEX\_COMBINE Hint: Example

Suppose that all three columns that are referenced in the WHERE predicate of the statement in the slide (CUST\_MARITAL\_STATUS, CUST\_GENDER, and CUST\_YEAR\_OF\_BIRTH) have a bitmap index. When you enable AUTOTRACE, the execution plan of the statement might appear as shown in the next slide.

## 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'
  
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### INDEX\_COMBINE Hint: Example (continued)

In the example in the slide, the following bitmap row sources are used:

| Bitmap Row Source            | Description  |
|------------------------------|--|
| BITMAP CONVERSION TO ROWIDS: | Converts bitmaps into ROWIDs to access a table<br>COUNT: Returns the number of entries if the actual values are not needed |
| BITMAP OR                    | Computes the bitwise OR of two bitmaps   |
| BITMAP AND                   | Computes the bitwise AND of two bitmaps  |
| BITMAP INDEX                 | SINGLE VALUE: Looks up the bitmap for a single key<br>RANGE SCAN: Retrieves bitmaps for a value range                      |
| BITMAP MERGE                 | Merges several bitmaps resulting from a range scan into one (using a bitwise AND operator)                                 |



# Hints for Query Transformation

|                   |  |
|-------------------|--|
| <b>USE_CONCAT</b> | <b>Rewrites OR into UNION ALL and disables INLIST processing</b> |
| <b>NO_EXPAND</b>  | <b>Prevents OR expansions</b>                                    |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hints for Query Transformation

### **USE\_CONCAT**

The **USE\_CONCAT** hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator. Generally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them.

The **USE\_CONCAT** hint disables IN-list processing and OR-expands all disjunctions, including IN-lists.

### **NO\_EXPAND**

The **NO\_EXPAND** hint prevents the cost-based optimizer from considering the OR-expansion for queries having OR conditions or IN-lists in the WHERE clause. Usually, the optimizer considers using the OR-expansion and uses this method if it decides that the cost is lower than not using it.

# Hints for Query Transformation

|                            |  |
|----------------------------|--|
| <b>MERGE</b>               | <b>Merges a view for each query</b>  |
| <b>NO_MERGE</b>            | <b>Prevents merging of mergeable views</b>   |
| <b>STAR_TRANSFORMATION</b> | <b>Makes the optimizer use the best plan in which the transformation can be used</b> |
| <b>FACT</b>                | <b>Indicates that the hinted table should be considered as a fact table</b>          |
| <b>NO_FACT</b>             | <b>Indicates that the hinted table should not be considered as a fact table</b>      |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hints for Query Transformation (continued)

### **MERGE**

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

If a view's query contains a GROUP BY clause or DISTINCT operator in the SELECT list, then the optimizer can merge the view's query into the accessing statement only if complex view merging is enabled. Complex merging can also be used to merge an IN subquery into the accessing statement if the subquery is uncorrelated.

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

### **NO\_MERGE**

The NO\_MERGE hint causes the Oracle Server not to merge mergeable views. This hint lets the user have more influence over the way in which the view is accessed.

## Hints for Query Transformation (continued)

When the `NO_MERGE` hint is used without an argument, it should be placed in the view query block. When `NO_MERGE` is used with the view name as an argument, it should be placed in the surrounding query.

### **STAR\_TRANSFORMATION**

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

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

### **FACT**

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

### **NO\_FACT**

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

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

# Hints for Join Orders

|                |   |
|----------------|---|
| <b>ORDERED</b> | <b>Causes the Oracle Server to join tables in the order in which they appear in the FROM clause</b> |
| <b>LEADING</b> | <b>Uses the specified table as the first table in the join order</b>                                |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hints for Join Orders

The following hints are used to suggest join orders.

### **ORDERED**

The **ORDERED** hint causes the Oracle Server to join tables in the order in which they appear in the **FROM** clause. If you omit the **ORDERED** hint from a SQL statement performing a join, then the optimizer chooses the order in which to join the tables. You might want to use the **ORDERED** hint to specify a join order if you know something that the optimizer does not know about the number of rows that are selected from each table. Such information lets you choose an inner and outer table better than the optimizer could.

If you analyze the tables, the optimizer usually selects an efficient star plan. You can also use hints to improve the plan. The most precise method is to order the tables in the **FROM** clause in the order of the keys in the index, with the large table last. Then use the following hints:

```
/*+ ORDERED USE_NL(FACTS) INDEX(facts fact_concat) */
```

Here, `facts` is the table and `fact_concat` is the index. A more general method is to use the **STAR** hint.

## Hints for Join Orders (continued)

### LEADING

The LEADING hint causes the Oracle Server to use the specified table as the first table in the join order.

If you specify two or more LEADING hints on different tables, all of the hints are ignored.

If you specify the ORDERED hint, it overrides all LEADING hints.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

## Hints for Join Operations

|                     |  |
|---------------------|--|
| <b>USE_NL</b>       | <b>Joins the specified table using a nested loop join</b>  |
| <b>NO_USE_NL</b>    | <b>Does not use nested loops to perform the join</b>       |
| <b>USE_MERGE</b>    | <b>Joins the specified table using a sort-merge join</b>   |
| <b>NO_USE_MERGE</b> | <b>Does not perform sort-merge operations for the join</b> |
| <b>USE_HASH</b>     | <b>Joins the specified table using a hash join</b>         |
| <b>NO_USE_HASH</b>  | <b>Does not use hash join</b>                              |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Hints for Join Operations

Each hint described in this section suggests a join operation for a table. In the hint, you must specify a table exactly the same way it appears in the statement. If the statement uses an alias for the table, then you must use the alias rather than the table name in the hint. However, the table name in the hint should *not* include the schema name if the schema name is present in the statement.

Use of the **USE\_NL** and **USE\_MERGE** hints is recommended with the **ORDERED** hint. The Oracle Server uses these hints when the referenced table is forced to be the inner table of a join; the hints are ignored if the referenced table is the outer table.

#### **USE\_NL**

The **USE\_NL** hint causes the Oracle Server to join each specified table to another row source with a nested loop join, using the specified table as the inner table. However, you might want to optimize the statement for best response time or for the minimal elapsed time that is necessary to return the first row selected by the query, rather than for best throughput. If so, then you can force the optimizer to choose a nested loop join by using the **USE\_NL** hint.

## Hints for Join Operations (continued)

### **NO\_USE\_NL**

The `NO_USE_NL` hint causes the optimizer to exclude the nested loops join. However, in some cases tables can only be joined using nested loops. In such cases, the optimizer ignores the hint for those tables.

In many cases, a nested loop join returns the first row faster than a sort-merge join does. A nested loop join can return the first row after reading the first selected row from one table and the first matching row from the other and combining them. But a sort-merge join cannot return the first row until after reading and sorting all selected rows of both tables and then combining the first rows of each sorted row source.

In the following statement in which a nested loop is forced through a hint, `orders` is accessed through a full table scan and the filter condition `l.order_id = h.order_id` is applied to every row. For every row that meets the filter condition, `order_items` is accessed through the index `order_id`.

```
SELECT /*+ USE_NL(l h) */ h.customer_id, l.unit_price *  
      l.quantity  
FROM   oe.orders h ,oe.order_items l  
WHERE  l.order_id = h.order_id;
```

Adding an `INDEX` hint to the query could avoid the full table scan on `orders`, resulting in an execution plan similar to one that is used on larger systems, even though it might not be particularly efficient here.

### **USE\_MERGE**

The `USE_MERGE` hint causes the Oracle Server to join each specified table with another row source by using a sort-merge join, as in the following example:

```
SELECT /*+USE_MERGE(employees departments)*/ * FROM employees,  
      departments WHERE employees.department_id =  
      departments.department_id;
```

### **NO\_USE\_MERGE**

The `NO_USE_MERGE` hint causes the optimizer to exclude the sort-merge join to join each specified table to another row source using the specified table as the inner table.

### **USE\_HASH**

The `USE_HASH` hint causes the Oracle Server to join each specified table with another row source using a hash join, as in the following example:

```
SELECT /*+USE_HASH(l l2) */ l.order_date, l.order_id,  
      l2.product_id, SUM(l2.unit_price*quantity)  
FROM   oe.orders l, oe.order_items l2  
WHERE  l.order_id = l2.order_id  
GROUP BY l2.product_id, l.order_date, l.order_id;
```

Here is another example:

```
SELECT /*+use_hash(employees departments)*/ *  
FROM   hr.employees, hr.departments  
WHERE  employees.department_id =  
      departments.department_id;
```

### **NO\_USE\_HASH**

The `NO_USE_HASH` hint causes the optimizer to exclude the hash join to join each specified table to another row source using the specified table as the inner table.

## Other Hints

|                             |   |
|-----------------------------|---|
| <b>APPEND</b>               | <b>Enables direct-path INSERT</b>   |
| <b>NOAPPEND</b>             | <b>Enables regular INSERT</b>   |
| <b>ORDERED_PREDICATES</b>   | <b>Forces the optimizer to preserve the order of predicate evaluation</b> |
| <b>CURSOR_SHARING_EXACT</b> | <b>Prevents replacing literals with bind variables</b>                    |
| <b>DYNAMIC_SAMPLING</b>     | <b>Controls dynamic sampling to improve server performance</b>            |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Other Hints

#### **APPEND**

The APPEND hint lets you enable direct-path INSERT if your database is running in serial mode. Your database is in serial mode if you are not using Enterprise Edition. Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode. In direct-path INSERT, data is appended to the end of the table rather than using existing space currently allocated to the table. As a result, direct-path INSERT can be considerably faster than conventional INSERT.

#### **NOAPPEND**

The NOAPPEND hint enables direct-path INSERT by disabling parallel mode for the duration of the INSERT statement. (Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode.)

#### **ORDERED\_PREDICATES**

The ORDERED\_PREDICATES hint forces the optimizer to preserve the order of predicate evaluation, except for predicates that are used as index keys. Use this hint in the WHERE clause of SELECT statements.



## Other Hints (continued)

If you do not use the `ORDERED_PREDICATES` hint, the Oracle Server evaluates all predicates in the following order:

1. Predicates without user-defined functions, type methods, or subqueries are evaluated first, in the order specified in the `WHERE` clause.
2. Predicates with user-defined functions and type methods that have user-computed costs are evaluated next, in the order of increasing cost.
3. Predicates with user-defined functions and type methods without user-computed costs are evaluated next, in the order specified in the `WHERE` clause.
4. Predicates that are not specified in the `WHERE` clause (for example, predicates that are transitively generated by the optimizer) are evaluated next.
5. Predicates with subqueries are evaluated last, in the order specified in the `WHERE` clause.

### **CURSOR\_SHARING\_EXACT**

The Oracle Server can replace literals in SQL statements with bind variables if it is safe to do so. This is controlled with the `CURSOR_SHARING` startup parameter. The `CURSOR_SHARING_EXACT` hint causes this behavior to be disabled. In other words, the Oracle Server executes the SQL statement without any attempt to replace literals with bind variables.

### **DYNAMIC\_SAMPLING**

The `DYNAMIC_SAMPLING` hint lets you control dynamic sampling to improve server performance by determining more accurate selectivity and cardinality estimates. You can set the value of `DYNAMIC_SAMPLING` to a value from 0 to 10. The higher the level, the more effort the compiler puts into dynamic sampling and the more broadly it is applied. Sampling defaults to cursor level unless you specify a table.

Consider the following example:

```
SELECT /*+ dynamic_sampling(1) */ * FROM ...
```

This example enables dynamic sampling if all of the following conditions are true:

- There is more than one table in the query.
- At least one table has not been analyzed and has no indexes.
- The optimizer determines that a relatively expensive table scan is required for the table that has not been analyzed.

## Hints for Suppressing Index Usage

| Hint                              | Description  |
|-----------------------------------|--|
| <b>NO_INDEX</b>                   | <b>Disallows use of any indexes</b>  |
| <b>FULL</b>                       | <b>Forces a full table scan</b>  |
| <b>INDEX or<br/>INDEX_COMBINE</b> | <b>Forces the optimizer to use a specific index or a set of listed indexes</b> |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Hints for Suppressing Index Usage

In some cases, you might want to prevent a SQL statement from using an access path that uses an existing index. You might want to do this if you know that the index is not very selective and that a full table scan would be more efficient. If the statement contains a construct that makes such an index access path available, then you can force the optimizer to use a full table scan by using one of the following methods:

- Use the **NO\_INDEX** hint to give the query optimizer maximum flexibility while disallowing the use of a certain index.
- Use the **FULL** hint to force the optimizer to choose a full table scan instead of an index scan.
- Use the **INDEX** or **INDEX\_COMBINE** hints to force the optimizer to use one index or a set of listed indexes instead of another.

# 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.**

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Hints and Views

You should not use hints in or on views because views can be defined in one context and used in another; such hints can result in unexpected plans. In particular, hints in views are handled differently from hints on views depending on whether or not the view is mergeable into the top-level query.

### View Optimization

To optimize a statement that accesses a view, the optimizer has two alternatives. The statement is normally transformed into an equivalent statement that accesses the view's base tables. The optimizer can use one of the following techniques to transform the statement:

- Merge the view's query into the referencing query block in the accessing statement.
- Push the predicate of the referencing query block inside the view.

When these transformations are impossible, the view's query is executed and the result is accessed as if it were a table. This appears as a VIEW step in execution plans.

## Hints and Views (continued)

### Mergeable Views

The optimizer can merge a view into a referencing query block if the view definition does not contain the following:

- Set operators (UNION, UNION ALL, INTERSECT, MINUS)
- CONNECT BY clause
- ROWNUM pseudocolumn
- Group functions (AVG, COUNT, MAX, MIN, SUM) in the select list

### Hints and Mergeable Views

Optimization-approach and goal hints can occur in a top-level query or in views:

- If there is such a hint in the top-level query, that hint is used regardless of any other such hints in the views.
- If there is no top-level optimizer-mode hint, then mode hints in referenced views are used as long as all mode hints in the views are consistent.
- If two or more mode hints in the referenced views conflict, then all mode hints in the views are discarded and the session mode is used, whether default or user specified.

Access-method and join hints on referenced views are ignored unless the view contains a single table (or references another view with a single table). For such single-table views, an access-method hint or a join hint on the view applies to the table in the view.

Access-method and join hints can also appear in a view definition:

- If the view is a subquery (that is, if it appears in the FROM clause of a SELECT statement), then all access-method and join hints in the view are preserved when the view is merged with the top-level query.
- For views that are not subqueries, access-method and join hints in the view are preserved only if the top-level query references no other tables or views (that is, if the FROM clause of the SELECT statement contains only the view).

### Hints and Nonmergeable Views

With nonmergeable views, optimizer-mode hints in the view are ignored. The top-level query decides the optimization mode.

Because nonmergeable views are optimized separately from the top-level query, access-method and join hints in the view are always preserved. For the same reason, access-method hints on the view in the top-level query are ignored.

However, join hints on the view in the top-level query are preserved because (in this case) a nonmergeable view is similar to a table.

## Hints for View Processing

|                 |  |
|-----------------|--|
| <b>MERGE</b>    | <b>Merges complex views or subqueries with the surrounding query</b> |
| <b>NO_MERGE</b> | <b>Does not merge mergeable views</b>                                |

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Hints for View Processing

You can cause a view to be merged on a per-query basis by using the MERGE and NO\_MERGE hints. The NO\_MERGE hint causes the Oracle Server not to merge mergeable views.

#### Example

```
SELECT /*+ MERGE(v) */
      p.prod_id, p.prod_name, v.sold
FROM   products p,
      (select s.prod_id, sum(amount_sold) as sold
       from sales s
       group by s.prod_id) v
WHERE  p.prod_id = v.prod_id
AND    p.prod_status = 'obsolete';
```

**Note:** Because the group function is used on line 6, the MERGE (v) hint is ignored.

# 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

```
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;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Extended Hint Syntax

Hints that specify a table generally refer to tables in the DELETE, SELECT, or UPDATE query block in which the hint occurs, rather than to tables inside any views that are referenced by the statement. When you want to specify hints for tables that appear inside views, Oracle Corporation recommends using global hints instead of embedding the hint in the view.

The table hints that are described in this lesson can be transformed into global hints by using an extended tablespec syntax that includes view names with the table name. In addition, an optional query block name can precede the tablespec syntax:

`view_name.table_name`

If a global hint references a table name or alias that is used twice in the same query (for example, in a UNION statement), the hint applies to only the first instance of the table (or alias).

# Specifying a Query Block in a Hint

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

1

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

```
SELECT /*+ QB_NAME(qb) FULL(@qb e) */
       employee_id, last_name
FROM hr.employees e
WHERE employee_id = 100;
```

2

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Specifying a Query Block in a Hint

To identify a query block within a query, an optional query block name can be used in a hint to specify the query block to which the hint applies. The syntax of the query block argument is of the form @queryblock, where queryblock is an identifier that specifies a query block in the query. The queryblock identifier can either be system generated or user specified. The system-generated identifier can be obtained by using EXPLAIN PLAN for the query (as shown in example 1). The output from the EXPLAIN PLAN statement is:

PLAN\_TABLE\_OUTPUT

-----Plan hash  
value: 2219862869

|           |                             |             |       |      | Id     |     |
|-----------|-----------------------------|-------------|-------|------|--------|-----|
| Operation | Name                        | Rows        | Bytes | Cost | (%CPU) |     |
| 0         | SELECT STATEMENT            |             | 1     | 12   | 2      | (0) |
| 1         | TABLE ACCESS BY INDEX ROWID | EMPLOYEES   | 1     | 12   | 2      | (0) |
| * 2       | INDEX RANGE SCAN            | EMP_NAME_IX | 1     |      | 1      | (0) |

Query Block Name / Object Alias (identified by operation id):

- 1 - SEL\$1 / E@SEL\$1
- 2 - SEL\$1 / E@SEL\$1

## Specifying a Query Block in a Hint (continued)

...

Predicate Information (identified by operation id):

-----

2 - access("LAST\_NAME"='Smith')

Column Projection Information (identified by operation id):

-----

1 - "EMPLOYEE\_ID" [NUMBER,22], "LAST\_NAME" [VARCHAR2,25]

2 - "E".ROWID [ROWID,10], "LAST\_NAME" [VARCHAR2,25]

The query block name can then be used in a hint, as follows:

```
SELECT /*+ FULL(@SEL$1 e) */
        employee_id, last_name
FROM employees e
WHERE last_name = 'Smith';
```

This statement results in a full table scan on the EMP\_NAME\_IX index being used.

2. The user-specified name can be set with the QB\_NAME hint (as shown in example 2).



## 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;
```

ORACLE

Copyright © 2007, Oracle. All rights reserved.

### Specifying a Full Set of Hints

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

In the example, the `LEADING` hint specifies the exact join order to be used. The join methods to be used on the different tables are also specified.

# Summary

**In this lesson, you should have learned how to:**

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

ORACLE

Copyright © 2007, Oracle. All rights reserved.

## Summary

In this lesson, you should have learned about additional optimizer settings and hints.

By using hints, you can influence the optimizer at the statement level. Use hints as a last remedy when tuning SQL statements. There are several hint categories, one of which includes hints for access-path methods.

To specify a hint, use the hint syntax in the SQL statement.

# 14

## Materialized Views

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

# 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**

**ORACLE**

Copyright © Oracle Corporation, 2007. All rights reserved.

# 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**

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Materialized Views

A materialized view stores both the definition of a view and the rows resulting from the execution of the view. Like a view, it uses a query as the basis. However, the query is executed at the time the view is created, and the results are stored in a table. You can define the table with the same storage parameters that any other table has, and you can place it in the tablespace of your choice. You can also index the materialized view table in the same way that you index other tables to improve the performance of queries executed against them.

When a query can be satisfied with data in a materialized view, the Oracle Server transforms the query to reference the view rather than the base tables. By using a materialized view, expensive operations such as joins and aggregations do not need to be reexecuted.

**Note:** The query is not always executed when a materialized view is created. It depends on the `BUILD` clause. If `BUILD IMMEDIATE` is coded, the data for the query is built when the view is created. If `BUILD DEFERRED` is coded, the query data is built at first refresh. The `ON PREBUILT TABLE` clause lets you register an existing table as a preinitialized materialized view. The table must have the same name and be in the same schema as the resulting materialized view. If the materialized view is dropped, then the preexisting table reverts to its identity as a table.

## If Materialized Views Are Not Used

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

```
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;
```

```
SELECT * FROM cust_sales_sum;
```

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### If Materialized Views Are Not Used

Some organizations that use summaries spend a significant amount of time creating them manually, identifying which ones to create, indexing them, updating them, and advising their users about which ones to use.

For example, to enhance performance for the SQL query of the application in the slide, you can create a summary table called CUST\_SALES\_SUM and inform users of its existence. In turn, users use this summary table instead of the original query.

Obviously, the time required to execute the SQL query from the preceding summary table is minimal compared to the time required for the original SQL query.

On the other hand, users must be aware of summary tables and need to rewrite their applications to use those tables.

Also, you must manually refresh the summary tables to keep them up-to-date with the corresponding original tables.

It can very quickly become difficult to maintain such a system.

# Benefits of Using Materialized Views

```
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 ...)
```

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Benefits of Using Materialized Views

With materialized views, end users no longer must be aware of the summaries that have been defined. You can create one or more materialized views that are the equivalent of summary tables. The advantage provided by creating a materialized view instead of a table is that a materialized view not only materializes the result of a query into a database table, but also generates metadata information used by the query rewrite engine to automatically rewrite the SQL query to use the summary tables. Furthermore, a materialized view optionally offers another important possibility: refreshing data automatically.

The slide shows that using materialized views is transparent to the user. If your application must execute the same SQL query as in the previous slide, all you need to do is create the materialized view called `cust_sales_mv`. Then, whenever the application executes the SQL query, the Oracle Server automatically rewrites it to use the materialized view instead.

Compared to the time that is required by the summary table approach, the query response time is the same. The primary difference is that the application need not be rewritten. The rewrite phase is automatically handled by the system. In addition, the SQL statement that defines the materialized view does not need to match the SQL statement of the query itself.

## How Many Materialized Views?

- **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.

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### How Many Materialized Views?

The query optimizer automatically recognizes when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries go directly to the materialized view and not to the underlying detail tables. In general, rewriting queries to use materialized views rather than detail tables improves queries response time.

You thus need to decide which materialized views to create. One materialized view can be used by the optimizer to rewrite any number of suitable queries. When using query rewrite, you should therefore create materialized views that satisfy the largest number of queries.

You can also create nested materialized views. A nested materialized view is a materialized view whose definition is based on another materialized view. A nested materialized view can reference other relations in the database in addition to referencing materialized views. By using nested materialized views, you can create multiple single-table materialized views based on a joins-only materialized view; the join is performed just once. In addition, optimizations can be performed for this class of single-table aggregate materialized view, and thus refresh is very efficient.



## 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 ...
```

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### How Many Materialized Views (continued)

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

### Creating Materialized Views: Syntax Options

The `CREATE MATERIALIZED VIEW` syntax is similar to the `CREATE SNAPSHOT` command, which it replaces. There are some additional options:

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

## Creating Materialized Views: Example

```
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.
```

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### Creating Materialized Views: Example

If a query involving aggregates, large or multiple joins, or both aggregates and joins is likely to be used multiple times, it can be more efficient to create a materialized view of the query results. This requires a single execution of the query and the storage space to preserve the results.

If the queries are likely to be reused over time, you may also need a mechanism to update the materialized view as the base tables change. The performance and storage costs of maintaining the materialized view must be compared to the costs of reexecuting the original query whenever it is needed. This is discussed in more detail later in this lesson.

When the optimizer chooses to use the view rather than the base table, this event is called a *query rewrite*. This is also discussed in more detail later in this lesson.

**Note:** This example is a `CREATE MATERIALIZED VIEW` command. Details about optional command clauses are discussed later in this lesson.

**Note:** On the previous page, the default options in the syntax are underlined. This is not the complete syntax; see *Oracle Database 10g SQL Reference* for more details.

# Types of Materialized Views

- **Materialized views with aggregates**

```
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**

```
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;
```

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Types of Materialized Views

The SELECT clause in the materialized view creation statement defines the data that the materialized view is to contain. Only a few restrictions limit what can be specified. Any number of tables can be joined together. However, they cannot be remote tables if you want to take advantage of query rewrite. In addition to tables, other elements such as views, inline views (subqueries in the FROM clause of a SELECT statement), subqueries in the WHERE clause, and materialized views can all be joined or referenced in the SELECT clause.

In data warehouses, materialized views normally contain aggregates. This is the origin of the term *summaries* in data warehouses. Some materialized views contain only joins and no aggregates. The advantage of creating this type of materialized view is that expensive joins are precalculated.

**Note:** The CREATE statements in the slide must include the ENABLE QUERY REWRITE clause. If they do not, then rewrite does not occur, and these materialized views are treated as replication materialized views.

# Refresh Methods

- 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.

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Refresh Methods

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

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

## Refresh Methods (continued)

Remember the following when deciding whether to use nested materialized views:

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

# Refresh Modes

- **Manual refresh**
  - Specify the **ON DEMAND** option
  - By using the **DBMS\_MVIEW** package
- **Automatic refresh Synchronous**
  - Specify the **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**

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Refresh Modes

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

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

## Refresh Modes (continued)

As with refresh types, refresh modes are not all available for every materialized view. *Oracle Database 10g Supplied PL/SQL Packages and Types Reference* describes the modes and the circumstances for the use of each mode.

For more information about materialized views, see the *Oracle Database 10g Data Warehousing Guide*.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

## Manual Refresh with DBMS\_MVIEW

- **For ON DEMAND refresh only**
- **Three procedures with the DBMS\_MVIEW package:**
  - REFRESH
  - REFRESH\_ALL\_MVIEWS
  - REFRESH\_DEPENDENT

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### Manual Refresh with DBMS\_MVIEW

You can control the time at which refresh of the materialized views occurs by specifying ON DEMAND refreshes. In this case, the materialized view can be refreshed only by calling one of the procedures in the DBMS\_MVIEW package. DBMS\_MVIEW provides three different types of refresh operations:

- DBMS\_MVIEW.REFRESH : Refresh one or more materialized views
- DBMS\_MVIEW.REFRESH\_ALL\_MVIEWS : Refresh all materialized views
- DBMS\_MVIEW.REFRESH\_DEPENDENT : Refresh all table-based materialized views that depend on a specified detail table or list of detail tables



# Materialized Views: Manual Refresh

## Specific materialized views:

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

## Materialized views based on one or more tables:

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

## All materialized views due for refresh:

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

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Materialized Views: Manual Refresh

Possible refresh scenarios for materialized views include the following:

- Use the REFRESH procedure to refresh specific materialized views.
- Use the REFRESH\_DEPENDENT procedure to refresh all materialized views that depend on a given set of base tables.
- Use the REFRESH\_ALL\_MVIEWS procedure to refresh all materialized views that have not been refreshed since the last bulk load to one or more detail tables. This accepts an out parameter that is used to return the number of failures.

The procedures in the package use several parameters to specify the following:

- Refresh method
- Whether to proceed if an error is encountered
- Whether to use a single transaction (consistent refresh)
- Rollback segment to use

# Query Rewrites

- 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.

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Query Rewrites

Because accessing a materialized view may be significantly faster than accessing the underlying base tables, the optimizer rewrites a query to access the view when the query allows it. The query rewrite activity is transparent to applications. In this respect, their use is similar to the use of an index.

Users do not need explicit privileges on materialized views to use them. Queries that are executed by any user with privileges on the underlying tables can be rewritten to access the materialized view.

A materialized view can be enabled or disabled. A materialized view that is enabled is available for query rewrites, as in the following example:

```
ALTER MATERIALIZED VIEW cust_sales_mv DISABLE QUERY REWRITE;
```

# Query Rewrites

- **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.

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Query Rewrites (continued)

The best way to determine whether query rewrites occur is to use the EXPLAIN PLAN command or the AUTOTRACE setting in SQL\*Plus. The execution plan shows whether a rewrite has taken place that uses the materialized view. You should also notice improved response time if a materialized view is used by the optimizer. You can also use the DBMS\_MVIEW.EXPLAIN\_REWRITE procedure to verify whether rewrites can occur.

**Note:** There are several system privileges that control whether you are allowed to create materialized views and modify them, and whether query rewrites are enabled for you. Contact your local database administrator to set this up properly.

There are also many data dictionary views that contain information about materialized views. For details, see *Oracle Database 10g Performance Guide and Reference* and *Oracle Database 10g Reference*.

# Enabling and Controlling Query Rewrites

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

```
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**

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Enabling and Controlling Query Rewrites

**OPTIMIZER\_MODE:** Query rewrites are available with cost-based optimization only.

**QUERY\_REWRITE\_ENABLED:** This is a dynamic instance/session parameter that can be set to the following values:

- **TRUE:** Cost-based rewrite
- **FALSE:** No rewrite
- **FORCE:** Forced rewrite

**QUERY\_REWRITE\_INTEGRITY:** This is also a dynamic instance/session parameter.

It accepts the following values:

- **ENFORCED** (default) enables query rewrites only if the optimizer can guarantee consistency. Only fresh materialized views and enabled validated constraints are used for query rewrites.
- **TRUSTED** allows query rewrites based on declared (not necessarily enforced) relationships. All fresh materialized views and constraints with the **RELY** flag are used for query rewrites.
- **STALE\_TOLERATED** allows stale materialized views that do not contain the latest data to be used.

**Note:** If query rewrite does not occur, try **STALE\_TOLERATED** mode first. If the optimizer does not rewrite in this mode, it will never rewrite.

## Enabling and Controlling Query Rewrites (continued)

**Note:** There are no object privileges that are associated with query rewrites. Users with access to the base tables implicitly benefit from query rewrites.

You can use the `REWRITE` hint to restrict the materialized views that are considered for query rewrites. The `NOREWRITE` hint is available to suppress query rewrites.

swati bhatia (swati.june@gmail.com) has a non-transferable license to use this Student Guide.

## 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...)
```

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### Query Rewrite: Example

The execution plan shows that the materialized view is accessed instead of joining all four base tables to produce the result.


A REWRITE or NOREWRITE hint overrides a materialized view's definition that is set in the CREATE or ALTER MATERIALIZED VIEW command with the QUERY REWRITE clause.

This example shows a transparent query rewrite in which the query exactly matches the materialized view definition. The next slide shows an example of a query that does not match the materialized view definition.

## Query Rewrite: Example

```
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;
```



ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### Query Rewrite: Example (continued)

The optimizer can use the materialized view created earlier to satisfy the query.

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

## Verifying Query Rewrite

```
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

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### Verifying Query Rewrite

The example in the slide shows a situation in which creating a materialized view is not useful. Using the `REWRITE_OR_ERROR` hint in a query causes the following error if the query fails to rewrite:

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

You can then decide if such a materialized view is worth retaining, based on testing with other queries.



# SQL Access Advisor

**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

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## SQL Access Advisor

The SQL Access Advisor (SAA) helps you achieve your performance goals by recommending the proper set of materialized views, materialized view logs, and indexes for a given workload. Understanding and using these structures is essential when optimizing SQL, because they can result in significant performance improvements in data retrieval. The advantages, however, come with a cost. Creation and maintenance of these objects can be time consuming, and space requirements can be significant. The SAA also recommends how to optimize materialized views so that they can be fast refreshable and take advantage of general query rewrite.

The SAA can be run from Oracle Enterprise Manager by using the SQL Access Advisor Wizard or by invoking the `DBMS_ADVISOR` package. The `DBMS_ADVISOR` package consists of a collection of analysis and advisory functions and procedures that can be called from any PL/SQL program. If a workload is not provided, the SAA can generate and use a hypothetical workload also.

# Using the DBMS\_MVIEW Package

## DBMS\_MVIEW methods

- **EXPLAIN\_MVIEW**
- **EXPLAIN\_REWRITE**
- **TUNE\_MVIEW**

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Using the DBMS\_MVIEW Package

Several DBMS\_MVIEW procedures help you with materialized view fast refresh and query rewrite.

The EXPLAIN\_MVIEW procedure tells you whether a materialized view is fast refreshable or eligible for general query rewrite. EXPLAIN\_REWRITE tells you whether query rewrite will occur. However, neither tells you how to achieve fast refresh or query rewrite.

To further facilitate the use of materialized views, the TUNE\_MVIEW procedure shows how to optimize your CREATE MATERIALIZED VIEW statement and how to meet other requirements (such as materialized view log for fast refresh and general query rewrite).

# Tuning Materialized Views for Fast Refresh and Query Rewrite

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

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Tuning Materialized Views for Fast Refresh and Query Rewrite

The `DBMS_MVIEW.TUNE_MVIEW` method analyzes and processes the input statement and generates two sets of output results: one for the materialized view implementation and the other for undoing the create materialized view operations. The two sets of output results can be accessed through Oracle views or can be stored in external script files created by the SAA. These external script files are ready to execute to implement the materialized view.

The `TUNE_MVIEW` procedure takes two input parameters: `task_name` and `mv_create_stmt`. `Task_name` is a user-provided task identifier used to access the output results. `mv_create_stmt` is a complete `CREATE MATERIALIZED VIEW` statement that is to be tuned. If the input `CREATE MATERIALIZED VIEW` statement does not have the clauses `REFRESH FAST` or `ENABLE QUERY REWRITE`, or both, `TUNE_MVIEW` will use the default clauses `REFRESH FORCE` and `DISABLE QUERY REWRITE` to tune the statement to be fast refreshable (if possible) or only complete refreshable.

## Results of Tune\_MVIEW

- **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**

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### Results of Tune\_MVIEW

When the `ENABLE QUERY REWRITE` clause is specified, `TUNE_MVIEW` also fixes the statement by using a process similar to `REFRESH FAST`, which redefines the materialized view so that many of the advanced forms of query rewrite are possible.

The `TUNE_MVIEW` procedure generates two sets of output results as executable statements. One set of output results, `IMPLEMENTATION`, is for implementing materialized views and required components such as materialized view logs to achieve fast refreshability and query rewritability. The other set of output results, `UNDO`, is for dropping the materialized views (if you decide that they are not required).

The output statements for the `IMPLEMENTATION` process include:

- `CREATE MATERIALIZED VIEW LOG` statements, which create any missing materialized view logs required for fast refresh
- `ALTER MATERIALIZED VIEW LOG FORCE` statements, which fix any materialized view log–related requirements (such as missing filter columns, sequence, and so on) that are required for fast refresh

## Results of TUNE\_MVIEW (continued)

- One or more CREATE MATERIALIZED VIEW statements: If there is one output statement, the original defining query is directly restated and transformed. Simple query transformation could be simply adding required columns. For example, add a row ID column for a materialized join view and add an aggregate column for a materialized aggregate view. With decomposition, multiple CREATE MATERIALIZED VIEW statements are generated and form a nested materialized view hierarchy in which one or more submaterialized views are referenced by a new top-level materialized view modified from the original statement. This achieves as much fast refresh and query rewrite as possible. Submaterialized views are often fast refreshable.

Note that the decomposition result implies no sharing of submaterialized views. That is, with decomposition, the TUNE\_MVIEW output will always contain new submaterialized view and will not reference existing materialized views.

The output statements for the UNDO process include DROP MATERIALIZED VIEW statements to reverse the materialized view creations (including submaterialized views) in the IMPLEMENTATION process.

## Viewing TUNE\_MVIEW Output Results

### Using Dictionary Views

After TUNE\_MVIEW executes, the results are output into the SAA repository tables and are accessible through the Oracle views USER\_TUNE\_MVIEW and DBA\_TUNE\_MVIEW.

### Script Generation DBMS\_ADVISOR Function and Procedure

It is straightforward to have the SAA generate scripts using the DBMS\_ADVISOR.GET\_TASK\_SCRIPT function. The following is a simple example.

First, define a directory in which to store the results:

```
CREATE DIRECTORY TUNE_RESULTS AS '/tmp/script_dir';
GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;
```

Then generate both the implementation and undo scripts and place them in /tmp/script\_dir/mv\_create.sql and /tmp/script\_dir/mv\_undo.sql, respectively, as follows:

```
EXECUTE
  DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name), - 'TUNE_RESULTS', 'mv_create.sql');
EXECUTE
  DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name), - 'UNDO'), 'TUNE_RESULTS', 'mv_undo.sql');
```

## 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**
- **utlxmlv.sql must be executed as the current user to create MV\_CAPABILITIES\_TABLE.**

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### DBMS\_MVIEW.EXPLAIN\_MVIEW Procedure

The purpose of the Explain Materialized View procedure is to advise what is and is not possible with a given materialized view or potential materialized view. This package advises the user by providing answers to the following questions:

- Is this materialized view fast refreshable?
- What are the types of query rewrite that can be done with this materialized view?

The process for using this package is very simple. The DBMS\_MVIEW.EXPLAIN\_MVIEW procedure is called, passing in as parameters the schema and materialized view name for an existing materialized view. Alternatively, you can specify the select string for a potential materialized view. The materialized view or potential materialized view is then analyzed, and the results are written either to a table called MV\_CAPABILITIES\_TABLE (the default) or to a VARRAY of type ExplainMVArrayType called MSG\_ARRAY.

**Note:** Except when using VARRAYS, you must run the utlxmlv.sql script prior to calling EXPLAIN\_MVIEW. The script creates the MV\_CAPABILITIES\_TABLE in the current schema. The script is found in the admin directory.

## Explain Materialized View: Example

```
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;
```

| CAPABILITY_NAME  | P | RELATED_TE | MSGTXT   |
|------------------|---|------------|--|
| ...              |   |            |  |
| REFRESH_COMPLETE | Y |            |  |
| REFRESH_FAST     | N |            |  |
| REWRITE          | N |            |  |
| PCT_TABLE        | N | SALES      | no partition key or<br>PMARKER in select<br>list |
| PCT_TABLE        | N | CUSTOMERS  | relation is not a<br>partitioned<br>table        |
| ...              |   |            |  |

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

### Explain Materialized View: Example

In the example in the slide, suppose that the MV\_CAPABILITIES\_TABLE has already been created. You want to analyze an already existing materialized view called CUST\_SALES\_MV that was created in the SH schema. You need to assign an ID for this analysis so that we can retrieve it subsequently in the MV\_CAPABILITIES\_TABLE. You also need to use the SEQ column in an ORDER BY clause so that the rows appear in a logical order.

If a capability is not possible, N appears in the P column and an explanation appears in the MSGTXT column. If a capability is not possible for more than one reason, a row is displayed for each reason.

# Designing for Query Rewrite

## Query rewrite considerations:

- **Constraints**
- **Outer joins**
- **Text match**
- **Aggregates**
- **Grouping conditions**
- **Expression matching**
- **Date folding**
- **Statistics**

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Designing for Query Rewrite

**Constraints:** Make sure that all inner joins that are referred to in a materialized view have referential integrity (foreign key and primary key constraints) with additional NOT NULL constraints on the foreign key columns. Because constraints tend to impose a large overhead, you could make them NO VALIDATE and RELY and set the parameter QUERY\_REWRITE\_INTEGRITY to STALE\_TOLERATED or TRUSTED. However, if you set QUERY\_REWRITE\_INTEGRITY to ENFORCED, all constraints must be enabled, enforced, and validated to obtain maximum rewritability. You should avoid using the ON DELETE clause because it can lead to unexpected results.

**Outer joins:** Another way of avoiding constraints is to use outer joins in the materialized view. Query rewrite will be able to derive an inner join in the query, such as (A.a = B.b), from an outer join in the materialized view (A.a = B.b(+)) as long as the row ID of B or column B.b is available in the materialized view. Most of the support for rewrites with outer joins is provided for materialized views with joins only. To exploit it, a materialized view with outer joins should store the row ID or primary key of the inner table of an outer join.



## Designing for Query Rewrite (continued)

**Text match:** If you need to speed up an extremely complex, long-running query, you can create a materialized view with the exact text of the query. Then the materialized view would contain the query results, thus eliminating the time that is required to perform any complex joins and search through all the data for what is required.

**Aggregates:** To get the maximum benefit from query rewrite, make sure that all aggregates that are computed in the targeted set of queries are present in the materialized view. The conditions for aggregates are quite similar to those for incremental refresh. For instance, if `AVG (x)` is in the query, then you should store `COUNT (x)` and `AVG (x)` or store `SUM (x)` and `COUNT (x)` in the materialized view.

**Grouping conditions:** Aggregating data at lower levels in the hierarchy is better than aggregating at higher levels because lower levels can be used to rewrite more queries. For example, instead of grouping on the state, you can group on the city. Also, instead of creating multiple materialized views with overlapping or hierarchically related `GROUP BY` columns, you can create a single materialized view with all those `GROUP BY` columns. For example, instead of using a materialized view that groups by city and another materialized view that groups by month, use a single materialized view that groups by both city and month.

**Expression matching:** If several queries share the same common subselect, it is advantageous to create a materialized view with the common subselect as one of its `SELECT` columns. In this way, the performance benefit due to precomputation of the common subselect can be obtained across several queries.

**Date folding:** When creating a materialized view that aggregates data by folded date granules (such as months, quarters, or years), always use the year component as the prefix but not as the suffix. For example, `TO_CHAR (date_col, 'yyyy-q')` folds the date into quarters, which collate in year order, whereas `TO_CHAR (date_col, 'q-yyyy')` folds the date into quarters, which collate in quarter order. The former preserves the ordering while the latter does not. For this reason, any materialized view created without a year prefix will not be eligible for date-folding rewrite.

**Statistics:** Optimization with materialized views is based on cost. To make a cost-based choice, the optimizer needs statistics on both the materialized view and the tables in the query. Materialized views should thus have statistics collected using the `DBMS_STATS` package.

# Materialized View Hints

|                         |   |
|-------------------------|---|
| <b>REWRITE</b>          | <b>Rewrites a query in terms of materialized views</b>    |
| <b>REWRITE_OR_ERROR</b> | <b>Forces an error if a query rewrite is not possible</b> |
| <b>NO_REWRITE</b>       | <b>Disables query rewrite for the query block</b>         |

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Materialized View Hints

REWRITE, NOREWRITE, and REWRITE\_OR\_ERROR are hints that are used with materialized views to control query rewrites

### **REWRITE**

The REWRITE hint forces the cost-based optimizer to rewrite a query in terms of materialized views (when possible) without cost consideration. Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, then the Oracle Server uses that view regardless of its cost.

The server does not consider views outside of the list. If you do not specify a view list, then the server searches for an eligible materialized view and always uses it regardless of its cost.

### **REWRITE\_OR\_ERROR**

This hint forces an error if a query rewrite is not possible.

### **NOREWRITE**

The NOREWRITE hint disables query rewrite for the query block, overriding the setting of the QUERY\_REWRITE\_ENABLED parameter. Use the NOREWRITE hint on any query block of a request.

# Summary

**In this lesson, you should have learned how to:**

- **Create materialized views**
- **Enable query rewrites using materialized views**

ORACLE

Copyright © Oracle Corporation, 2007. All rights reserved.

## Summary

This lesson introduced you to materialized views and query rewrites. A materialized view stores both the definition of a view and the rows resulting from the execution of the view. Like a view, it uses a query as its basis; however, the query is executed at the time the view is created and the results are stored in a table.

Because accessing a materialized view can be significantly faster than accessing the underlying base tables, the optimizer rewrites a query to access the view when the query allows it. The query rewrite activity is transparent to applications.

swati bhatia (swati.june@gmail.com) has a non-transferable license to  
use this Student Guide.