
Oracle9i New Features for Application Developers

Student Guide • Volume 1

30083GC10
Production 1.0
June 2001
D33085

ORACLE®

Authors

Priya Vennapusa
Craig Hollister

Technical Contributors and Reviewers

Brian Boxx
Taj ul Islam
Dan Gabel
Laslo Czinkocski
Anna Atkinson
Sarah Jones
Helen Robertson
Sander Rekveld
Craig Davis
Stefan Lindberg
Jasmine Robayo
Thomas Hoogerwerf
Martin Jensen
Bruce Ernst
Matt Taylor
Jim Womack
Susan Kotsovlos
Geeta Aurora
Sundeep Abraham
Barry Trute
Sandeepan Bannerjee
Mark Scardina
Diana Lorentz
Ulrike Schwinn
Peter Sharman

Publisher

Sheryl Domingue

Copyright © Oracle Corporation, 2000, 2001. All rights reserved.

This documentation contains proprietary information of Oracle Corporation. It is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited. If this documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions for commercial computer software and shall be deemed to be Restricted Rights software under Federal law, as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

This material or any portion of it may not be copied in any form or by any means without the express prior written permission of Oracle Corporation. Any other copying is a violation of copyright law and may result in civil and/or criminal penalties.

If this documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights," as defined in FAR 52.227-14, Rights in Data-General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them in writing to Education Products, Oracle Corporation, 500 Oracle Parkway, Box SB-6, Redwood Shores, CA 94065. Oracle Corporation does not warrant that this document is error-free.

Oracle and all references to Oracle products are trademarks or registered trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Contents

Preface

1 Introduction to Oracle9i New Features for Application Developers

Objectives 1-2

Enhancements in Oracle9i: Focus Areas 1-3

Oracle9i SQL New Features 1-4

Oracle9i PL/SQL New Features 1-5

Oracle9i Java New Features 1-6

Oracle9i XML New Features 1-7

Oracle9i OCI New Features 1-8

Oracle9i Objects New Features 1-9

Oracle9i Availability New Features 1-10

Oracle9i Performance New Features 1-11

Oracle9i Security New Features 1-12

Oracle9i Business Intelligence Enhancements 1-13

Oracle9i Globalization New Features 1-14

Other Oracle9i Database Functionality 1-15

Course Prerequisites 1-16

Course Objectives 1-17

Agenda 1-18

Summary 1-20

2 Data Integrity Enhancements

Objectives 2-2

Overview 2-3

Overview of Explicit Defaults 2-4

Explicit Default 2-5

Overview of Constraint Enhancements 2-7

No Share Lock on Unindexed Foreign Keys 2-8

Primary Key Lookup During Foreign Key Creation 2-9

Example of Creating Primary Key Constraint 2-10

Example of Dropping Indexes 2-11

Constraints on Views 2-12

Overview of FOR UPDATE WAIT Clause 2-13

Example of Using FOR UPDATE WAIT Clause 2-14

Index-Only Scans on Function-Based Indexes 2-15

Summary 2-16

Practice 2 Overview 2-17

Practice 2 Overview 2-18

3 New DML Features for Data Warehousing

Objectives 3-2

Overview 3-3

External Tables 3-4

Example of Defining External Tables 3-5

Querying External Tables 3-6

Multitable INSERT Statements 3-7

Advantages of Multitable INSERTs 3-8

Types of Multitable INSERT Statements 3-9

Example of Unconditional INSERT 3-10

Example of Pivoting INSERT ALL 3-11

Example of Conditional INSERT ALL 3-13

Example of Conditional INSERT FIRST 3-14

MERGE statements 3-15

Example of Using the MERGE Statement in Data Warehousing 3-16

Summary 3-17

Practice 3 Overview 3-18

4 Business Intelligence Enhancements

Objectives 4-2

Overview of Grouping Sets 4-3

Overview of Grouping Sets 4-4

GROUPING SETS 4-5

Grouping Sets Example 4-6

Grouping Sets Example: Results 4-7

GROUPING SETS Versus CUBE and ROLLUP Statements 4-8

Composite Columns 4-9

Concatenated Groupings 4-11

The WITH Clause: Usage Notes 4-18

The WITH Clause: Benefits 4-19

Analytical Function Enhancements 4-20

Example of Inverse Percentile Functions 4-22

What-if Rank and Distribution Functions 4-23

RANK and DENSE_RANK: Example 4-24

FIRST and LAST Aggregate Values 4-25

The WIDTH_BUCKET Function 4-27

The WIDTH_BUCKET: Example 4-28

Summary 4-30

Practice 4 Overview 4-31

5 ANSI/ISO SQL: 1999 Standard Support in Oracle9i

Objectives 5-2

Overview 5-3

Applications for SQL: 1999 Features 5-4

SQL: 1999 Joins 5-5

Types of SQL: 1999 Compliant Joins 5-6

Creating CROSS Joins 5-7

Creating NATURAL Joins 5-8

Retrieving Records with NATURAL Joins 5-9

Creating Joins with the USING Clause 5-10

Retrieving Records with the USING Clause 5-11

Creating Joins With the ON Clause 5-12

Example of Retrieving Records with the ON Clause 5-13

Creating Complex Joins 5-14

Join Predicates and the ON Clause 5-15

Creating Multitable Joins 5-16

INNER Versus OUTER Joins 5-17

Example of LEFT OUTER Joins 5-18

Example of RIGHT OUTER Joins 5-19

Example of FULL OUTER Joins 5-20

Example of Using Subqueries 5-21

Example of Join Using NOT EXISTS 5-22

CASE Expressions in SQL: 1999 5-23

Simple CASE Expression 5-24

Simple CASE Expression: Example 5-25

Searched CASE Expression	5-26
Searched CASE Expression: Example	5-27
SQL NULLIF and COALESCE	5-28
NULLIF and COALESCE	5-29
SQL NULLIF Expression: Example	5-30
SQL COALESCE Expression: Example	5-31
COALESCE with Multiple Expressions: Example	5-32
Scalar Subqueries in SQL: 1999	5-33
Using Scalar Subqueries	5-34
Scalar Subqueries in SELECT List	5-35
Scalar Subqueries in the WHERE Clause	5-36
Scalar Subqueries in the ORDER BY Clause	5-37
Scalar Subqueries in CASE Expressions	5-38
Scalar Subqueries in Functions	5-39
Summary	5-40
Practice 5 Overview	5-41

6 Datetime Enhancements

Objectives	6-2
TIME_ZONE Session Parameter	6-3
Datetime and Interval Data types	6-4
Datetime Data Types	6-5
Datetime Fields	6-6
TIMESTAMP Data Type	6-7
Difference between DATE and TIMESTAMP	6-8

TIMESTAMP WITH TIME ZONE Data Type	6-9
TIMESTAMP WITH TIME ZONE: Example	6-10
TIMESTAMP WITH LOCAL TIMEZONE	6-12
TIMESTAMP WITH LOCAL TIME ZONE: Example	6-13
INTERVAL Data Types	6-14
INTERVAL Fields	6-15
INTERVAL YEAR TO MONTH Data Type	6-16
INTERVAL YEAR TO MONTH: Example	6-17
INTERVAL YEAR TO MONTH Data Type: Example	6-18
INTERVAL DAY TO SECOND Data Type	6-19
INTERVAL DAY TO SECOND Data Type: Example	6-20
Daylight Savings Time Boundaries	6-21
Datetime Functions	6-22
Datetime Conversion Functions	6-23
Datetime EXTRACT Function	6-24
CURRENT_DATE, CURRENT_TIMESTAMP and LOCALTIMESTAMP	6-25
CURRENT_DATE: Example	6-26
CURRENT_DATE: Example	6-27
DBTIMEZONE: Example	6-28
Using FROM_TZ: Example	6-29
Using TO_DSINTERVAL: Example	6-30
TO_TIMESTAMP: Example	6-31
TZ_OFFSET: Example	6-32
Using EXTRACT: Example	6-33
Summary	6-34
Practice 6 Overview	6-35

7 Migrating LONGs to LOBs

Objectives 7-2

Overview 7-3

Oracle9i LONG to LOB Migration 7-4

LONG to LOB Migration: Example 7-5

Restrictions on LOB Migration 7-7

SQL Support for LOB Migration 7-9

Using SQL Functions on LOBs: Example 7-10

PL/SQL Support for LOB Migration 7-11

Implicit Assignment and Parameter Passing to LOBs 7-12

Implicit Conversion of LOBs: Example 7-13

Support for LOB Migration in OCI 7-14

Other LOB Enhancements 7-15

Summary 7-16

Practice 7 Overview 7-17

8 Object and Collection Type Enhancements

Objectives	8-2
Overview	8-3
Type Inheritance	8-4
Type Hierarchy	8-6
Type Declarations	8-7
FINAL and NOT FINAL Types	8-8
NOT INSTANTIABLE Types	8-9
Substitutability	8-10
Attribute Inheritance	8-11
Method Inheritance	8-12
Method Override	8-13
Dynamic Method Dispatch	8-14
Rights Model	8-15
Object and REF Assignment	8-16
Widening: Example	8-17
Narrowing by Using TREAT: Example	8-18
Inheritance Support in Data Dictionary	8-19
Object Type Evolution	8-20
Type Dependencies	8-21
Propagating Type Changes	8-22
Propagating Nonstructural Changes	8-23
Propagating Structural Changes	8-24
Altering the Attributes of an Object Type	8-25
Table Validation	8-26

ALTER TYPE Statement Options	8-27
Propagating Changes to Dependent Tables	8-28
Multilevel Collection Types	8-29
Applications of Multilevel Collections	8-30
Creating Multilevel Collections	8-31
Nested Tables in Multilevel Collections	8-32
Varrays in Multilevel Collections	8-33
Creating Tables with Multilevel Collections	8-34
Collection Unnesting	8-35
SYS.AnyType	8-36
SYS.AnyData	8-37
SYS.AnyDataSet	8-38
Summary	8-39
Practice 8 Overview	8-40

9 SQL Support in PL/SQL

Objectives	9-2
Overview	9-3
PL/SQL CASE Statement	9-4
Types of CASE	9-5
Simple Versus Searched CASE	9-6
Simple CASE Expression: Example	9-7
Simple CASE Statement: Example	9-8
Searched CASE Expression: Example	9-9
Searched CASE Statement: Example	9-10

NULLIF and COALESCE Expressions	9-11
Using NULLIF in PL/SQL	9-12
Using COALESCE in PL/SQL	9-13
Overview of Cursor Subquery	9-14
Cursor Subquery in Ref Cursor: Example	9-15
Cursor Subquery in Explicit Cursor: Example	9-16
Cursor Subquery : Nested Cursor Attributes	9-17
Introducing the Common SQL Parser	9-18
Visible Effects of Common SQL Parser	9-19
Other PL/SQL Language Enhancements	9-20
Transparent Performance Enhancements	9-21
Summary	9-22
Practice 9 Overview	9-23

10 Performance Enhancements in PL/SQL

Objectives	10-2
Overview of Native Compilation of PL/SQL	10-3
New Parameters for Native Compilation	10-4
Steps for Enabling Native Compilation	10-6
Performance	10-8
Benefits of Native Compilation of PL/SQL	10-9
Restrictions to Native Compilation of PL/SQL	10-11
Overview of Oracle9i Bulk Bind Enhancements	10-12
FORALL Statement Enhancements	10-13
Error Handling for Bulk Binds	10-14

Example of Exception Handling	10-15
Benefits of Bulk Bind Enhancement	10-17
Overview of Bulk Dynamic SQL	10-18
Bulk Bind with Dynamic SQL: Example	10-19
Bulk Bind for Input Variables: Example	10-20
Bulk Binding In Output Variables: Example	10-21
Benefits of Bulk Dynamic SQL	10-22
Overview of Table Functions	10-23
PL/SQL Table Functions	10-24
Example of Creating Table Functions	10-25
Using Table Functions	10-26
Advantages of PL/SQL Table Functions	10-27
Transparent Performance Enhancements	10-28
Summary	10-29
Practice 10 Overview	10-30

11 Globalization Support

Objectives	11-2
Overview of Unicode	11-3
Unicode Encoding	11-4
Overview of Oracle9i Unicode Support	11-5
Unicode Storage	11-7
Unicode Solutions: Database	11-8
Unicode Solutions: Data Type	11-9
Choosing a Unicode Solution: Unicode Database	11-10

Choosing a Unicode Solution: Unicode Data Type	11-11
NCHAR Interoperability	11-12
Exception Handling for Data Loss	11-15
Byte and Character Semantics for CHAR and VARCHAR2	11-16
Character Semantics	11-17
Character Semantics Support in Oracle9i	11-18
Length Semantics	11-19
Multilingual Linguistic Sorts	11-20
Binary and Multilingual Sort	11-21
SQL Collation Functions	11-23
NLSSORT Function: Example	11-25
Programming Interfaces	11-27
Summary	11-28

12 Performance and Availability Enhancements

Objectives	12-2
Overview of Performance Enhancements	12-3
Index-Organized Table Enhancements	12-4
Mapping Tables	12-5
Creating a Mapping Table	12-6
What Is a Bitmap Join Index?	12-7
Advantages and Disadvantages of Bitmap Join Indexes	12-9
Skip Scanning of Indexes	12-11
Skip Scanning: Example	12-13
Skip Scanning: Search for Switzerland	12-14

Identifying Unused Indexes	12-18
Enabling and Disabling the Monitoring of Index Use	12-19
New First Rows Optimization	12-20
New Statistics-Gathering Estimations	12-21
Optimizer Cost Model Enhancements	12-23
Gathering System Statistics	12-25
Gathering System Statistics: Example	12-26
Safe and Unsafe Cursor Sharing	12-28
Cursor Sharing	12-30
Overview of Outline Editing	12-31
Editable Attributes	12-33
Outline Cloning	12-34
Outline: Administration and Security	12-35
Configuration Parameters	12-37
Create Outline Syntax Changes	12-38
Overview of Availability Enhancements	12-39
Online Index Rebuild	12-40
The Quiesce Database Feature	12-41
Benefits of the Quiesce Database Feature	12-42
Resumable Space Allocation	12-43
Enabling and Disabling Resumable Space Allocation Mode	12-44
Automatic Undo Management	12-45
Metadata API: Advantages	12-46
Metadata Unload in Oracle9i	12-47
Summary	12-48
Practice 12 Overview	12-49

13 Data Definition and Data Protection Features

Objectives	13-2
Online Table Redefinition	13-3
Limitations to Online Table Redefinition	13-4
Online Table Redefinition: Example	13-7
Oracle9i Stored Data Encryption	13-9
Selective Data Encryption	13-11
Oracle8i Virtual Private Database (VPD)	13-12
Oracle9i Virtual Private Database Features	13-13
Oracle Policy Manager	13-14
Partitioned Fine-Grained Access Control	13-15
Fine-Grained Auditing	13-18
Fine-Grained Auditing Concepts	13-20
Fine-Grained Auditing	13-21
Triggering Audit Events	13-22
Fine-Grained Auditing	13-23
Secure Application Role	13-24
Secure Application Role Implementation	13-25
Secure Application Role Creation	13-26
Secure Application Role Benefits	13-27
Global Application Context	13-28
Summary	13-31

14 Database Workspace Manager

Objectives	14-2
Overview	14-3

Database Workspaces	14-4
How Does Workspace Manager Work?	14-5
Example	14-6
Workspace Manager Benefits and Applications	14-7
Workspace Manager Benefits	14-8
Workspace Manager Concepts	14-9
Version-Enable a Table	14-11
Guidelines for Tables Participating in a Workspace	14-12
Disabling Workspace Participation for a Table	14-14
Creating a Workspace	14-15
Associating a User Session with a Workspace	14-16
Granting Privileges	14-17
Setting Locks	14-18
Create Workspace Savepoint	14-20
Implicit and Explicit Savepoints	14-21
Compare Savepoints and Find Differences	14-22
Deleting a Savepoint	14-24
Freezing a Workspace	14-25
Rollback a Workspace	14-26
Refresh a Workspace	14-27
Resolve Workspace Conflicts	14-28
Check for Existence of Conflicts	14-29
Resolve Conflicts	14-30
Merge a Workspace	14-31
Workspace Views	14-32
Summary	14-33
Practice 14 Overview	14-35

15 Support for Web-Based Applications

Objectives 15-2

Introduction to XML 15-3

A Simple XML Document 15-4

XML Components 15-5

What Is XSL? 15-7

Sample Code 15-8

Sample XSL Style Sheet: address.xsl 15-9

HTML Output 15-10

Overview of XDKs in Oracle9i 15-11

What are the Components of Oracle9i XDK? 15-12

XML Development Kit Components 15-13

XML Schema Processors 15-14

Oracle9i XDK for JavaBeans 15-15

XML Parsers 15-16

XSL Processors 15-17

XML Class Generators 15-18

XML SQL Utility 15-19

XSQL Servlet 15-20

XSQL File Example 15-22

What Are the New Database XML Features? 15-23

Overview of XMLType 15-24

XMLType 15-25

Using XMLType 15-27

Example Using XMLType 15-28

XMLType Storage Characteristics	15-29
XMLType Functions	15-30
Example Using XMLType	15-31
Indexing XMLTypes	15-32
Benefits of XMLType	15-33
Native XML Generation	15-34
DBMS_XMLGEN	15-35
DBMS_XMLGEN: Example	15-36
Generated XML	15-37
Generating Complex XML	15-38
SYS_XMLGEN	15-41
SYS_XMLAGG	15-42
URI References	15-43
The New URI Reference Data Types	15-45
Using URITypes	15-46
URIType Methods	15-48
Benefits of URI Reference Types	15-49
SYS_DBURIGEN()	15-50
UTL_HTTP Enhancements in Oracle9i	15-52
Cookies in UTL_HTTP	15-53
Summary	15-55

16 API Enhancements

Objectives	16-2
Overview	16-3

New Features of the Extensibility Framework	16-4
User-Defined Aggregate Functions	16-5
Aggregation Logic	16-6
Aggregation Logic	16-7
Aggregation Implementation	16-8
Steps for Creating an Aggregate Function	16-9
Creating the Object Type Specification	16-10
Creating the Object Type Body	16-11
Create the Object Type Body	16-12
Creating the Aggregate Function	16-13
Using the Aggregate Function	16-14
Pipelined Table Functions	16-15
Implementing Table Functions	16-16
ODCITable Interface	16-17
Local Domain Indexes for Range-Partitioned Tables	16-18
Function-Based Domain Indexes	16-19
Domain Indexes on Embedded Object Types	16-20
Domain Indexes on IOTs	16-21
Oracle9i New Features in JDBC	16-22
SQLJ Objects	16-24
Creating SQLJ Object Types	16-25
Creating the SQLJ Class	16-26
Loading the SQLJ Class into the Database	16-28
Creating SQLJ Object Types in the Database	16-29
Other SQLJ Enhancements	16-30

Overview of the Oracle C++ Call Interface	16-31
Applications of OCCI	16-32
Establishing a Connection Using OCCI	16-33
Associative Relational Access	16-34
Associative Access to Access Objects	16-37
Navigational Access to Objects	16-40
Oracle Call Interface (OCI) New Features	16-45
Overview of Scrollable Cursors	16-46
Connection Pooling in OCI	16-47
New Object Type Translator (OTT) Features	16-48
OTT: Example	16-49
Summary	16-50

A Practices

B Practice Solutions

C Frequently Used Table Descriptions

D Review of Objects

E Specialized APIs

F Overview of Oracle8i Release 3 (8.1.7) New Features

G Introduction to iSQL*Plus

Index

Preface

Profile

Before You Begin This Course

Before you begin this course, you should be able to program proficiently in the SQL and PL/SQL languages. You should also be familiar with the Oracle8i database and all features introduced in Oracle8i or prior releases such as objects, XML, Java and so on. The prerequisites for this class are as follows:

Prerequisites

Required

- Introduction to Oracle for Experienced SQL Users or Introduction to Oracle: SQL and PL/SQL
- PL/SQL Program Units
- Have experience programming in an Oracle8i database environment
- Knowledge of object oriented concepts

Suggested

- Oracle8 for Application Developers or Oracle8i: Managing Data
- Oracle8i SQL Statement Tuning
- Advanced PL/SQL
- Have knowledge of developing programs in a language such as C, C++ or Java.

How This Course Is Organized

Oracle9i New Features for Application Developers is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills introduced.

Related Publications

Oracle Publications

Title

Advanced Security Administrator's Guide

OCI Programmer's Guide

Oracle C++ Call Interface Programmer's Guide

Oracle Label Security Administrator's Guide (formerly Oracle Military Security Administrator's Guide)

Oracle Single Sign-On Application Developer's Guide

Oracle9i Application Developer's Guide - Fundamentals

Oracle9i Application Developer's Guide - Large Objects (LOBs)

Oracle9i Application Developer's Guide - Object-Relational Features

Oracle9i Application Developer's Guide - XML, 9.0.1

Oracle9i Data Warehousing Guide

Oracle9i Database New Features (formerly Getting to Know Oracle9i)

Oracle9i Database Performance Guide and Reference (Formerly Designing and Tuning for Performance)

Oracle9i Globalization Support Guide

Oracle9i Security Overview

Oracle9i SQL Reference

Oracle9i Supplied PL/SQL Packages and Types Reference

Oracle9i XML Reference (9.0.1)

PL/SQL User's Guide and Reference, Beta (8.2.0)

Additional Publications

- System release bulletins
- Installation and user's guides
- read.me files
- *Oracle Magazine*

Typographic Conventions

What follows are two lists of typographical conventions used specifically within text or within code.

Typographic Conventions within Text

Convention	Object or Term	Example
Uppercase	Commands, functions, column names, table names, PL/SQL objects, schemas	Use the <code>SELECT</code> command to view information stored in the <code>LAST_NAME</code> column of the <code>EMPLOYEES</code> table.
Lowercase, italic	Filenames, syntax variables, usernames, passwords	where: <i>role</i> is the name of the role to be created.
Initial cap	Trigger and button names	Assign a When-Validate-Item trigger to the ORD block. Choose Cancel.
Italic	Books, names of courses and manuals, and emphasized words or phrases	For more information on the subject see the <i>Oracle Server SQL Language Reference Manual</i> Do <i>not</i> save changes to the database.
Quotation marks	Lesson module titles referenced within a course	This subject is covered in Lesson 3, “Working with Objects.”

Typographic Conventions (continued)

Typographic Conventions within Code

Convention	Object or Term	Example
Uppercase	Commands, functions	SELECT <i>employee_id</i> FROM <i>employees</i> ;
Lowercase, <i>italic</i>	Syntax variables	CREATE ROLE <i>role</i> ;
Initial cap	Forms triggers	Form module: ORD Trigger level: S_ITEM.QUANTITY <i>item</i> Trigger name: When-Validate-Item . . .
Lowercase	Column names, table names, filenames, PL/SQL objects	. . . OG_ACTIVATE_LAYER (OG_GET_LAYER ('prod_pie_layer')) . . . SELECT <i>last_name</i> FROM <i>employees</i> ;
Bold	Text that must be entered by a user	CREATE USER <i>scott</i> IDENTIFIED BY <i>tiger</i> ;

1

Introduction to Oracle9i New Features for Application Developers

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Identify the Oracle9i new features**
- **Identify the course prerequisites**
- **Discuss the course objectives**
- **Discuss the course agenda**

ORACLE

1-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Objectives

This lesson is an introduction to the course. It gives a brief overview of the different features discussed during the course.

Enhancements in Oracle9i: Focus Areas

SQL

PL/SQL

Java

XML

OCI

Objects



Availability

Performance

Security

**Business
Intelligence**

Globalization

Overview of New Features in Oracle9i

Oracle9i introduces several new features. For easy comprehension these features have been classified into focus areas based on functionality and application. The focus areas discussed in this course are:

Development Platforms

This area consists of SQL, PL/SQL, Java, XML, OCI and Objects. This area deals with features that help in application development in any one of the above languages.

Availability

This area deals with the new features that are available to protect against data loss, through system failures, unintended changes and so on.

Performance

This area deals with improved access to data, when executing queries or DML.

Security

This area explains the new security features that have been introduced for data protection.

Business Intelligence

This area covers the new SQL syntax and functions introduced to help OLAP and data warehousing applications

Globalization

This area covers the new datetime, interval, and time zone features as well as Unicode enhancements.

Oracle9i SQL New Features

- **SQL: 1999 compliance**
- **Enhancements to data integrity**
- **Additional DML syntax**
- **AnyType and AnyData**
- **Functional index support for large objects (LOBs)**

ORACLE

1-4

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL Enhancements

The SQL enhancements can be broadly classified into the following:

SQL: 1999 Compliance

New SQL syntax has been introduced in the database which increases the Oracle database conformance to ANSI/ISO SQL:1999 standards.

Data Integrity

Enhancements have been made to constraints to improve the ability to maintain a 100% integrity of data in the database.

Additional DML Syntax

New DML has been introduced to improve performance and to aid data warehousing mechanisms.

AnyType and AnyData

These are new object types introduced in this release which can be used as columns in tables or as variables and parameters in program units.

Functional Indexes

Functional indexes can support LOBs in Oracle9i.

Oracle9i PL/SQL New Features

- **SQL support in PL/SQL**
- **Performance enhancements**
- **Enhancements to supplied packages**

ORACLE

1-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i PL/SQL New Features

The PL/SQL features can be classified into the following:

SQL Support

These features are extensions of features that are introduced in SQL and allow the usage of the new SQL features in PL/SQL code.

Performance Enhancements

These features have been introduced to improve the performance of PL/SQL programs.

Enhancements to Supplied Packages

Some Oracle-supplied packages have been modified to facilitate Web application development.

Oracle9i Java New Features

- **Java Database Connectivity (JDBC) enhancements**
- **Objects support in Java**
- **SQLJ objects**

ORACLE

1-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Java New Features

The Java enhancements are primarily in the JDBC area. JDBC now provides support for REF cursors as well as object enhancements such as inheritance and multilevel collections. Also new objects called SQLJ objects have been introduced in the database.

Oracle9i XML New Features

- **Extensible Markup Language (XML) Developer's Kit**
- **Database enhancements**

ORACLE

1-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i XML New Features

XML was first introduced in the Oracle database in Oracle8i. In this release more components have been added to the XML Developer's Kit such as new JavaBeans, W3C standards compliance, and so on. New database objects have also been introduced, such as the `XMLType` and the `DBURIType`, which allow the storage and manipulation of XML through the database.

Note: For more information on XML, please refer to the appendix on XML.

Oracle9i OCI New Features

- **Oracle C++ Call Interface**
- **AnyType AnyData**
- **Scrollable cursors**

ORACLE

1-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i OCI New Features

Enhancements have been made to the Oracle Call Interface with the most important being the introduction of the Oracle C ++ Call Interface. The existing C interface (OCI), has also been modified to provide support for scrollable cursors and the new objects, AnyType and AnyData.

Oracle9i Objects New Features

Support for object-oriented programming (OOP):

- Object inheritance
- Object type evolution
- Multilevel collections

ORACLE

1-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Objects New Features

The Objects model was introduced to the Oracle database in Oracle8i. In Oracle9i the Objects model has been enhanced to support all object-oriented principles (OOPS). This has been implemented by providing support for true inheritance and allowing Object types to evolve. Also multiple levels of nesting are now permitted in collection objects.

Oracle9i Availability New Features

- **Quiesce database state**
- **Resumable statements**
- **Metadata unload**
- **Undo tablespace**
- **Online table redefinition**

ORACLE

1-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Objects New Features

The Oracle9i database supports several new features that protect the database against data loss. These features include the new database state, Quiesce database state, the ability to resume the execution of statements, the ability to unload metadata, the new undo tablespace which eliminates the need for DBAs to manually configure rollback segments, as well as the change data capture feature.

Oracle9i Performance New Features

- Enhancements to indexes
- Enhanced control of safe cursor sharing
- Cost-based optimizer improvements
- Enhancements to materialized views
- Enhancements to stored outlines

ORACLE

1-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Performance New Features

Performance has been greatly enhanced in the Oracle9i database through the following:

- Enhancements to indexes
- Enhanced control of safe cursor sharing in SQL statements
- Cost-based optimizer improvements
- Enhancements to materialized views
- Enhancements to stored outlines

Oracle9i Security New Features

- **Database workspace management**
- **Secure application role**
- **Virtual Private Database enhancements**
- **Selective data encryption**

ORACLE

1-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Security New Features

The following security features have been introduced:

- **Workspace management:** A new feature called database workspace manager has been introduced.
- **Secure application role:** A new role for applications has been introduced.
- **Virtual Private Database** has been enhanced.
- **Selective data encryption** has been improved.

Oracle9i Business Intelligence Enhancements

- **Grouping sets**
- **External tables**
- **Analytical functions**
- **WITH clause**

ORACLE

1-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Business Intelligence Enhancements

Data warehousing has gained increasing importance in the technology sector and the Oracle9i database provides several new features that support data warehousing applications. The new external tables, MERGE statements, multilevel inserts and so on, tremendously facilitate the Extraction, Transformation and Loading (ETL) process in data warehousing. The new Online analytical processing (OLAP) functions and the WITH clause and grouping sets are powerful analytical tools.

Oracle9i Globalization New Features

- **Datetime data types**
- **Interval data types**
- **Time zones**
- **Datetime functions**
- **Unicode support**

ORACLE

1-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Globalization New Features

With the spread of multinational companies, the demand for global databases has also increased. In Oracle9i several new features have been added to support the storage of different date models and timestamps. Also, the database itself now supports time zone and, year-month and day-second intervals. Additionally, Unicode data can now be stored in the database without changing the database character set.

Other Oracle9i Database Functionality

- **Extensibility**
 - Domain indexes
 - Dynamic data types
 - Table functions
 - User Defined Aggregates
- ***interMedia***
- **Oracle Text**
- **Ultra Search**
- **Advanced Queuing**

ORACLE

1-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Other Oracle9i Database Functionality

The Oracle9i database includes features that enhance the extensibility of the database such as the ability to create user defined aggregate functions and indexes using the Oracle Data cartridge Interface.

The *interMedia* Oracle Text and Ultra Search features and Advanced Queuing features have also been enhanced.

Note: The *intermedia*, Ultra Search, Oracle Text and Advanced Queuing are not intended to be part of the regular three day class. An appendix on specialized APIs, which covers these topics, has been included.

Course Prerequisites

Prior to attending this course participants are expected to be familiar with:

- **Oracle8i database functionality in the areas of security, performance and availability**
- **Basics of objects-oriented programming**
- **XML fundamentals**
- **Java fundamentals**
- **SQL syntax**
- **PL/SQL programming**

ORACLE

1-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Course Prerequisites

Please refer to appendix D through F for information on some of these topics.

Course Objectives

To understand the functionality and applications of the enhancements to the Oracle database in the following areas:

- SQL, PL/SQL and Objects
- Globalization
- Business intelligence
- Availability
- Performance
- Security
- Various APIs such as JDBC and Oracle Call Interface (OCI)
- Web development

ORACLE

Agenda

- **Day 1 Lesson 1-5**
 - Lesson 1: Introduction
 - Lesson 2: Data Integrity Enhancements
 - Lesson 3: New DML Features for Data Warehousing
 - Lesson 4: Business Intelligence Enhancements
 - Lesson 5: SQL: ANSI/ISO SQL: 1999 Standard support in Oracle 9i
- **Day 2 Lesson 6-10**
 - Lesson 6: Datatype Enhancements
 - Lesson 7: Migration LONGS to LOBs
 - Lesson 8: Object and Collection Type Enhancements
 - Lesson 9: SQL Support in PL/SQL
 - Lesson 10: Performance Enhancements in PL/SQL

ORACLE

Note: This is only a guideline. The actual agenda will depend on the background of the students and the preferences of the instructor.

Agenda

- **Day 3 Lesson 11-16**
 - **Lesson 11: Globalization Support**
 - **Lesson 12: Performance and availability Enhancements**
 - **Lesson 13: Data definition and data protection features**
 - **Lesson 14: Database Workspace Manager**
 - **Lesson 15: Support for Web-Based Applications**
 - **Lesson 16: API Enhancements**

ORACLE

Note: This is only a guideline. The actual agenda will depend on the background of the students and the preferences of the instructor.

Summary

In this lesson, you should have learned how to:

- **Identify the Oracle9i new features in different focus areas**
- **Identify the course prerequisites**
- **Identify the course objectives**
- **Discuss the course agenda**

ORACLE

2

Data Integrity Enhancements

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Use new syntax for column default values in `INSERT` and `UPDATE` statements**
- **Efficiently manage indexes used to enforce constraints**
- **Create constraints on views**
- **Use the `FOR UPDATE WAIT` clause to wait a specified period of time for locks**

ORACLE

2-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

For more information, see *Oracle9i Application Developer's Guide - Fundamentals* and *Oracle9i SQL Reference*.

Overview

- **Explicit default column values**
- **Constraint enhancements**
- **FOR UPDATE WAIT clause**

ORACLE

2-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview

In this lesson, you learn about changes to data integrity constraint management and other Oracle9i database enhancements that relate to preserving the integrity of data in the database.

Overview of Explicit Defaults

- The explicit default feature allows use of the **DEFAULT** keyword to control where and when the default value should be applied to data.
- This feature conforms with the **SQL: 1999** standard.
- The explicit default can be used in two types of statements:
 - **INSERT** statements
 - **UPDATE** statements

ORACLE

2-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Explicit Defaults

This feature allows the use of a keyword in **UPDATE** and **INSERT** statements to specify that the new data value for a table column will be the default defined for that column.

Benefits

Explicit defaults provide the following benefits:

- You can ensure data integrity without hard coding literals in applications.
- The explicit default is user friendly and provides a more flexible interface to developers.
- The default value feature conforms with the **SQL: 1999** standard.

Explicit Default

```
ALTER TABLE departments MODIFY location_id  
DEFAULT 1400;
```

```
UPDATE departments SET location_id = DEFAULT  
WHERE location_id = 1500;
```

```
INSERT INTO departments (  
department_id, department_name, ...  
location_id)  
VALUES ( 45, 'Research', ... DEFAULT);
```

ORACLE

2-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Explicit Default

The **DEFAULT** keyword is not strictly required for insert statements, unless you want to insert a row with default values for all columns. In **UPDATE** statements, the **DEFAULT** keyword must be used if the default value is required.

Example

The above example assumes the **DEPARTMENTS** table has been modified so that there is a default for the **LOCATION_ID** column. It also shows how an **UPDATE** and an **INSERT** can use the explicit default.

Explicit Default (continued)

INSERT Statements

This INSERT statement causes the default value to be used for the DEPARTMENT_ID column:

```
INSERT INTO employees (  
    employee_id,  
    last_name,  
    email,  
    hire_date,  
    job_id,  
    salary)  
VALUES (  
    1,  
    'Scott',  
    'Scott@acme.com',  
    SYSDATE,  
    'AD_ASST',  
    3000);
```

It can be rewritten as:

```
INSERT INTO employees(  
    employee_id,  
    last_name,  
    email,  
    hire_date,  
    job_id,  
    salary,  
    department_id)  
VALUES (  
    1,  
    'Scott',  
    'Scott@acme.com',  
    SYSDATE,  
    'AD_ASST',  
    3000,  
    DEFAULT);
```

The second statement provides more clarification because a column is explicitly specified to use the default value.

UPDATE Statements

In an UPDATE statement, you can change a field to the default value associated with a column. For example, the following statement replaces values of the department ID with the default value in all rows where the department ID is 10.

```
UPDATE employees set department_id = DEFAULT  
WHERE department_id = 10;
```

This functionality was not possible prior to Oracle9i.

Overview of Constraint Enhancements

- **Prevention of share lock on unindexed foreign keys**
- **Explicit control over the use of indexes by primary and unique keys**
- **The ability to create primary and foreign keys on views**

ORACLE

2-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Constraint Enhancements

Share locks on unindexed foreign keys do not block DML statements.

You can control the use of indexes by primary key and unique constraints.

You can create primary and foreign keys on views.

Applications:

- An improvement has been made in the locking of unindexed foreign keys during primary key updates.
- You have explicit control over how indexes are affected when creating or dropping unique and primary key constraints.
- Improvement in the lookup of primary keys during foreign key insertion allows faster foreign key insertion.

No Share Lock on Unindexed Foreign Keys

- **A table level share lock is still placed on unindexed foreign keys when doing an update or delete on the primary key.**
- **However, the lock is released immediately after obtaining it.**

ORACLE

2-8

Copyright © Oracle Corporation, 2001. All rights reserved.

No Share Lock on Unindexed Foreign Keys

A table level share lock is still placed on unindexed foreign keys in a child table when doing an update or delete on the primary key column in the referenced parent table. The lock is released immediately after obtaining it. If multiple primary keys are updated or deleted, the lock is obtained and released once per row. The obtaining and releasing of the shared lock are coded as follows:

1. Get a save point.
2. Obtain a share lock.
3. Roll back to save point.

Primary Key Lookup During Foreign Key Creation

- **Primary keys are cached so that the time taken for primary key lookup is reduced.**
- **The creation of foreign keys is faster.**
- **The cache is set up when the second foreign key insert is processed.**

ORACLE

2-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Primary Key Lookup During Foreign Key Creation

The lookup of matching primary keys at the time of foreign key insertion takes time. In release Oracle9i, the first 256 primary keys can be cached so the addition of multiple foreign keys becomes significantly faster. The cache is only set up after the second row is processed. This avoids the overhead of setting up a cache for single row data manipulation language (DML).

Example of Creating Primary Key Constraint

```
CREATE TABLE employees
( employee_id NUMBER PRIMARY KEY USING INDEX
  (CREATE INDEX employees_id_idx ON
    employees(employee_id)),
  first_name VARCHAR2(15),
  last_name VARCHAR2(15),
  ...) ;
```

ORACLE

2-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Creating Primary Key Constraint

In the above example the `CREATE INDEX` clause is used in the `CREATE TABLE` statement to create a separate primary key index explicitly. The `parallel` and `partition` clauses can be used in this syntax to make index creation more efficient. The index can be a concatenated index.

Example of Creating Unique Constraints

```
CREATE TABLE departments(
  department_id NUMBER(4) PRIMARY KEY,
  department_name VARCHAR2(30),
  manager_id NUMBER(6),
  location_id NUMBER(4),
  CONSTRAINT depts_dname_uq
  UNIQUE(department_name, location_id)
  USING INDEX
  (CREATE INDEX depts_dname_idx ON
    departments(department_name, location_id)));
```

Example of Dropping Indexes

```
ALTER TABLE departments
  DISABLE CONSTRAINT depts_dname_uq
  CASCADE DROP INDEX;
ALTER TABLE employees
  DROP PRIMARY KEY KEEP INDEX;
```

ORACLE

2-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Dropping Indexes

In the above example the `DROP INDEX` and `KEEP INDEX` clauses are used to specify whether an index should be dropped or kept when unique or primary key constraints are dropped or disabled.

Note: By default unique indexes that are implicitly created during the creation of primary and foreign keys are dropped when the unique or primary key constraints are dropped or disabled.

Constraints on Views

- **Unique, primary key, and foreign key constraints are now allowed on views.**
- **View constraints are always in the `DISABLE NOVALIDATE` state.**
- **View constraints facilitate query rewrite.**
- **A view constraint in the `RELY` state allows query rewrite to occur when the query integrity level is set to `TRUSTED`.**

ORACLE

2-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Constraints on Views

You can specify constraints on views and object views. You can also specify constraints on view columns and attributes of object views (by specifying the appropriate alias).

View constraints (at the view and at the column or attribute level) are declarative only. That is, the Oracle database does not enforce them. However, operations on views are subject to the integrity constraints defined on the underlying base tables, so you can enforce constraints on views through constraints on base tables.

View constraints are a subset of table constraints, and are subject to these restrictions:

- You can specify only unique, primary key, and foreign key constraints on views.
- Because view constraints are not enforced, you cannot specify any `DEFERRED` or `DEFERRABLE` clause.
- View constraints are supported only in `DISABLE NOVALIDATE` mode. You cannot specify any other mode.
- You cannot specify the `USING INDEX` clause, the `EXCEPTIONS INTO` clause, or the `ON DELETE` clause in view constraints.

Overview of FOR UPDATE WAIT Clause

- The **SELECT . . . FOR UPDATE** statement has been modified to allow the user to specify how long the command should wait if the rows being selected are locked.
- If **NOWAIT** is specified, then an error is returned immediately if the lock cannot be obtained.

ORACLE

2-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of FOR UPDATE WAIT Clause

In prior releases the **SELECT . . . FOR UPDATE** statement had only two alternatives when the rows being selected were already locked: wait for the lock to be released, or return immediately with an error message. Another alternative has been added in Oracle9i to allow the user to specify the time interval to wait before returning with the error.

Benefits

The **UPDATE WAIT** clause provides these benefits:

- It prevents indefinite waits on locked rows.
- It allows more control over the wait time for locks in applications.
- It is very useful for interactive applications because these users cannot wait for indeterminate time intervals.

Example of Using FOR UPDATE WAIT Clause

```
SELECT *  
FROM EMPLOYEES  
WHERE DEPARTMENT_ID = 10  
FOR UPDATE WAIT 20;
```

ORACLE

2-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Using FOR UPDATE WAIT Clause

An integer can be specified after the keyword `WAIT` to indicate the number of seconds to wait for a lock. In the above example the query waits 20 seconds to obtain a lock, and if it is unable to obtain the lock in that time interval it returns an error.

The wait interval must be specified as a numeric literal. It cannot be an expression, bind variable, or PL/SQL variable.

When the lock cannot be obtained within the specified time, the following error occurs:
`ORA-30006: resource busy; acquire with WAIT timeout expired.`

Index-Only Scans on Function-Based Indexes

- The deduction of nulls in expressions based on the underlying columns in Oracle9i allows the optimizer to choose index-only scans on function-based indexes.
- All built-in operators know whether their result is guaranteed to be not null when all their inputs are known to be not null.

ORACLE

2-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Index-Only Scans on Function-Based Indexes

In prior releases index-only scans could be done only when an indexed column was known not to contain nulls. A table column can be constrained to be NOT NULL, but the expression used in a function-based index cannot. Prior to Oracle9i, function-based indexes could not be used for index-only scans unless the query somehow disallowed nulls in the indexed expressions. The deduction of nulls in expressions based on the underlying columns in Oracle9i allows queries to do index-only scans on function-based indexes. All built-in operators know whether their result is guaranteed to be not null when all their inputs are not null.

Example

Create a function-based index on the expression `salary * (1 + commission_pct)`:

```
CREATE INDEX sal_comm_idx  
ON employees (salary * (1 + commission_pct));
```

Consider the following query:

```
SELECT salary * (1 + commission_pct) AS total_comp  
FROM employees;
```

This query can do an index only scan on the SAL_COMM_IDX function-based index when both SALARY and COMMISSION_PCT columns are declared NOT NULL.

Note: To create a function-based index, you must have the QUERY REWRITE system privilege.

Summary

In this lesson, you should have learned how to:

- **INSERT and UPDATE using default values**
- **Manage locking in foreign key, and primary key relationships**
- **Make efficient use of indexes used to enforce constraints**
- **Create constraints on views**
- **Wait for locks**

ORACLE

2-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you have learned about changes to data integrity constraint management and other Oracle9i database enhancements that relate to preserving the integrity of data in the database.

You can INSERT and UPDATE using default values, and simplify procedural logic in applications that require explicit locking.

Practice 2 Overview

This practice covers the following topics:

- **Writing an update statement that sets a column to a default value**
- **Verifying that a function-based index can be used for a fast full scan**
- **Waiting to obtain a lock**

ORACLE

2-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 2 Overview

In this practice you will write an update statement that sets a column to a default value. You will also investigate locking in foreign key and primary key relationships in application code.

To perform this practice go to Appendix A, “Practices.”

3

New DML Features for Data Warehousing

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Query data from an external table**
- **Simultaneously insert data into multiple tables**
- **Use the `MERGE` statement to conditionally update or insert data**

ORACLE

3-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

Data warehousing applications often have requirements to extract data from multiple sources, transform that data to different formats, transport that data to different platforms, and load that data into database tables. The Oracle9i database has a number of new features that streamline these processes.

For more information, see *Oracle9i Data Warehousing Guide* and *Oracle9i SQL Reference*.

Overview

- **Using external tables**
- **Multitable INSERT statements**
- **MERGE statements**

ORACLE

3-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview

New syntax in the `CREATE TABLE` statement allows you to create an external table where the data resides outside the database. This can simplify the process of loading data from outside sources.

New DML statements allow you to insert data into multiple tables and merge data from multiple tables into one table simultaneously.

External Tables

- **External tables are read-only tables where the data is stored outside the database in flat files.**
- **The data can be queried using SQL but UPDATE, INSERT, DELETE, and MERGE statements are not allowed.**
- **Indexes cannot be created on external tables.**
- **The metadata for an external table is created using a CREATE TABLE statement.**
- **An external table definition describes how the external data should be presented to the database.**

ORACLE

3-4

Copyright © Oracle Corporation, 2001. All rights reserved.

External Tables

External tables are like regular SQL tables with the exception that the data is read-only and does not reside in the database; thus the organization is external. The external table can be queried directly and in parallel using SQL. The metadata for the external table is created using the `CREATE TABLE . . . ORGANIZATION EXTERNAL` statement. In this sense, the external table acts as a view.

No DML operations are possible and no indexes can be created on external tables.

The `CREATE TABLE . . . ORGANIZATION EXTERNAL` operation involves only the creation of metadata in the Oracle data dictionary, because the external data must exist outside the database. Once the metadata is created, the external table feature enables the user to easily perform parallel extraction of data from the specified external sources.

Applications

External tables allow external data to be queried and joined directly and in parallel without requiring it to be loaded into the database. They can eliminate the need for staging the data within the database for extract, transport, transform, and load operations in data warehousing applications. They are useful in environments where an external source has to be joined with database objects and then transformed. They are most useful when the external data is large and not queried frequently.

External tables can be used for loading data into the database. They allow access to data stored outside the database. These tables are intended primarily for usage in `INSERT . . . AS SELECT` and `DELETE . . . AS SELECT` statements and not for adhoc querying.

Example of Defining External Tables

```
CREATE table employees_ext(employee_id NUMBER,  
first_name CHAR(30)), last_name CHAR(30), ...)  
ORGANIZATION EXTERNAL  
  TYPE LOADER  
(DEFAULT DIRECTORY delta_dir  
  ACCESS PARAMETERS  
    (RECORDS DELIMITED BY NEWLINE  
    FIELDS TERMINATED BY ','  
      (enum INTEGER(2), fname CHAR(11),  
      lname CHAR(18), ...)  
  LOCATION ('/employee/delete_emp1.txt',  
    '/employee/delete_emp2.txt') PARALLEL;
```

ORACLE

3-5

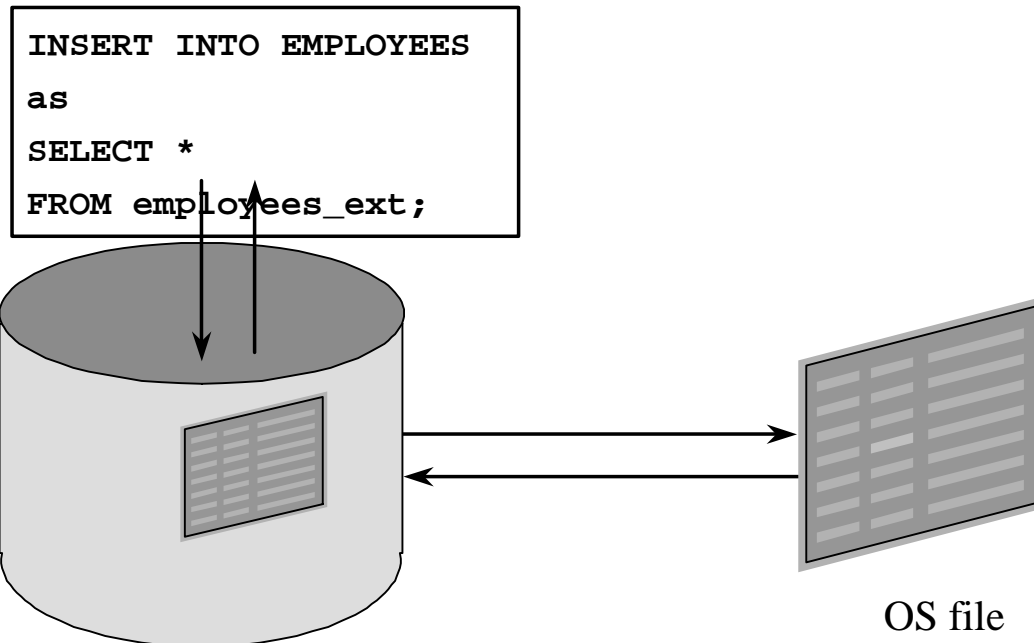
Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Defining External Tables

In the example above an external table named EMPLOYEES_EXT is defined. DELTA_DIR is a directory where the external flat files reside. The access parameters control the extraction of data from the flat file using record and file formatting information. The directory is a nonschema object introduced in Oracle8 that defines the path to an operating system directory. The parameters specified are the same parameters that are used by the SQL*Loader utility.

Note: The above example only provides partial code.

Querying External Tables



ORACLE

3-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Querying External Tables

In the above example, when the external `EMPLOYEES_EXT` table is queried the data is retrieved from the external data files. When the user selects data from the external table, the dataflow goes from the external data source to the Oracle SQL engine where data is processed. As data is extracted, it is transparently converted by the external agent from its external representation into an equivalent Oracle native representation (the loader stream).

Multitable INSERT Statements

- **Allow the `INSERT . . . SELECT` statement to insert rows into multiple tables as part of a single DML statement**
- **Can be used in data warehousing systems to transfer data from one or more operational data sources to a set of target tables**
- **Can be used for refreshing materialized views**

ORACLE

3-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Multitable INSERT Statements

The `INSERT... SELECT` statement with the new syntax can be parallelized and used with the direct load mechanism. The multitable `INSERT` statement inserts computed rows derived from the rows returned from the evaluation of a subquery. There are two forms of the multitable `INSERT` statement: unconditional and conditional. For the unconditional form, an `INTO` clause list is executed once for each row returned by the subquery. For the conditional form, `INTO` clause lists are guarded by `WHEN` clauses that determine whether the corresponding `INTO` clause list is executed.

An `INTO` clause list consists of one or more `INTO` clauses. The execution of an `INTO` clause list causes the insertion of one row for each `INTO` clause in the list.

An `INTO` clause specifies the target into which a computed row is inserted. The target specified can be any table expression that is legal for an `INSERT . . . SELECT` statement. However aliases cannot be used. The same table can be specified as the target for more than one `INTO` clause.

An `INTO` clause also provides the value of the row to be inserted using a `VALUES` clause. An expression used in the `VALUES` clause can be any legal expression, but can only refer to columns returned by the select list of the subquery. If the `VALUES` clause is omitted, the select list of the subquery provides the values to be inserted. If a column list is given, each column in the list is assigned a corresponding value from the `VALUES` clause or the subquery. If no column list is given, the computed row must provide values for all columns in the target table.

Advantages of Multitable INSERTs

- **Eliminates the need for multiple INSERT . . . SELECT statements to populate multiple tables**
- **Eliminates the need for a procedure to do multiple inserts using conditional logic**
- **Significant performance improvement over the above two methods due to the elimination of the cost of materialization and repeated scans on the source data**

ORACLE

Types of Multitable INSERT Statements

- **Unconditional INSERT ALL**
- **Conditional INSERT ALL**
- **Conditional INSERT FIRST**

ORACLE

3-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Types of Multitable INSERT Statements

This feature is an Oracle extension to SQL and is not part of the SQL: 1999 standard.

Example of Unconditional INSERT

```
INSERT ALL INTO product_activity VALUES(  
    today, product_id, quantity)  
INTO product_sales VALUES(  
    today, product_id, total)  
SELECT TRUNC(order_date) today, product_id,  
    SUM(unit_price) total, SUM(quantity) quantity  
FROM orders, order_items  
WHERE orders.order_id = order_items.order_id  
    AND order_date = TRUNC(SYSDATE)  
GROUP BY TRUNC(order_date), product_id;
```

ORACLE

Example of Unconditional INSERT

The above example inserts the values of order date, product_id and sum(quantity) into the product_activity table and order date, product_id and sum(unit_price) into the product_sales table simultaneously. The values are obtained through a subquery selecting from a join between the orders and order_items tables.

Example of Pivoting INSERT ALL

```
INSERT ALL INTO sales VALUES (  
product_id, TO_DATE(week_id, 'WW'), sales_sun)  
INTO sales VALUES (  
product_id, TO_DATE(week_id, 'WW'), sales_mon)  
INTO sales VALUES (  
product_id, TO_DATE(week_id, 'WW'), sales_tue)  
INTO sales VALUES (  
product_id, TO_DATE(week_id, 'WW'), sales_wed)  
INTO sales VALUES (  
product_id, TO_DATE(week_id, 'WW'), sales_thu)  
...
```

ORACLE

3-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Pivoting INSERT ALL

The above slide is an example of inserting into the same table several times, pivoting from a nonnormalized form to a normalized form.

Example of Pivoting INSERT ALL

```
...  
INTO sales VALUES (  
product_id, TO_DATE(week_id, 'WW'), sales_fri)  
INTO sales VALUES (  
product_id, TO_DATE(week_id, 'WW'), sales_sat)  
SELECT product_id, week_id,  
       sales_sun, sales_mon, sales_tue, sales_wed,  
       sales_thu, sales_fri, sales_sat  
FROM sales_source_data;
```

ORACLE

Example of Pivoting INSERT ALL

This is a continuation of the example in the previous slide.

Example of Conditional INSERT ALL

```
INSERT ALL
  WHEN product_id IN
    (select product_id FROM promotional_items)
  INTO promotional_sales
    VALUES (product_id, list_price)
  WHEN order_mode = 'online'
  INTO web_orders
    VALUES (product_id, order_total)
SELECT product_id, list_price, order_total,
       order_mode FROM orders;
```

ORACLE

Example of Conditional INSERT ALL

The above example inserts a row into the `promotional_sales` table for products sold that are on the promotional list, and into the `web_orders` table for products for which online orders were used. It is possible that two rows are inserted for some rows from the subquery, and none for others. An `ELSE` clause can be added to guarantee that each row from the subquery is inserted at least once.

Example of Conditional INSERT FIRST

```
INSERT FIRST
  WHEN order_total > 10000 THEN
    INTO priority_handling VALUES (id)
  WHEN order_total > 5000 THEN
    INTO special_handling VALUES (id)
  WHEN order_total > 3000 THEN
    INTO privilege_handling VALUES (id)
  ELSE
    INTO regular_handling VALUES (id)
SELECT order_total, order_id id FROM orders;
```

ORACLE

Example of Conditional INSERT FIRST

The above statement inserts into an appropriate handling table according to the total of an order. Because the **FIRST** keyword is used, each row from the subquery is inserted no more than one time. Because of the **ELSE** clause, each row from the subquery is guaranteed to be inserted.

MERGE statements

- **Provide the ability to “upsert,” that is, conditionally update or insert into the database**
- **Do an update if the row exists, and an insert if it is a new row**
- **Avoid multiple updates**
- **Can be used in data warehousing applications**

ORACLE

3-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Applications of MERGE statements

- MERGE statements use a single SQL statement to complete an UPDATE or INSERT or both.
- The statement can be parallelized transparently.
- The array interface can be used for bulk operations.
- Performance is improved because fewer statements require fewer scans of the source tables.

Example of Using the MERGE Statement in Data Warehousing

```
MERGE INTO customer C USING cust_src s
ON (c.customer_id = s.src_customer_id)
WHEN MATCHED THEN
    UPDATE SET c.cust_address = s.cust_address
WHEN NOT MATCHED THEN
    INSERT (customer_id, cust_first_name, ...)
VALUES (src_customer_id, src_first_name,
    ...);
```

ORACLE

3-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of MERGE Statement

This is an example of using the MERGE statement in data warehousing. `customer` is a large fact table and `cust_src` is a smaller table with rows that need to be inserted into `customer` conditionally. A table like the `cust_src` table is sometimes said to contain the "deltas" or changes that in this case need to be applied to the other table.

This MERGE statement indicates that the `customer` table has to be merged with the rows returned from the evaluation of the ON clause. The ON clause in this case is the `cust_src` table, but it can be an arbitrary query. Each row from the `cust_src` table is checked for a match to any row in the `customer` table by satisfying the join condition specified by the ON clause. If so, each row in the `customer` table is updated using the UPDATE SET clause of the MERGE statement. If no such row exists in the `customer` table, then the rows are inserted into the `customer` table.

Summary

In this lesson, you should have learned how to:

- **Query data from an external table**
- **Simultaneously insert data into multiple tables**
- **Use the `MERGE` statement to conditionally update or insert data**

ORACLE

3-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

The ability to create an external table can simplify the process of loading data from outside sources.

New DML statements allow you to insert data into multiple tables and merge data from multiple tables into one table simultaneously.

Practice 3 Overview

This practice covers the following topics:

- **Writing a multitable `INSERT` statement**
- **Creating and querying an external table**
- **Writing a `MERGE` statement**

ORACLE

3-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 3 Overview

In this practice you will use new DML statements and work with an external table in a scenario that involves the extraction, transformation and loading of data.

Scenario

Your company has decided to conduct a marketing campaign to increase awareness of a new product among existing and potential customers. Using data from your company's data warehouse, and a list purchased from an outside supplier, you will populate two tables. One table will contain information about existing customers who are less likely to purchase the new product. The other table will contain information about existing customers that are more likely to purchase the new product, plus the purchased information.

To perform this practice go to appendix A, "Practices."

4

Business Intelligence Enhancements

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Use grouping sets**
- **Create SQL statements with the new `WITH` clause**
- **Identify the enhanced Oracle9i analytical functions**

ORACLE

4-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Objectives

In this lesson, you will learn about the new analytical function introduced in the Oracle9i database. You will see the syntax and applications of these functions.

Overview of Grouping Sets

- **Specify groupings in the GROUP BY clause**
- **Superset of ROLLUP and CUBE**
- **Produces a single result set which is equivalent to a UNION ALL approach**
- **Composite columns: Treat columns enclosed in parenthesis as a unit**
- **Concatenated groupings: A concise way to generate useful combinations of groupings**

ORACLE

4-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Grouping Sets

A grouping set is a set of groupings that you want the system to perform, in order to apply aggregate functions on those groupings. This is done in the GROUP BY clause.

In Oracle8i the CUBE and ROLLUP enhancements to the GROUP BY clause were introduced; grouping sets in Oracle9i add even more flexibility.

Without the enhancements in Oracle9i, multiple queries combined together with UNION ALL are required to achieve these tasks. A multiquery approach is inefficient, for it requires multiple scans of the same data. The extensions to the GROUP BY clause in Oracle9i allow the optimizer to choose better plans, enabling the SQL execution engine to execute the query efficiently.

The GROUPING SET clause allows you to identify the exact groups you are interested in. For example, GROUP BY CUBE (a, b, c) produces $2^3 = 8$ groupings in total; but you may be only interested in three of those eight groupings.

Composite columns and concatenated groupings are discussed in more detail in the following pages.

Grouping Sets

A grouping set is a set of groups that the user wants the system to form.

Gives the user the power to specify exactly the groupings of interest in the GROUP BY clause

Produces a single result set which is equivalent to a UNION ALL approach

Adheres to ISO SQL:1999 standards

Grouping set efficiency:

- Only one pass over base table is required
- No need to write complex UNION statements
- The more elements the grouping sets have, the higher the gain using grouping sets

ORACLE

4-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Grouping Sets

A grouping set is a set of groups that the user wants the system to form.

Without the enhancements in Oracle9i, multiple queries combined together with UNION ALL are required to achieve these tasks. A multiquery approach is inefficient, for it requires multiple scans of the same data. The extensions to the GROUP BY clause in Oracle9i allow the optimizer to choose better plans, enabling the SQL execution engine to execute the query efficiently. Users can analyze data in one dimension without completely rolling it up, analyze across multiple dimensions without computing the whole CUBE, or specify multiple arbitrary groupings to meet any need.

As with the analytic functions, the GROUP BY extensions follow the international SQL:1999 standards.

GROUPING SETS

- Allows you to define multiple groupings in the same query
- GROUP BY computes all the groupings specified and combines them with the UNION ALL operator
- The following two syntax clauses are equivalent

```
GROUP BY GROUPING SETS (hire_date,  
department_id, job_id)
```

```
GROUP BY hire_date  
UNION ALL  
GROUP BY department_id  
UNION ALL  
GROUP BY job_id;
```

ORACLE

Grouping Sets

Using the UNION ALL operator instead of the GROUPING SETS clause, more scans of the base table are required, making it inefficient. For example the query:

```
SELECT      manager_id, null hire_date , null job_id,  
            count(*)  
FROM        employees  
GROUP BY    manager_id, 2, 3  
UNION ALL  
SELECT      null, hire_date, null, count(*)  
FROM        employees  
GROUP BY    1,hire_date, 3  
UNION ALL  
SELECT      null, null, job_id, count(*)  
FROM        employees  
GROUP BY    1,2,job_id;
```

Can be written using grouping sets as:

```
SELECT      Hire_date, manager_id, job_id, count(*)  
FROM        employees  
GROUP BY    GROUPING SETS (hire_date, manager_id, job_id);
```

Grouping Sets Example

```
SELECT time_id, channel_id, prod_id,  
       ROUND(SUM(amount_sold)) AS sales  
FROM sales  
WHERE (time_id = '01-DEC-1999' or  
time_id = '02-DEC-1999') AND prod_id IN  
(10, 20, 45)  
GROUP BY GROUPING SETS  
  ((time_id, channel_id, prod_id),  
   (time_id, channel_id),  
   (channel_id, prod_id));
```

ORACLE

Grouping Sets Example

This example is based on the `sales` table, with foreign keys to several dimension tables. In this example, you specify three groupings:

- `(TIME_ID, CHANNEL_ID, PROD_ID)`
This results in aggregate information per day per channel per product.
- `(TIME_ID, CHANNEL_ID)`
This results in aggregate information per day per channel.
- `(CHANNEL_ID, PROD_ID)`
This results in aggregate information per channel per product.

Grouping Sets Example: Results

TIME_ID	C	PROD_ID	SALES	
01-DEC-99	C	45	79	Time, channel, and product
01-DEC-99	I	20	266	
01-DEC-99	S	45	316	
01-DEC-99	T	45	790	
02-DEC-99	C	20	14	
02-DEC-99	C	45	4187	
02-DEC-99	T	45	158	
01-DEC-99	C		79	Time and channel
01-DEC-99	I		266	
01-DEC-99	S		316	
01-DEC-99	T		790	
02-DEC-99	C		4201	
02-DEC-99	T		158	
	C	20	14	Channel and product
	C	45	4266	
...				

ORACLE

4-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Grouping Sets Example (continued)

The first two rows of the result above can be read as follows:

- On December 1, through channel C, product 45 was sold for a total amount of 79.
- On December 1, through channel I, product 20 was sold for a total amount of 266.

The results for grouping set of time and channel are interpreted as:

- On December 1, through channel C, total amount was 79.
- On December 2, through channel C, total amount was 4201 (4187 +14).

The results for the grouping set of channel and product should be interpreted as:

- Through channel C, product 20 was sold for a total of 14.
- Through channel C, product 45 was sold for a total of 4266(4187 +79).

GROUPING SETS Versus CUBE and ROLLUP Statements

- The **GROUPING SET** clause allows you to identify exact groups.
- **GROUP BY CUBE** (time, channel, product) produces $2^3=8$ groupings.
- Only 3 groupings are needed.

```
SELECT time_id, channel_id, prod_id,
       round(sum(amount_sold)) AS sales
FROM   sales
WHERE  (time_id = '01-DEC-1999' OR
       time_id = '02-DEC-1999')
       AND prod_id IN (10, 20, 45)
GROUP BY CUBE(time_id, channel_id, prod_id);
```

ORACLE

4-8

Copyright © Oracle Corporation, 2001. All rights reserved.

GROUPING SETS Versus CUBE and ROLLUP Statements

The **GROUPING SET** statement shown previously uses composite columns to identify the exact sets wanted. The **GROUP BY CUBE** statement computes all the 8 ($2*2*2$) groupings. The output is more than needed with more being calculated, and more overhead.

TIME_ID	C	PROD_ID	SALES
-----	-	-----	-----
01-DEC-99	C	45	79
01-DEC-99	C		79
01-DEC-99	I	20	266
01-DEC-99	I		266
01-DEC-99	S	45	316
01-DEC-99	S		316
01-DEC-99	T	45	790
01-DEC-99	T		790
01-DEC-99		20	266
01-DEC-99		45	1185
01-DEC-99			1451

...

Composite Columns

Treat a group of columns as a unit:

- **GROUP BY ROLLUP (a,b,c) gives four groupings**
- **GROUP BY ROLLUP (a,(b,c)) gives three groupings**

```
SELECT prod_id, channel_id, time_id,  
       ROUND(SUM(amount_sold)) sales  
FROM sales  
WHERE (time_id = '01-DEC-1999' or  
       time_id = '02-DEC-1999')  
      AND prod_id IN (10, 20, 45)  
GROUP BY ROLLUP (prod_id, (channel_id,  
                       time_id));
```

ORACLE

4-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Composite Columns

A composite column is a collection of columns that are treated as a unit during the computation of groupings. With CUBE and ROLLUP, you do not have full control over the aggregation levels. For example, the following statement:

```
SQL> SELECT ...  
2 FROM sales  
3 WHERE ...  
4 GROUP BY ROLLUP(prod_id, channel_id, time_id);
```

would result in computing four groupings:

- product, channel, time
- product, channel
- product
- grand total

However, the example in the slide above results in only three groups:

- product, channel, time
- product
- grand total

Composite Columns

PROD_ID	C	TIME_ID	SALES

20	C	02-DEC-99	14
20	I	01-DEC-99	266
20			280
45	C	01-DEC-99	79
45	C	02-DEC-99	4187
45	S	01-DEC-99	316
45	T	01-DEC-99	790
45	T	02-DEC-99	158
45			5530
			5810

Totals per product
per channel per time

Total per product (20)

Totals per product per
channel per time

Total per product (45)

Grand total

ORACLE

4-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Composite Columns (continued)

This slide shows the results for the example query on the previous page. As you can see, the result shows groupings for:

- product, channel, time
- product
- grand total

Concatenated Groupings

- **A concise way to generate useful combinations of groupings**
- **Groupings specified with the concatenated groupings yield the cross-product of groupings from each set**
- **Concatenated groupings are specified by listing multiple grouping sets, cubes, and rollups**
- **Easy to develop**
- **Useful for OLAP applications**

ORACLE

4-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Concatenated Groupings

Concatenated groupings offer a concise way to generate useful combinations of groupings. Groupings specified with concatenated groupings yield the cross-product of groupings from each grouping set. The cross-product operation enables even a small number of concatenated groupings to generate a large number of final groups. The concatenated groupings are specified simply by listing multiple grouping sets, cubes, rollups, and separating them with commas.

Benefits of Concatenated Groupings

- Easy to develop
- Use by applications, SQL generated OLAP applications often involve concatenation of grouping sets, with each grouping set defining groupings needed for a dimension.

Concatenated Groupings

Example: Aggregate costs values for each product rolled up across time, and across channel.

```
SELECT prod_id, channel_id, time_id,  
       ROUND(SUM(amount_sold)) AS sales  
FROM sales  
WHERE (time_id = '01-DEC-1999' or  
       time_id = '02-DEC-1999')  
      AND prod_id IN (10, 20, 45)  
GROUP BY prod_id, CUBE(channel_id),  
         ROLLUP(time_id);
```

ORACLE

Concatenated Groupings (continued)

This results in the following groupings:

- product, channel, time
- product, channel
- product, time
- product

Concatenated Groupings (continued)

The result of the query on the previous page are as follows:

PROD_ID	C	TIME_ID	SALES
20	I	01-DEC-99	266
20	C	02-DEC-99	14
45	C	01-DEC-99	79
45	S	01-DEC-99	316
45	T	01-DEC-99	790
45	C	02-DEC-99	4187
45	T	02-DEC-99	158
20		01-DEC-99	266
20		02-DEC-99	14
45		01-DEC-99	1185
45		02-DEC-99	4345
20	C		14
20	I		266
20			280
45	C		4266
45	S		316
45	T		948
45			5530

The WITH Clause: Overview

- **Allows you to use the same query block in a `SELECT` statement when it occurs more than once within a complex query**
- **Retrieves the results of a query block and stores them in the user's temporary tablespace**
- **Can improve performance**

ORACLE

4-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The WITH Clause: Overview

Using the WITH clause, you can reuse the same query through materialization when it is high cost to evaluate the query block and it occurs more than once within a complex query.

The WITH clause allows you to define a query block before using it in a query. This can improve performance.

The WITH Clause

- Name a query block in a **SELECT** statement to reference it more than once within a query
- The **WITH** clause can hold multiple query blocks, separated by commas
- Resolved as in-line views or temporary tables

ORACLE

4-15

Copyright © Oracle Corporation, 2001. All rights reserved.

The WITH Clause

Using the **WITH** clause, you can reuse the same query block when it is expensive to evaluate the query block and it occurs more than once within a complex query.

The **WITH** clause allows you to define a query block before using it in a query. This can improve performance, and it certainly improves readability and maintainability of your SQL statements. Using the **WITH** clause, you can isolate the business question from the data gathering.

A query name is visible to all **WITH** element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).

The **WITH** clause is internally resolved as an in-line view or a temporary table; the cost-based optimizer chooses the appropriate resolution.

The WITH Clause: Example

Example: Find all departments whose total salary cost is above one-eighth the total salary cost of the whole.

```
WITH
summary AS (
  SELECT department_name, SUM(salary) AS dept_total
  FROM employees, departments
  WHERE employees.department_id =
         departments.department_id
  GROUP BY department_name )
SELECT department_name, dept_total
FROM summary
WHERE dept_total > (
                     SELECT SUM(dept_total) * 1/8
                     FROM summary )
ORDER BY dept_total DESC;
```

ORACLE

The WITH Clause: Example

The output from the query identifies Sales and Shipping departments as having salary costs one-eighth above the total salary cost for the company.

DEPARTMENT_NAME	DEPT_TOTAL
-----	-----
Sales	304500
Shipping	156400

The WITH Clause: Implementation

Equivalent query, less powerful syntax:

```
SELECT department_name, SUM(salary) AS dept_total
FROM employees, departments
WHERE employees.department_id =
      departments.department_id
GROUP BY department_name HAVING
      SUM(salary) > (
          SELECT SUM(salary) * 1/8
          FROM employees, departments
          WHERE employees.department_id =
                departments.department_id)
ORDER BY sum(salary) DESC;
```

ORACLE

The WITH Clause: Implementation

The query tries to retrieve all departments whose total salary cost is above one-eighth for the total salary cost of the company. In the query, there are two blocks, one main query block, and one subquery block. Both blocks need to do some common joins and aggregations.

Using the WITH clause, you can materialize the query block that does the GROUP BY, thus avoiding summarizing the department total cost more than once.

The WITH Clause: Usage Notes

- **Used only for `SELECT` statements**
- **A query name is visible to all `WITH` element query blocks defined after it and the main query block itself.**
- **When the query name is the same as an existing table name, the parser searches from the inside out. The query block name takes precedence over the table name.**
- **The `WITH` clause can hold more than one query. Each query is separated by a comma.**

ORACLE®

4-18

Copyright © Oracle Corporation, 2001. All rights reserved.

The WITH Clause: Usage Notes

A query name is visible to all `WITH` element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).

A query name can be the same as some persistent table name or query name in `WITH` list of another query block. If this happens, the parser searches for the right definition inside out (in terms of query block nesting). The innermost query name definition is used for resolution.

Users must make sure that it does not conflict with permanent tables if that is not their intention. This is similar to the C language, in which a variable has priority when its name is the same as some global variable.

The WITH Clause: Benefits

Using the WITH clause adds business value because:

- **The business question is isolate from data gathering.**
- **The query is easy to read.**
- **A clause is evaluated only once, even if it might appear multiple times in the query.**
- **Performance is improved since the instantiated subquery clause must be calculated only once.**

ORACLE

Analytical Function Enhancements

- **Inverse percentile functions:**
 - `PERCENTILE_CONT`
 - `PERCENTILE_DISC`
- **What-if rank and distribution functions:**
 - `RANK`, `DENSE_RANK`
 - `PERCENT_RANK`, `CUME_DIST`
- **FIRST and LAST aggregate functions**
- **WIDTH_BUCKET function**
- **Grouping sets**

ORACLE

4-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Analytical Function Enhancements

Using inverse percentile functions, you can find the data that corresponds to a specified percentile value. Hypothetical rank and distribution allow queries to find the rank or percentile value of a hypothetical value if it is added to an existing data set.

The `FIRST` and `LAST` functions enable you to specify sorted aggregate groups and return the first or last value of each group. The `WIDTH_BUCKET` function returns the histogram bucket in which a certain input value would fall, given a certain histogram specification.

Grouping sets are extensions to the `GROUP BY` clause that determine which result set rows are subtotals, and specify the exact level of aggregation for a given subtotal.

Benefits

The processing optimizations supported by these functions enable significantly better query performance, in particular for OLAP products such as Oracle Express.

You can perform complex data analysis with much clearer and more concise SQL code. Tasks which in the past required multiple SQL statements or the use of procedural languages can now be expressed using single SQL statements.

The syntax leverages existing aggregate functions, such as `SUM` and `AVG`, so that these well-understood keywords can be used in extended ways.

Inverse Percentile Functions: Description

Two new functions, PERCENTILE_CONT and PERCENTILE_DISC, determine the value that corresponds to a specific percentile.

- **Require a sort specification and a parameter that takes a value between 0 and 1**
- **Use the new WITHIN GROUP clause to specify the data ordering**
- **Can be used as either aggregate functions or reporting aggregate functions**

ORACLE

4-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Inverse Percentile Functions Description

One very common analytic question is to find the value in a data set that corresponds to a specific percentile. Two new Oracle9i functions, PERCENTILE_CONT and PERCENTILE_DISC, compute inverse percentiles. These functions require a sort specification and a percentile parameter value between 0 and 1. The functions can be used as either aggregate functions or reporting aggregate functions. When used as aggregate functions, they return a single value per ordered set. When used as reporting aggregate functions, they repeat the data on each row.

The functions use a new WITHIN GROUP clause to specify the data ordering.

The PERCENTILE_DISC function returns the actual "discrete" value which is closest to the specified percentile values.

The PERCENTILE_CONT function calculates a "continuous" percentile value by using linear interpolation.

Example of Inverse Percentile Functions

```
SELECT department_id,  
PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY  
salary DESC)  
"Median cont",  
PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY  
salary DESC)  
"Median disc"  
FROM employees  
GROUP BY department_id;
```

ORACLE

4-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Inverse Percentile Functions

The above example computes the median salary in each department.

DEPARTMENT_ID	Median-cont	Median-disc
10	4400	4400
20	9500	13000
30	2850	2900
40	6500	6500
50	3100	3100
60	4800	4800
70	10000	10000
80	8800	8800

...

What-if Rank and Distribution Functions

Find out how a value would rank if it were added to a group of data:

- **RANK** and **DENSE_RANK** rank items in a group
- **PERCENT_RANK** returns the percent rank
- **CUME_DIST** computes the cumulative distribution

ORACLE

What-if Rank and Distribution: Description

In certain analyses, such as financial planning, you may want to know how a data value would rank if it is added to a data set. For example, if a new worker is hired at a salary of \$10,000, how would his or her salary rank compare to the other salaries in the company?

The hypothetical rank and distribution functions support this form of what-if analysis. They return the rank or percentile value which a row would be assigned if the row was hypothetically inserted into a set of other rows.

The hypothetical functions can calculate **RANK**, **DENSE_RANK**, **PERCENT_RANK**, and **CUME_DIST**. Like the inverse percentile functions, the hypothetical rank and distributions functions use a **WITHIN GROUP** clause containing an **ORDER BY** specification.

Note: the **RANK** and **DENSE_RANK** functions are already available with Oracle8i, but in Oracle9i they are enhanced to accept arguments for what-if analysis.

RANK and DENSE_RANK: Example

```
SELECT department_id
,      ROUND(AVG(salary))      AS avg_sal
,      RANK(10000) WITHIN GROUP
      (ORDER BY salary DESC)   AS rank
,      DENSE_RANK(10000) WITHIN GROUP
      (ORDER BY salary DESC)   AS dense_rank
FROM   employees
GROUP  BY department_id;
```

ORACLE

4-24

Copyright © Oracle Corporation, 2001. All rights reserved.

RANK and DENSE_RANK: Example

This example determines where a new salary of \$10,000 ranks when compared to the salaries for each department. The results look like the following:

DEPARTMENT_ID	AVG_SAL	RANK	DENSE_RANK
10	4400	1	1
20	9500	2	2
30	4150	2	2
40	6500	1	1
50	3476	1	1
60	5760	1	1
70	10000	1	1
80	8900	9	7

...

Note that the salaries for department 90 are (17000, 17000, and 24000) and the salaries for department 100 are (6900, 7700, 7800, 8200, 9000, and 12000.)

In department 100, a new salary of 10,000 ranks second highest.

In department 90, a new salary of 10,000 ranks fourth. The dense rank rates third, as there are two salaries which are equal in department 90.

FIRST and LAST Aggregate Values

Specify sorted aggregate groups and return the first or last value of each group.

- **Obtain the first or last value of a column based on ordering of *another* column**
- **Can be used in nonadditive aggregate calculations**
- **Result in a simpler syntax and better performance**

ORACLE

4-25

Copyright © Oracle Corporation, 2001. All rights reserved.

FIRST and LAST Aggregate Values: Description

FIRST and LAST aggregate functions allow you to specify an order within the aggregated groups and then return the first or last row of each group.

While an equivalent query can be created using a join or subquery, the SQL syntax is cumbersome and performance can be inefficient. The FIRST and LAST functions do this work with simpler SQL syntax and greater performance.

FIRST and LAST Aggregate Values

```
SELECT manager_id
,      MIN(salary) KEEP
      (DENSE_RANK FIRST
       ORDER BY commission_pct) AS low_comm
,      MAX(salary) KEEP
      (DENSE_RANK LAST
       ORDER BY commission_pct) AS high_comm
FROM    employees
WHERE   commission_pct IS NOT NULL
GROUP  BY manager_id;
```

ORACLE

4-26

Copyright © Oracle Corporation, 2001. All rights reserved.

FIRST and LAST Aggregate Values: Example

The above example retrieves the salary with the lowest commission and the salary with the highest commission per manager, using the FIRST/LAST functions.

Lines 3 and 4 define a group of rows with the lowest commissions (per manager), and out of that group the minimum salary is retrieved; lines 6 and 7 select the maximum salary from the group of the highest commission percentages.

The result looks like the following:

MANAGER_ID	LOW_COMM	HIGH_COMM
100	10500	14000
145	7000	10000
146	7000	10000
147	6200	10500
148	6100	11500
149	6200	11000

The results show that the lowest commission is earned on the salary amount of 10500 for manager 100. The highest commission is earned on the salary amount of 14000 for manager 100. Per each manager ID, the report displays the salary with the lowest commission and the salary with the highest commission.

The WIDTH_BUCKET Function

Returns the bucket number that the result of an expression will be assigned to after it is evaluated.

```
WIDTH_BUCKET(input_expression,  
             low_boundary, high_boundary,  
             bucket_count)
```

ORACLE

4-27

Copyright © Oracle Corporation, 2001. All rights reserved.

The WIDTH_BUCKET Function

For any given expression, the WIDTH_BUCKET function returns the bucket number that the result of this expression will be assigned to after it is evaluated.

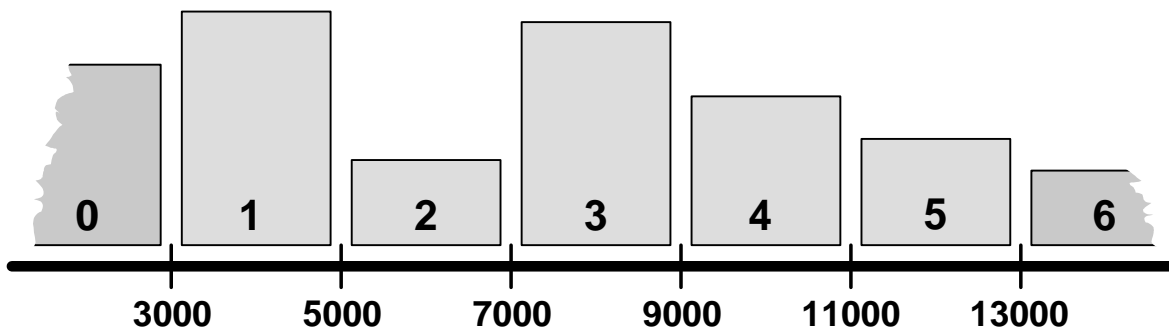
You can generate equiwidth histograms with this function. Equiwidth histograms divide data sets into buckets with an equal interval size.

You provide the input expression, the minimum boundary value, the maximum boundary value, and the number of buckets.

Note: If you ask for (n) buckets, you actually get (n + 2) buckets. The two additional buckets will hold any values below the low boundary value or above the high boundary value.

The WIDTH_BUCKET: Example

```
SELECT last_name, salary,  
       WIDTH_BUCKET(salary,3000,13000,5)  
FROM   employees;
```



ORACLE

4-28

Copyright © Oracle Corporation, 2001. All rights reserved.

The WIDTH_BUCKET: Example

In the above example, salaries less than 3000 are placed in bucket 0; salaries greater than 13000 are placed in bucket 6. The other salaries are placed in buckets 1 through 5, depending on the salary value.

LAST_NAME	SALARY	WIDTH_BUCKET
King	24000	6
Kochhar	17000	6
De Haan	17000	6
Hunold	9000	4
Ernst	6000	2
Austin	4800	1

...

Business Scenario

The width bucket can be used in cases where you want to find the high, low and average of a set of values. For example, sales representatives are required to sell \$10,000 every month. The acceptable range is from \$7000 through \$12000. Divide this into 5 buckets. Sales representatives in buckets 1 and 2 get 10% commissions. Sales representatives in bucket 3 get 15% commissions. Buckets 4 and 5 get 20% commissions. Underachievers in bucket 0 get no commission and overachievers in bucket 6 get 30% commission. This can be applied to other scenarios such as the stock market or best selling products.

Benefits of Analytical Functions

Key benefits provided by the new functions include:

- **Improved query speed**
- **Enhanced developer productivity**
- **Minimized learning effort**
- **Standardized syntax**

ORACLE

4-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Benefits

Improved Query Speed: By using native SQL, the processing optimizations supported by these functions enable significantly better query performance. The performance enhancements enabled by the new functions enhance query speeds for Oracle's Express system and other OLAP products.

Enhanced Developer Productivity: The functions enable developers to perform complex analyses with much clearer and more concise SQL code. Tasks which in the past required multiple SQL statements or the use of procedural languages can now be expressed using single SQL statements.

Minimized Learning Effort: The syntax leverages existing aggregate functions, such as SUM and AVG, so that these well-understood keywords can be used in extended ways.

Standardized Syntax: The new syntax is part of the ANSI SQL standard. The new analytic functions will be supported by a large number of independent software vendors.

Summary

In this lesson, you should have learned how to:

- **Use grouping sets**
- **Use the new `WITH` clause**
- **Use new analytical functions in Oracle9i**

ORACLE

4-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson you should have learned about the syntax and functionality of various data warehousing functions. You have seen examples of `GROUPING SETS` and the `WITH` clause. You have also seen how to use the Inverse Percentile functions and the `WIDTH_BUCKET`.

Practice 4 Overview

This practice covers the following topics:

- Using grouping sets
- Using the `WITH` clause
- Using Inverse Percentile functions

ORACLE

4-31

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing This Practice

To perform this practice go to appendix A, “Practices.”

ANSI/ISO SQL: 1999 Standard Support in Oracle9i

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Use SQL: 1999 compliant joins**
- **Use `CASE` expressions**
- **Understand scalar subqueries**

ORACLE

5-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

In this lesson, you will learn about the join syntax as well as `CASE` expressions and scalar subqueries.

Overview

Several new features have been included in Oracle9i so that Oracle will be in conformance with ANSI/ISO SQL: 1999.

The most important ones are discussed in this lesson:

- SQL: 1999 join compliance
- Introduction of CASE expressions
- Introduction of scalar subqueries

ORACLE

Note: Throughout this lesson the phrase "SQL: 1999" refers to the SQL: 1999 notations of both ANSI and ISO standards.

Applications for SQL: 1999 Features

- **Allows Oracle SQL to support ANSI/ISO SQL: 1999 standards**
- **Easier migration of third-party applications without modifying existing code**
- **Allows Oracle database to provide ANSI/ISO standard functionality within the database**
- **Easier learning curve for customers moving from other database products**

ORACLE

SQL: 1999 Joins

The SQL: 1999 join syntax differs from Oracle joins in the following ways:

- The join type is specified explicitly in the **FROM** clause in SQL: 1999 syntax.
- The join condition is distinguished from the search condition in the **WHERE** clause and is specified in the **ON** clause.

ORACLE

SQL: 1999 Joins

Oracle joins do not require the **JOIN** keyword whereas the SQL: 1999 standard requires the keyword to be specified. In addition to the join condition, a **WHERE** clause can be specified for other criteria for restricting the rows.

Types of SQL: 1999 Compliant Joins

- **CROSS joins**
- **NATURAL joins**
- **USING clause**
- **FULL OUTER joins**
- **Arbitrary join conditions for joins**

ORACLE

Note: The new SQL: 1999 compliant join syntax does not offer any performance benefits over the Oracle proprietary join syntax.

Creating CROSS Joins

- The **CROSS** join produces the cross-product of two tables
- This is the same as a Cartesian product between the two tables. For example:

```
SELECT last_name, department_name
       FROM employees
       CROSS JOIN departments;
```

ORACLE

5-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating CROSS Joins

The above example gives the same results as :

```
SELECT last_name, department_name FROM employees, departments;
```

LAST_NAME

DEPARTMENT_NAME

-

King
Kochhar
De Haan
Hunold
Ernst

Administration
Administration
Administration
Administration
Administration

...

King
Kochhar
De Haan
Hunold
Ernst

Marketing
Marketing
Marketing
Marketing
Marketing

...

2889 rows selected.

Creating NATURAL Joins

- The NATURAL join is based on all columns in the two tables that have the same name
- It selects rows from the two tables that have equal values in all matched columns
- If the columns having the same names have different datatypes then an error is returned
- If the `SELECT *` syntax is used, then the common columns appear only once in the result set
- Qualifiers such as table names or aliases may not be used for those columns involved in the NATURAL join

ORACLE

5-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating NATURAL Joins

It was not possible to do a join without specifying the columns in the corresponding tables in prior releases of Oracle database. In Oracle9i it is possible to let the join be automatically completed based on columns in the two tables which have matching datatypes and names using the keywords NATURAL JOIN.

Note: The join can only happen on columns having the same names and compatible datatypes in both the tables. If the columns have the same name, but have incompatible datatypes then the NATURAL join syntax will cause errors.

Retrieving Records with NATURAL Joins

```
SELECT
    department_id, employee_id
FROM employees
    NATURAL JOIN departments;
```

DEPARTMENT_ID	EMPLOYEE_ID
20	202
30	119
30	118
30	117
30	116
30	115
...	

ORACLE

Retrieving Records with NATURAL Joins

In the above example the `departments` table is joined to the `employees` table by `manager_id` and `department_id` which are columns of the same name and data type in both tables. If other common columns were present then the join would have used them all. This can also be written as

```
SELECT e.department_id, employee_id
FROM employees e,departments d
    WHERE d.department_id = e.department_id
        AND d.manager_id =e.manager_id;
```

Creating Joins with the USING Clause

- If several columns have the same names but all the data types do not match, then the **NATURAL JOIN** can be modified to contain the **USING** clause to specify those columns that should be used for an equijoin.
- The columns referenced in the **USING** clause should not have a qualifier (table name or alias) anywhere in the SQL statement.
- The keyword **NATURAL** and the **USING** keyword are mutually exclusive.

ORACLE

5-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Joins with the USING Clause

Natural joins use all columns with matching names and data types to join the tables. The **USING** clause can be used to specify only those columns that should be used for an equijoin. The columns referenced in the **USING** clause should not have a qualifier (table name or alias) anywhere in the SQL statement. For example:

```
SELECT l.city, d.department_name
FROM locations l JOIN departments d USING (location_id)
WHERE location_id = 1400 ;
```

Is valid, but:

```
SELECT l.city, d.department_name
FROM locations l JOIN departments d USING (location_id)
WHERE d.location_id = 1400 ;
```

is invalid because in the where clause you have qualified `location_id` with a table name. The same restriction applies to **NATURAL** joins also. Therefore, columns that have the same name in both tables have to be used without any qualifiers.

In both **NATURAL** joins and joins with the **USING** clause, if the columns with the same name are collections or LOBS the join will not be allowed. This behavior is the same in Oracle joins as well.

Retrieving Records with the USING Clause

```
SELECT e.employee_id, e.last_name, d.location_id
      FROM employees e JOIN departments d
        USING (department_id)
      ORDER BY employee_id;
```

EMPLOYEE_ID	LAST_NAME	LOCATION_ID
100	King	1700
101	Kochhar	1700
102	De Haan	1700
103	Hunold	1400
104	Ernst	1400
105	Austin	1400
106	Pataballa	1400
...		

ORACLE

Retrieving Records with the USING Clause

In the above example you are able to join the `employees` table to the `departments` table based only on the common `DEPARTMENT_ID` column. In order for the `USING` clause to work, the column names must be the same in both tables and must have compatible data types. Qualifiers such as aliases or table names cannot be used.

Creating Joins With the ON Clause

- The join condition for the **NATURAL** join is basically an equijoin of all columns with the same name.
- To specify arbitrary conditions or specify columns to join, the **ON** clause is used.
- The **ON** clause separates the join condition from other filter conditions.
- The **ON** clause makes code easy to understand.

ORACLE

5-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Joins With the ON Clause

The **ON** clause is very similar to the **WHERE** clause used in earlier versions of the Oracle database. It allows the specification of additional predicates in addition to the **JOIN** itself. The **JOIN** keyword indicates the join conditions and additional predicates can be specified using the **AND**, **OR** and **NOT** operators.

Example of Retrieving Records with the ON Clause

```
SELECT e.employee_id, e.last_name,
       e.department_id, d.department_id,
       d.location_id
FROM   employees e JOIN departments d
       ON (e.department_id = d.department_id);
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Goyal	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
...				

ORACLE

5-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Retrieving Records with the ON Clause

The ON clause can also be used to join columns which have different names:

```
SELECT e.last_name emp, m.last_name mgr
FROM   employees e JOIN employees m
       ON ( e.manager_id = m.employee_id);
```

EMP	MGR
Kochhar	King
De Haan	King
Raphaely	King
...	
Greenberg	Kochhar
Whalen	Kochhar
...	
Hunold	De Haan
...	

The above example is a self join of the employees table to itself based on the EMPLOYEE_ID and MANAGER_ID columns.

Creating Complex Joins

ON clause allows the use of:

- **Subqueries**
- **AND/OR operators**
- **[NOT] EXISTS**
- **[NOT] IN**

ORACLE

5-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Complex Joins

Similar to the WHERE clause, the ON clause also allows the use of the logical operators as well as subqueries. The only difference between using a WHERE clause and the ON clause is the difference in syntax.

Join Predicates and the ON Clause

- Separating join predicates from the other predicates by using the ON clause makes your code easier to understand.
- The ON clause allows any predicate, including the usage of subqueries and logical operators.

```
SELECT e.manager_id,    e.last_name ,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
       ON((e.department_id = d.department_id)
AND    e.manager_id = 102 );
```

ORACLE

5-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Join Predicates and the ON Clause

You can use the ON clause for any join to separate the join predicates from the nonjoin predicates. In the above example you are trying to retrieve the names, department names, and locations of all employees working for manager 102. The above query can also be written as:

```
SELECT e.manager_id,    e.last_name,
       d.department_id, d.location_id
FROM   employees  e JOIN
       departments d ON
       (e.department_id = d.department_id)
WHERE  e.manager_id = 102;
```

MANAGER_ID	LAST_NAME	DEPARTMENT_ID	LOCATION_ID
102	Hunold	60	1400

Creating Multitable Joins

```
SELECT employee_id, city, department_name
  FROM locations l JOIN departments d
    ON(l.location_id = d.location_id)
  JOIN employees e
    ON(d.department_id = e.department_id);
```

EMPLOYEE_ID	CITY	DEPARTMENT_NAME
100	Seattle	Executive
101	Seattle	Executive
...		
106	Southlake	IT
107	Southlake	IT
108	Seattle	Finance
...		

ORACLE

Creating Multitable Joins

A three way join is a join of three tables. In SQL:1999 compliant syntax, the tables referenced in the join condition are related in the order of the joined tables. In the above example, the first join condition can reference columns in LOCATIONS and DEPARTMENTS but cannot reference columns in employees. The second join condition can reference columns from all three tables.

INNER Versus OUTER Joins

- **IN SQL: 1999** the join of two tables returning only matched rows is considered an **INNER JOIN**.
- A join between two tables that returns the results of the **INNER** join as well as unmatched rows from 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**.

ORACLE

INNER Versus OUTER Joins

Oracle proprietary joins have both **INNER** and **OUTER** join concepts. A join returning matched rows such as an equijoin is an **INNER** join and a join using the (+) operator to return unmatched rows is an **OUTER** join. The Oracle database did not support **FULL OUTER** joins until Oracle9i.

Example of LEFT OUTER Joins

```
SELECT  e.last_name , d.department_name
FROM employees e LEFT OUTER JOIN departments d
      ON(e.department_id = d.department_id;
```

LAST_NAME	DEPARTMENT_NAME
-----	-----
...	
Faviet	Finance
Greenberg	Finance
Gietz	Accounting
Higgins	Accounting
Grant	

ORACLE

Example of LEFT OUTER Joins


This query retrieves all rows in the EMPLOYEES table which is the left table even if there is no match in DEPARTMENTS .

This query can also be completed as follows:

```
SELECT  e.last_name, d.department_name
FROM employees e, departments d
      WHERE e.department_id = d.department_id(+);
```

Example of RIGHT OUTER Joins

```
SELECT  e.last_name , d.department_name
FROM    employees e RIGHT OUTER JOIN departments d
        ON(e.department_id = d.department_id);
```

LAST_NAME	DEPARTMENT_NAME
-----	-----
...	
Baer	Public Relations
Higgins	Accounting
Gietz	Accounting
	NOC
	Manufacturing
	Construction
	Control And Credit
...	

ORACLE

Example of RIGHT OUTER Joins

This query retrieves all rows in the DEPARTMENTS table which is the right table even if there is no match in the EMPLOYEES table

This query can also be completed as follows:

```
SELECT e.last_name, d.department_name
FROM employees e, departments d
WHERE e.department_id(+) = d.department_id;
```

Example of FULL OUTER Joins

```
SELECT  e.employee_id ,  d.department_name
FROM employees e FULL OUTER JOIN departments d
ON(e.department_id = d.department_id);
```

EMPLOYEE_ID	DEPARTMENT_NAME
...	...
109	Finance
108	Finance
206	Accounting
205	Accounting
178	
	NOC
	Manufacturing
	Construction
...	...

ORACLE

Example of FULL OUTER Joins

This query retrieves all rows in both tables regardless of whether there is a match in the other.

It was not possible to complete this in earlier releases using outer joins. However, you could accomplish similar results by using the UNION operator as follows:

```
SELECT  e.employee_id ,  d.department_name
FROM employees e, departments d
WHERE e.department_id = d.department_id(+)
UNION
SELECT  e.employee_id ,  d.department_name
FROM employees e, departments d
WHERE e.department_id(+) = d.department_id;
```


Example of Using Subqueries

```
SELECT d.department_name, l.city
FROM locations l JOIN departments d
ON(l.location_id = d.location_id)
    AND d.department_id IN
        (SELECT e.department_id
         FROM employees e );
```

DEPARTMENT_NAME	CITY
-----	-----
Administration	Seattle
Marketing	Toronto
Purchasing	Seattle
Human Resources	London
...	
Finance	Seattle
Accounting	Seattle
11 rows selected.	

ORACLE

Example of Using Subqueries

In the above example the LOCATIONS table is joined to the DEPARTMENTS table which retrieves all matched rows in both tables. Additionally, through the use of a subquery, the output is restricted to only those departments which contain employees. Therefore, only 11 rows are retrieved versus the 27 which would have been retrieved through a join on these tables without the subquery.

```
SELECT d.department_name, l.city
FROM locations l JOIN departments d
ON(l.location_id = d.location_id);
```

DEPARTMENT_NAME	CITY
-----	-----
Administration	Seattle
Marketing	Toronto
Purchasing	Seattle
Human Resources	London
...	
27 rows selected.	

Example of Join Using NOT EXISTS

```
SELECT department_name , city
FROM locations l JOIN departments d
ON (l.location_id = d.location_id)
AND NOT EXISTS (select 1 from employees e
WHERE e.department_id IN d.department_id);
```

DEPARTMENT_NAME	CITY
-----	-----
Treasury	Seattle
Corporate Tax	Seattle
Control And Credit	Seattle
...	
Retail Sales	Seattle
Recruiting	Seattle
Payroll	Seattle

ORACLE

Example of Join Using NOT EXISTS

In the above example the departments and cities are retrieved by joining DEPARTMENTS to LOCATIONS based on the LOCATION_ID column. However, the output is restricted to those departments without employees all of which are in Seattle.

CASE Expressions in SQL: 1999

SQL has the following types of CASE statements:

- **Simple CASE expression**
- **Searched CASE expression**
- **NULLIF**
- **COALESCE**

ORACLE

5-23

Copyright © Oracle Corporation, 2001. All rights reserved.

CASE Expressions in SQL: 1999

The SQL: 1999 standard has four types of CASE expressions:

- Simple CASE expression
- Searched CASE expressions
- NULLIF
- COALESCE

The simple CASE expression was available in Oracle8i release 2 (8.1.6). All the other expressions have been included in Oracle9i.

Note: The CASE expression is supported in PL/SQL in Oracle9i.

Simple CASE Expression

- The simple CASE expression is similar to the `DECODE` function.
- It can be used to search and replace values within a given expression.
- For each searched value a return value can be specified.
- Comparison operators are not allowed.

ORACLE

Simple CASE Expression

A simple CASE expression works in a similar way to a `DECODE` function and can be used to search and replace values within a given expression. The simple CASE expression does not support any comparison operators, and for every searched value a return value can be specified.

Simple CASE Expression: Example

```
SELECT last_name,  
       CASE department_id  
         WHEN 10 THEN 'Administration'  
         WHEN 20 THEN 'Marketing'  
         WHEN 30 THEN 'Purchasing'  
         WHEN 40 THEN 'Human Resources'  
         ELSE 'N/A'  
       END as "Department Names"  
FROM employees  
ORDER by department_id;
```

ORACLE

5-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Simple CASE Expression: Example

In the above example the department ID is evaluated to return the department names. The ELSE keyword is used to specify an alternative if none of the conditions are matched. "Department Names" is an alias for the entire CASE expression.

The results of the above query are as follows:

LAST_NAME	Department Name
-----	-----
Whalen	Administration
Hartstein	Marketing
Goyal	Marketing
Raphaely	Purchasing
Khoo	Purchasing
Baida	Purchasing
Colmenares	Purchasing
Himuro	Purchasing
Tobias	Purchasing
Marvis	Human Resources
...	
Grant	N/A

Searched CASE Expression

- The searched CASE expression is similar to an IF... THEN ... ELSE construct.
- It can conditionally be used to search and replace values within expressions.
- Each WHEN condition can be different and multiple conditions can be combined with logical operators.
- Comparison operators are allowed in the conditional expression.
- Searched CASE expressions are more flexible than simple CASE expressions.

ORACLE

5-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Searched CASE Expression

The searched CASE statement is very similar to an IF...THEN ... ELSE construct. Each searched CASE statement can evaluate multiple conditions and each condition can reference different variables and operators. Multiple conditions may be combined through the use of logical operators. The searched CASE statement provides more flexible usage than the simple CASE statement.

Searched CASE Expression: Example

```
INSERT INTO raise
  SELECT last_name,
         CASE
           WHEN job_id LIKE 'AD%' THEN '10%'
           WHEN job_id LIKE 'IT%' THEN '15%'
           WHEN job_id LIKE 'PU%' THEN '18%'
           ELSE '20%'
         END
  FROM employees;
```

ORACLE

5-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Searched CASE Expression

In the above example you are trying to determine the raise for an employee based on job_id. If job_id is anything starting with AD, a raise of 10% is returned. For any job_id starting with IT, a raise of 15% is returned, and so on. The RAISE table is created as follows:

```
CREATE TABLE raise( name VARCHAR2(30), raise_pct
VARCHAR2(10) );
```

The contents of the table following the INSERT is as follows:

```
SELECT * FROM raise;
```

NAME	RAISE_PCT
-----	-----
King	10%
Kochhar	10%
De Haan	10%
...	

SQL NULLIF and COALESCE

- The **NULLIF** function returns **NULL** if the first argument is equal to the second. Otherwise, the value of the first argument is returned.
- The **COALESCE** function is a generalization of Oracle's **NVL** function.
 - Accepts any number of arguments
 - Returns the first **NOT NULL** argument

ORACLE

5-28

Copyright © Oracle Corporation, 2001. All rights reserved.

NULLIF and COALESCE

The **NULLIF** function returns **NULL** if the first argument is equal to the second. Otherwise, the value of the first argument is returned.

The **COALESCE** function is a generalization of the **NVL** function: it accepts any number of arguments, and returns the first **NOT NULL** argument.

NULLIF and COALESCE

Both can be written as CASE expressions:

```
NULLIF(expr1,expr2) ⇔  
CASE WHEN expr1 = expr2  
      THEN NULL  
      ELSE expr1  
END
```

```
COALESCE(expr1,expr2,expr3,...) ⇔  
CASE WHEN expr1 IS NOT NULL  
      THEN expr1  
      ELSE COALESCE(expr2,expr3,...)  
END
```

ORACLE

5-29

Copyright © Oracle Corporation, 2001. All rights reserved.

NULLIF Expression

The semantics of the SQL NULLIF expression are defined by rewriting it as:

```
CASE  
  WHEN expression 1 = expression 2 THEN NULL  
  ELSE expression 1  
END
```

COALESCE Expression

When a SQL COALESCE expression has exactly two arguments, the semantics are defined by rewriting it as:

```
CASE  
  WHEN expression 1 IS NOT NULL THEN expression 1  
  ELSE expression 2  
END
```

When COALESCE has three or more arguments, it has the form COALESCE (expression 1, expression 2, expression 3, . . . , expression n), the semantics are defined by rewriting it as:

```
CASE  
  WHEN expression1 IS NOT NULL THEN expression 1  
  ELSE COALESCE (expression 2, expression 3, . . . ,expression  
n)  
END
```

SQL NULLIF Expression: Example

A

```
SELECT employee_id id,  
       commission_pct comm1  
FROM employees  
ORDER BY id;
```

ID	COMM1
-----	-----
...	
161	.25
162	.25
163	.15
164	.1
165	.1
...	

B

```
SELECT employee_id id,  
       NULLIF(commission_pct,0.1)  
       comm2 FROM employees  
ORDER BY id;
```

ID	COMM2
-----	-----
...	
161	.25
162	.25
163	.15
164	
165	
...	

ORACLE

SQL NULLIF Expression: Example

The above example shows two queries. Query A retrieves ID and commission percentages for all employees, sorting by ascending order of commission percentage. Query B retrieves employee IDs and the commission percentages, but the commission percentages that are .1 are converted to NULL.

SQL COALESCE Expression: Example

A

```
SELECT last_name,
       commission_pct comm1
FROM employees;
```

LAST_NAME	COMM1
-----	-----
...	
Grant	.15
Johnson	.1
Taylor	
Fleaur	
...	

B

```
SELECT last_name, COALESCE
(TO_CHAR
 (commission_pct), 'n/a')
       comm2
FROM employees;
```

LAST_NAME	COMM2
-----	-----
...	
Grant	.15
Johnson	.1
Taylor	n/a
Fleaur	n/a
...	

ORACLE

5-31

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL COALESCE Expression: Example

The above example shows two queries. Query A retrieves employee last names and the commission percentage and shows that some employees are noncommissioned employees, as the commission percentage is NULL. Query B retrieves the last names and COALESCE commission percentage. The COALESCE function can only return similar data types so therefore the commission percentage is first converted to a character expression and then COALESCED to return n/a. The TO_CHAR conversion is not required if a numeric substitution value is specified.

The semantics of the above COALESCE operation is the same as:

```
SELECT
  ( CASE
    WHEN TO_CHAR(commission_pct) IS NOT NULL THEN
      TO_CHAR( commission_pct)
    ELSE ' n/a'
    END ) comm2
FROM employees;
```

COALESCE with Multiple Expressions: Example

- The advantage of using COALESCE over the NVL function is that the COALESCE function can take multiple alternate values.
- If the first expression is NOT NULL then it returns that expression. Otherwise it does a COALESCE of the remaining expressions. For example:

```
SELECT last_name, salary, commission_pct,
       COALESCE
         (salary*(1+ commission_pct), salary, 0)
  compensation
  FROM employees
 ORDER BY commission_pct;
```

ORACLE

5-32

Copyright © Oracle Corporation, 2001. All rights reserved.

COALESCE with Multiple Expressions: Example

The above query returns the following output.

LAST_NAME	SALARY	COMMISSION_PCT	COMPENSATION
Sully	9500	.35	12825
Russell	14000	.4	19600
King	24000		24000
Kochhar	17000		17000
De Haan	17000		17000
Hunold	9000		9000
Ernst	6000		6000 ...

The semantics of this query are the same as:

```
SELECT last_name,
       (CASE WHEN salary*(1+commission_pct) IS NOT NULL THEN
            salary*(1+commission_pct)
          WHEN salary IS NOT NULL then salary
          ELSE 0 END) compensation
  FROM employees ORDER BY commission_pct;
```

Scalar Subqueries in SQL: 1999

- The scalar subquery is used to specify a scalar value derived from a query expression.
- Scalar subqueries were supported in Oracle8i only in a limited set of cases.
- In Oracle9i scalar subqueries can be used in the **WHERE** clause, **SELECT** list, or in any place where a valid expression can be used.
- A scalar subquery can return only one value.
- The data type of the return value matches the value being selected in the subquery.

ORACLE

5-33

Copyright © Oracle Corporation, 2001. All rights reserved.

Rules for Scalar Subqueries

The rules for Scalar subqueries are as follows:

- The degree of the scalar subquery is one.
- The type of the scalar subquery is the type of the column of subquery.
- If the cardinality of the scalar subquery is greater than 1, it results in an error.
- If the cardinality of the scalar-subquery is 0, the value of the scalar subquery is **NULL**.
- Assuming the cardinality to be 1, the value of the scalar subquery is the value of the column in the unique row of the subquery result.

Note: Scalar query operations are very resource intensive and should be avoided if another type of query is available to retrieve the desired results.

Using Scalar Subqueries

Scalar subqueries can be used:

- In all clauses of **SELECT** except **GROUP BY**
- In the **VALUES** clause of **INSERT** statement
- In the **SET** clause and **WHERE** clause of **UPDATE** statement
- In condition and expression part of **DECODE** and **CASE**

ORACLE

5-34

Copyright © Oracle Corporation, 2001. All rights reserved.

Usage of Scalar Subqueries

Scalar subqueries are not supported in the following:

- Default values for column
- Returning clauses
- Hash expressions for clusters
- Function based index expression
- Check constraints on columns
- **WHEN** condition of triggers
- **GROUP BY** clauses

Scalar subqueries can also negatively impact performance as they are as resource intensive as other forms of subqueries such as correlated subqueries.

Note: The usage of scalar subqueries in **SET** clause of an **UPDATE** statement and **VALUES** list of an **INSERT** statement was supported in Oracle8i .

Scalar Subqueries in SELECT List

The scalar subquery expression in Oracle SQL is extended to support subqueries as SELECT list items. For example:

```
SELECT employee_id, last_name, (  
    SELECT department_name  
    FROM departments d  
    WHERE  
        e.department_id = d.department_id  
    ) department_name  
FROM employees e  
ORDER BY department_name;
```

ORACLE

5-35

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries in SELECT List

In the above example employee ID, last names and department names are retrieved by using a subquery in the select list for retrieving the department names from the DEPARTMENTS table. Department_name is an alias for the results of the subquery.

The results of the above example are as follows:

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_NAME
205	Higgins	Accounting
206	Gietz	Accounting
200	Whalen	Administration
100	King	Executive
101	Kochhar	Executive
102	De Haan	Executive
...		
103	Hunold	IT
104	Ernst	IT
107	Lorentz	IT
201	Hartstein	Marketing
202	Goyal	Marketing
...		

Scalar Subqueries in the WHERE Clause

Scalar subqueries in Oracle SQL: 1999 can also be in the left side of the WHERE clause condition. For example:

```
SELECT employee_id, last_name
FROM employees e
WHERE ((SELECT location_id
        FROM departments d
        WHERE e.department_id = d.department_id )
       = (SELECT location_id
        FROM locations l
          WHERE state_province =
            'California') );
```

ORACLE

5-36

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries in the WHERE Clause

In the above example, you are retrieving the employee IDs and names of all employees located in California. The results of the above example are as follows:

EMPLOYEE_ID LAST_NAME

...

194 McCain

195 Jones

196 Walsh

197 Feeney

198 OConnell

199 Grant

Scalar Subqueries in the ORDER BY Clause

Scalar Subqueries can now also be used in the ORDER BY clause. For example:

```
SELECT employee_id, last_name
FROM employees e
ORDER BY
  (SELECT department_name
   FROM departments d
   WHERE e.department_id = d.department_id);
```

ORACLE

5-37

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries in the ORDER BY Clause

In the above example, you are retrieving employee IDs and last names by sorting the output by department_name. The results of the above example are as follows:

EMPLOYEE_ID LAST_NAME

...

138 Stiles

139 Seo

140 Patel

141 Rajs

142 Davies

143 Matos

144 Vargas

178 Grant

Scalar Subqueries in CASE Expressions

Scalar subqueries can be used in both the condition and the expression part of `DECODE` and `CASE` expressions. For example:

```
SELECT employee_id, last_name ,
(CASE
  WHEN department_id IN
    (SELECT department_id FROM departments
     WHERE location_id = 1800)
  THEN 'Canada' ELSE 'other' END) location
FROM employees;
```

ORACLE

5-38

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries in CASE Expressions

In the above example, you are retrieving the `employee_id` , `last_name` and locations of employees. For employees in location 1800 you display Canada and you display "other" for all others. Location is an alias for the `CASE` expression. The result of the above example is:

EMPLOYEE_ID	LAST_NAME	LOCATION
...		
199	Grant	other
200	Whalen	other
201	Hartstein	Canada
202	Fay	Canada
203	Mavris	other
204	Baer	other
205	Higgins	other
206	Gietz	other

Scalar Subqueries in Functions

Scalar subqueries can be used as arguments to Built-in functions, user defined functions, and type constructors. For example:

```
SELECT last_name, SUBSTR (
    (SELECT department_name
     FROM departments d
     WHERE d.department_id
           =e.department_id),
    1, 10) department
FROM employees e;
```

ORACLE

5-39

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries in Functions

In the above example, you are retrieving the last names and the first 10 characters of the department name using the SUBSTR function on a subquery. Department is an alias for the results of the SUBSTR function. The results of the above example are as follows:

LAST_NAME	DEPARTMENT
King	Executive
Kochhar	Executive
De Haan	Executive
Hunold	IT
Ernst	IT
...	
Mourgos	Shipping
...	

Summary

In this lesson, you should have learned how to:

- **Create SQL: 1999 compliant joins**
- **Create SQL CASE expressions**
- **Create scalar subqueries**

ORACLE

5-40

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

This lesson covered the additions to Oracle SQL syntax which is in conformance with the SQL: 1999 standard. You should have learned how to create joins using the new join syntax. You can see that it is now possible to create `FULL OUTER JOINS` and `NATURAL` joins. You can identify the syntax and functionality of the CASE expressions including `NULLIF` and `COALESCE`. The usage of scalar subqueries has also been explained.

Practice 5 Overview

This practice covers the following topics:

- **Creating NATURAL joins**
- **Creating FULL OUTER joins**
- **Using Scalar subqueries**

ORACLE

5-41

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing This Practice

To perform this practice go to Appendix A, “Practices.”

6 Datetime Enhancements

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Describe the datetime enhancements**
- **Use the new data types for datetime and interval**
- **Implement time zone with offsets and named regions**
- **Discuss daylight savings time (DST) with the datetime data types**
- **Use the new built-in SQL functions for datetime and interval**

ORACLE

6-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

In this lesson you will learn about the datetime data types, the new time zones and intervals, and Unicode support.

Note: For more information on globalization features in Oracle9i please refer to the Globalization reference manual

TIME_ZONE Session Parameter

TIME_ZONE may be set to:

- O/S local time zone
- An absolute offset
- Database time zone
- A named region

```
ALTER SESSION SET TIME_ZONE = '-05:00';  
ALTER SESSION SET TIME_ZONE = dbtimezone;  
ALTER SESSION SET TIME_ZONE = local;  
ALTER SESSION SET TIME_ZONE =  
'America/New_York';
```

ORACLE

TIME_ZONE Session Parameter

The ALTER SESSION command can be used to change time zone values in a users session. The time zone values can be set to an absolute offset, a named time zone, or a database time zone, or the local time zone.

Datetime and Interval Data types

- Most datetime and interval data types are compliant with SQL :1999 standards
- `TIMESTAMP WITH LOCAL TIME ZONE` is the only Oracle proprietary data type
- Eases development of applications accessed globally
- Enables display of datetime at user's session time zone
- Eases development of applications requiring sub-second precision

ORACLE

6-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Datetime and Interval Data Types

All datetime and interval data types, except `TIMESTAMP WITH LOCAL TIME ZONE`, are compliant with SQL:1999 standards. SQL:1999 defines a total of 18 data types for datetimes and intervals. There are five datetime types: `DATE`, `TIME`, `TIME WITH TIME ZONE`, `TIMESTAMP`, and `TIMESTAMP WITH TIME ZONE`.

Oracle9i defines an additional data type, `TIMESTAMP WITH LOCAL TIME ZONE`. There are also 13 interval types in the SQL:1999 standard divided into two classes. Oracle9i only defines two interval types, `INTERVAL YEAR TO MONTH` and `INTERVAL DAY TO SECOND`.

These data types are supported as columns in tables as well as parameter, variables or function return values in PL/SQL.

Note: The surface of the earth is divided into zones, called time zones, in which every correct clock tells the same time, known as local time. Local time is equal to Coordinated Universal Time (UTC) (previously called Greenwich mean time, or GMT) plus the time zone displacement, which is a value of `INTERVAL HOUR TO MINUTE`, between - 12:59 and +13:00.

Datetime Data Types

Data type	Fields
TIMESTAMP	Year, Month, Day, Hour, Minute, Second with fractional seconds
TIMESTAMP WITH TIME ZONE	Year, Month, Day, Hour, Minute, Second with fractional seconds, TimeZone_Hour, and TimeZone_Minute or TimeZone_Region
TIMESTAMP WITH LOCAL TIME ZONE	Year, Month, Day, Hour, Minute, Second with fractional seconds

ORACLE

6-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Datetime Data Types

TIMESTAMP (fractional_seconds_ precision)

This data type contains the year, month, and day values of date, as well as hour, minute, and second values of time, where significant fractional seconds precision is the number of digits in the fractional part of the SECOND datetime field. Accepted values of significant fractional_seconds_precision are 0 to 9. The default is 6.

TIMESTAMP (fractional_seconds_precision) WITH TIME ZONE

This data type contains all values of TIMESTAMP as well as time zone displacement value, where significant fractional seconds precision is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

TIMESTAMP (fractional_seconds_precision) WITH LOCAL TIME ZONE

This data type contains all values of TIMESTAMP, with the following exceptions:

- Data is normalized to the database time zone when it is stored in the database.
- When the data is retrieved, users see the data in the session time zone.

Datetime Fields

Datetime Field	Valid Values
YEAR	-4712..9999 (excluding year 0)
MONTH	01 to 12
DAY	01 to 31
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 59.9(N) where 9(N) is precision
TIMEZONE_HOUR	-12 to 14
TIMEZONE_MINUTE	00 to 59

ORACLE®

Datetime Fields

Each datetime data type is composed of several of these fields. Datetimes are mutually comparable and assignable only if they have the same datetime fields.

TIMESTAMP Data Type

- The **TIMESTAMP** data type is an extension of the **DATE** data type.
- It stores the year, month, and day of the **DATE** data type, plus hour, minute, and second values as well as the fractional second value.
- Variations in **TIMESTAMP** are:
 - **TIMESTAMP[(fractional_seconds_precision)]_**
 - **TIMESTAMP[(fractional_seconds_precision)]_WITH TIME ZONE**
 - **TIMESTAMP[(fractional_seconds_precision)]_WITH LOCAL TIME ZONE**

ORACLE

Datetime Data Types

TIMESTAMP data type contains the datetime fields **YEAR**, **MONTH**, **DAY**, **HOURL**, **MINUTE**, and **SECOND**.

TIME WITH TIME ZONE data type contains the datetime fields **HOURL**, **MINUTE**, **SECOND**, **TIMEZONE_HOUR**, and **TIMEZONE_MINUTE**.

TIMESTAMP WITH TIME ZONE data type contains the datetime fields **YEAR**, **MONTH**, **DAY**, **HOURL**, **MINUTE**, **SECOND**, **TIMEZONE_HOUR**, and **TIMEZONE_MINUTE**.

Note: Fractional second precision specifies the number of digits in the fractional part of the **SECOND** datetime field and can be a number in the range 0 to 9. The default is 6.

Difference between DATE and TIMESTAMP

A

```
-- when hire_date is  
of type DATE  
  
SELECT hire_date  
FROM employees;  
  
HIRE_DATE  
-----  
17-JUN-87  
21-SEP-89  
13-JAN-93  
03-JAN-90  
21-MAY-91  
...
```

B

```
ALTER TABLE employees  
MODIFY hire_date TIMESTAMP;  
  
SELECT hire_date  
FROM employees;  
  
HIRE_DATE  
-----  
17-JUN-87 12.00.00.000000 AM  
21-SEP-89 12.00.00.000000 AM  
13-JAN-93 12.00.00.000000 AM  
03-JAN-90 12.00.00.000000 AM  
21-MAY-91 12.00.00.000000 AM  
...
```

ORACLE

6-8

Copyright © Oracle Corporation, 2001. All rights reserved.

TIMESTAMP Data Type: Example

In the above slide, example A shows the data from the `hire_date` column of the `employees` table when the data type of the column is `DATE`. In example B the table is altered and the data type of the `hire_date` column is made into `TIMESTAMP`. The output shows the differences in display. It is possible to convert from `DATE` to `TIMESTAMP` when the column has data, but it is not possible to convert from `DATE` or `TIMESTAMP` to `TIMESTAMP WITH TIME ZONE` unless the column is empty.

It is possible to specify the fractional seconds precision for timestamp. If none is specified as in the above example then it defaults to 6.

For example, the following statement sets the fractional seconds precision as 7:

```
ALTER TABLE employees  
MODIFY hire_date TIMESTAMP(7);
```

TIMESTAMP WITH TIME ZONE Data Type

- **TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time zone displacement in its value.**
- **The time zone displacement is the difference, in hours and minutes, between local time and UTC.**
- **It is specified as:**

```
TIMESTAMP[(fractional_seconds_precision)]  
WITH TIME ZONE
```

ORACLE

6-9

Copyright © Oracle Corporation, 2001. All rights reserved.

TIMESTAMP WITH TIME ZONE Data Type

UTC stand for Coordinated Universal Time (formerly Greenwich Mean Time). Two **TIMESTAMP WITH TIME ZONE** values are considered identical if they represent the same instant in UTC, regardless of the **TIME ZONE** offsets stored in the data. For example:

```
TIMESTAMP '1999-04-15 8:00:00 -8:00'
```

is the same as

```
TIMESTAMP '1999-04-15 11:00:00 -5:00'
```

That is, 8:00 a.m. Pacific Standard Time is the same as 11:00 a.m. Eastern Standard Time.

This can also be specified as:

```
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
```

Note: **Fractional_seconds_precision** optionally specifies the number of digits in the fractional part of the **SECOND** datetime field and can be a number in the range 0 to 9. The default is 6.

TIMESTAMP WITH TIME ZONE: Example

```
CREATE TABLE web_orders (order_date TIMESTAMP WITH  
TIME ZONE);
```

```
INSERT INTO web_orders SELECT order_date FROM  
orders;
```

```
SELECT * FROM web_orders;  
ORDER_DATE
```

```
-----  
16-AUG-99 02.34.12.234359 PM -04:00  
19-NOV-99 03.41.54.696211 PM -04:00  
02-OCT-99 04.49.34.678340 PM -04:00  
14-JUL-00 05.18.23.234567 PM -04:00  
...
```

ORACLE

6-10

Copyright © Oracle Corporation, 2001. All rights reserved.

TIMESTAMP WITH TIME ZONE: Example

In the above example a new table `date_tab` is created with a column of data type `TIMESTAMP WITH TIME ZONE`. This table is populated by selecting the data from the `hire_date` column of `employees` table. The output from the `date_tab` table shows the offset from UTC which is `-8` hours for all these records.

TIMESTAMP WITH TIME ZONE: Example

```
DECLARE
  V_logon TIMESTAMP(3) WITH TIME ZONE;
  V_return v_logon%type;
BEGIN
  V_logon := sysdate;
  INSERT INTO date_tab values(v_logon) returning
    date_col INTO v_return ;
  DBMS_OUTPUT.PUT_LINE(v_return);
END;
/
15-MAY-01 12.10.07.000 PM -07:00
PL/SQL procedure successfully completed.
```

ORACLE

6-11

Copyright © Oracle Corporation, 2001. All rights reserved.

TIMESTAMP WITH TIME ZONE: Example

In the above example, you declare a variable of type `TIMESTAMP WITH TIME ZONE`, then assign a literal value to it. In this example, the time-zone displacement is `+02:00`. You can also specify the time zone by using a symbolic name. The specification can include a long form such as `US/Pacific`, a short form such as `PDT`, or a predefined data types combination. For example, the following literals all represent the same time. The third form is most reliable because it specifies the rules to follow at the point when switching to daylight savings time.

```
TIMESTAMP '1999-04-15 8:00:00 -8:00'
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
TIMESTAMP '1999-10-31 01:30:00 US/Pacific PDT'
```

TIMESTAMP WITH LOCAL TIMEZONE

- **TIMESTAMP WITH LOCAL TIME ZONE is another variant of TIMESTAMP that includes a time zone displacement in its value.**
- **Data stored in the database is normalized to the database time zone.**
- **The time zone displacement is not stored as part of the column data.**
- **Oracle database returns the data in the users' local session time zone.**
- **The TIMESTAMP WITH LOCAL TIME ZONE data type is specified as follows:**

```
TIMESTAMP[(fractional_seconds_precision)]  
WITH LOCAL TIME ZONE
```

ORACLE

TIMESTAMP WITH LOCAL TIMEZONE

Unlike **TIMESTAMP WITH TIME ZONE**, you can specify columns of type **TIMESTAMP WITH LOCAL TIME ZONE** as part of a primary or unique key. The time zone displacement is the difference (in hours and minutes) between local time and UTC. There is no literal for **TIMESTAMP WITH LOCAL TIME ZONE**.

TIMESTAMP WITH LOCAL TIME ZONE: Example

```
CREATE TABLE shipping (delivery_time TIMESTAMP WITH  
LOCAL TIME ZONE);  
INSERT INTO shipping VALUES('15-NOV-00 09:34:28 AM');
```

```
SELECT * FROM shipping;  
DELIVERY_TIME  
-----  
15-NOV-00 09.34.28.000000 AM
```

```
ALTER SESSION SET TIME_ZONE = 'EUROPE/LONDON';  
SELECT * FROM shipping;  
DELIVERY_TIME  
-----  
15-NOV-00 02.34.28.000000 PM
```

ORACLE

TIMESTAMP WITH LOCAL TIME ZONE: Example

In the above example a new table `date_tab` is created with a column of the data type `TIMESTAMP WITH LOCAL TIME ZONE`. This table is populated by inserting a record into it. The output from the `DATE_TAB` table shows that the data is stored without the timezone offset. Then the `ALTER SESSION` command is issued to change the time zone of the local session. A second query on the same table now reflects the data with the local time zone reflected in the time value.

INTERVAL Data Types

- **INTERVAL data types are used to store the difference between two datetime values**
- **There are two classes of intervals**
 - Year-month
 - Day-time
- **The precision of the interval is:**
 - The actual subset of fields that comprise an interval
 - Specified in the interval qualifier

Datatype	Fields
INTERVAL YEAR TO MONTH	Year, Month
INTERVAL DAY TO SECOND	Days, Hour, Minute, Second with fractional seconds

ORACLE

6-14

Copyright © Oracle Corporation, 2001. All rights reserved.

INTERVAL Data Types

INTERVAL data types are used to store the difference between two datetime values. There are two classes of intervals: year-month intervals and day-time intervals. A year-month interval is made up of a contiguous subset of fields YEAR, MONTH, whereas a day-time interval is made up of a contiguous subset of fields (DAY, HOUR, MINUTE, and SECOND). The actual subset of fields that comprise an interval is called the precision of the interval and is specified in the interval qualifier. Because the number of days in a year are calendar dependent, the year-month interval is NLS dependent whereas day-time interval is NLS independent.

The interval qualifier may also specify the leading field precision, which is the number of digits in the leading or only field, and in case the trailing field is SECOND, it may also specify the fractional seconds precision, which is the number of digits in the fractional part of the SECOND value. If not specified, the default value for leading field precision is 2 digits and the default value for fractional seconds precision is 6 digits.

INTERVAL YEAR (year_precision) TO MONTH

Stores a period of time in years and months, where year_precision is the number of digits in the YEAR datetime field. Accepted values are 0 to 9. The default is 6.

INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision)

Stores a period of time in days, hours, minutes, and seconds, where day_precision is the maximum number of digits in the DAY datetime field (accepted values are 0 to 9; the default is 2), and fractional_seconds_precision is the number of digits in the fractional part of the SECOND field. Accepted values are 0 to 9. The default is 6.

INTERVAL Fields

INTERVAL Field	Valid Values for Interval
YEAR	Any positive or negative integer
MONTH	00 to 11
DAY	Any positive or negative integer
HOURL	00 to 23
MINUTE	00 to 59
SECOND	00 to 59.9(N) where 9(N) is precision

ORACLE

6-15

Copyright © Oracle Corporation, 2001. All rights reserved.

INTERVAL Fields

INTERVAL YEAR TO MONTH can have fields of YEAR and MONTH.

INTERVAL DAY TO SECOND can have fields of DAY, HOUR, MINUTE and SECOND.

The actual subset of fields that comprise an item of either type of interval is defined by an interval qualifier and this subset is known as the precision of the item.

Year-month intervals are mutually comparable and assignable only with other year-month intervals, and day-time intervals are mutually comparable and assignable only with other day-time intervals.

INTERVAL YEAR TO MONTH Data Type

- **INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields.**

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

- **For Example:**

```
'312-2' assigned to INTERVAL YEAR(3) TO MONTH  
Indicates an interval of 312 years and 2 months
```

```
'312-0' assigned to INTERVAL YEAR(3) TO MONTH  
Indicates 312 years and 0 months
```

```
'0-3' assigned to INTERVAL YEAR TO MONTH  
Indicates an interval of 3 months
```

ORACLE

INTERVAL YEAR TO MONTH Data Type

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. Specify INTERVAL YEAR TO MONTH as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

Where year_precision is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

Restriction: The leading field must be more significant than the trailing field. For example, INTERVAL '0-1' MONTH TO YEAR is not valid.

The following INTERVAL YEAR TO MONTH literal indicates an interval of 123 years, 3 months:

```
INTERVAL '123-3' YEAR(3) TO MONTH
```

```
INTERVAL '123' YEAR(3) indicates an interval of 123 years 0 months.
```

```
INTERVAL '3' MONTH indicates an interval of 3 months.
```

INTERVAL YEAR TO MONTH: Example

```
CREATE TABLE warranty
( warranty_time INTERVAL YEAR(3) TO MONTH);
INSERT INTO warranty VALUES (INTERVAL '8' MONTH);
INSERT INTO warranty VALUES (INTERVAL '200'
YEAR(3));
INSERT INTO warranty VALUES ('200-11');
SELECT warranty_time FROM warranty;
WARRANTY_TIME
-----
+000-08
+200-00
+200-11
```

ORACLE

6-17

Copyright © Oracle Corporation, 2001. All rights reserved.

INTERVAL YEAR TO MONTH Data Type

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. Specify INTERVAL YEAR TO MONTH as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

where year_precision is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

Restriction: The leading field must be more significant than the trailing field. For example, INTERVAL '0-1' MONTH TO YEAR is not valid.

Oracle9i Database only supports two interval data types: Interval Year to Month and Interval Day to Second; column type, PL/SQL argument, variable and return type must be one of the two. However, for interval literals the system recognizes other ANSI interval types like INTERVAL '2' YEAR or INTERVAL '10' HOUR. In these cases each interval is converted to one of the two supported types.

In the above example, a WARRANTY table is created which contains a warranty_time column that takes the data type INTERVAL YEAR(3) TO MONTH. Different values are inserted into it to indicate years and months. When this row is retrieved from the table, it shows a year value displaced by the month value by a (-).

INTERVAL YEAR TO MONTH Data Type: Example

```
DECLARE
v_warranty INTERVAL YEAR(3) TO MONTH;
BEGIN
v_warranty := INTERVAL '2' YEAR;
DBMS_OUTPUT.PUT_LINE(v_warranty);
END;
/
+002-00
PL/SQL procedure successfully completed.
```

ORACLE

INTERVAL YEAR TO MONTH Data Type: Example

Oracle9i Database only supports two interval data types: INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND; column type, PL/SQL argument, variable and return type must be one of the two. However, for interval literals the system recognizes other ANSI interval types like INTERVAL '2' YEAR or INTERVAL '10' HOUR. In these cases each interval is converted to one of the two supported types.

In the above example, you declare a variable of type INTERVAL YEAR TO MONTH, then assign a literal value to it. In this example, the interval is 2 years and 0 months. When you output this value, it shows the year value displaced by the month value by a minus sign(-).

INTERVAL DAY TO SECOND Data Type

- **INTERVAL DAY TO SECOND**
(*fractional_seconds_precision*) stores a period of time in days, hours, minutes, and seconds.

```
INTERVAL DAY[(day_precision)] TO Second
```

- **For Example:**

```
INTERVAL '6 03:30:16' DAY TO SECOND
```

Indicates an interval of 6 days 3 hours 30 minutes and 16 seconds

```
INTERVAL '6 00:00:00' DAY TO SECOND
```

Indicates an interval of 6 days and 0 hours, 0 minutes and 0 seconds

ORACLE

INTERVAL DAY TO SECOND Data Type

INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision) stores a period of time in days, hours, minutes, and seconds, where day_precision is the maximum number of digits in the DAY datetime field (accepted values are 0 to 9; the default is 2), and fractional_seconds_precision is the number of digits in the fractional part of the SECOND field. Accepted values are 0 to 9. The default is 6.

In the above example, 6 represents the number of days and 03:30:15 indicates the values for hours, minutes, and seconds.

INTERVAL DAY TO SECOND Data Type: Example

```
CREATE TABLE lab
( test_start INTERVAL DAY(2) TO SECOND);

INSERT INTO lab VALUES ('90 00:00:00');
INSERT INTO lab VALUES (
    INTERVAL '6 03:30:16' DAY    TO SECOND);

SELECT * FROM lab;

TEST_TIME
-----
+90 00:00:00.000000
+06 03:30:16.000000
```

ORACLE

INTERVAL DAY TO SECOND Data Type: Example

In the above example you are creating the lab table with a test_time column of data type INTERVAL DAY TO SECOND. You then insert into it the value '90 00:00:00' to indicate 90 days and 0 hours minutes and seconds and INTERVAL '6 03:30:16' DAY TO SECOND. The select statements shows how this data is stored in the database.

Daylight Savings Time Boundaries

- **First Sunday in April**
 - Time jumps from 01:59:59 a.m. to 03:00:00 a.m.
 - Values from 02:00:00 a.m. to 02:59:59 a.m. are not valid
- **Last Sunday in October**
 - Time jumps from 02:00:00 a.m. to 01:00:01 a.m.
 - Values from 01:00:01 a.m. to 02:00:00 a.m. are ambiguous because they are visited twice

ORACLE

6-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Support for Daylight Savings Time (DST)

The Oracle database automatically determines, for any given time zone region, whether daylight savings is in effect and returns local time values based accordingly. The datetime value is sufficient for Oracle database to determine whether daylight savings time is in effect for a given region in all cases except boundary cases. A boundary case occurs during the period when daylight savings goes into or comes out of effect. For example, in the US-Eastern region, when daylight savings goes into effect, the time changes from 01:59:59 a.m. to 3:00:00 a.m. The one hour interval between 02:00:00 and 02:59:59 a.m. does not exist. When daylight savings goes out of effect, the time changes from 02:00:00 a.m. back to 01:00:01 a.m., and the one-hour interval between 01:00:01 and 02:00:00 a.m. is repeated.

ERROR_ON_OVERLAP_TIME

The `ERROR_ON_OVERLAP_TIME` is a session parameter to notify the system to issue an error when it encounters a datetime that falls in the overlapped period and no timezone abbreviation was specified to distinguish the period.

For example, last year daylight savings time ended on Oct. 29, 2000 at 02:00:01 a.m. The overlapped periods were:

- 10/29/2000 01:00:01 a.m. to 10/29/2000 02:00:00 a.m. (EDT)
- 10/29/2000 01:00:01 a.m. to 10/29/2000 02:00:00 a.m (EST)

If you input a datetime string which falls in one of these two periods, you need to specify the timezone abbreviation (for example, EDT or EST) in the input string for the system to determine the period. Without this timezone abbreviation, the system will do the following:

If the parameter `ERROR_ON_OVERLAP_TIME` is FALSE then it assumes the input time to standard time (for example, EST). Otherwise, issue an error.

Datetime Functions

To get current session and database time for all datetime data types:

CURRENT_DATE	Session Date & Time, DATE
CURRENT_TIMESTAMP	same, TIMESTAMP WITH TIME ZONE
LOCALTIMESTAMP	Session Date & Time, TIMESTAMP
SYSTIMESTAMP	Server Date & Time, TIMESTAMP
DBTIMEZONE	Server Time Zone, VARCHAR2
SESSIONTIMEZONE	Session Time Zone, VARCHAR2
SYSDATE	Server Date & Time, DATE

ORACLE®

6-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Get Information Functions

CURRENT_DATE returns the current date and time in the session time zone, in a value in the Gregorian calendar of data type **DATE**, which means without fractional seconds.

LOCALTIMESTAMP returns the same value, but in **TIMESTAMP** data type, that is, with fractional seconds.

CURRENT_TIMESTAMP returns the same value as **LOCALTIMESTAMP** but in the **TIMESTAMP WITH TIMEZONE** data type, and thus includes the session time zone.

SYSTIMESTAMP returns date and time of the server in **TIMESTAMP** format.

By contrast, the **SYSDATE** function is the server system date and time, without time zone adjustment.

DBTIMEZONE returns the value of the database time zone. The return type is a time zone offset or a time zone region name, depending on how you defined the database time zone.

SESSIONTIMEZONE returns the value of the current session's time zone. The return type is a time zone offset or a time zone region name, depending on how you specified the session time zone value in the most recent **ALTER SESSION** statement.

Datetime Conversion Functions

TO_DSINTERVAL	Character to INTERVAL DAY TO SECOND
TO_YMINTERVAL	Character to INTERVAL YEAR TO MONTH
TO_TIMESTAMP	Character to TIMESTAMP
TO_TIMESTAMP_TZ	Character to TIMESTAMP WITH TIME ZONE
FROM_TZ	TIMESTAMP to TIMESTAMP WITH TIME ZONE
TO_CHAR	All date and time types to character, extended for new formats

ORACLE

6-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Conversion Functions

The **TO_CHAR** function is extended to support the new datetime and interval types and converts these into a character string with the default or specified format mask.

TO_DSINTERVAL converts a character string to an **INTERVAL DAY TO SECOND** data type.

TO_YMINTERVAL converts a character string to an **INTERVAL YEAR TO MONTH** data type.

TO_TIMESTAMP converts a character string to a **TIMESTAMP** data type.

TO_TIMESTAMP_TZ converts a character string to a **TIMESTAMP WITH TIMEZONE** data type.

The **FROM_TZ** function converts a **TIMESTAMP** value to a **TIMESTAMP WITH TIMEZONE** value with a specified timezone.

TZ_OFFSET returns the time zone offset of the value entered. Adjusts return value on the date the statement is executed (that is, daylight saving time).

Datetime EXTRACT Function

EXTRACT

Gets appropriate field from datetime or interval data type, NUMBER

ORACLE

6-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Extract Function

EXTRACT returns the value as a NUMBER of a specified datetime field from a datetime or interval value expression.

CURRENT_DATE, CURRENT_TIMESTAMP and LOCALTIMESTAMP

- **CURRENT_DATE**
 - Returns the current date from the system
 - Has a data type of **DATE**
- **CURRENT_TIMESTAMP**
 - Returns the current timestamp from the system
 - Has a data type of **TIMESTAMP WITH TIME ZONE**
- **LOCALTIMESTAMP**
 - Returns the current timestamp from user session
 - Has a data type of **TIMESTAMP**

ORACLE

6-25

Copyright © Oracle Corporation, 2001. All rights reserved.

CURRENT_DATE, CURRENT_TIMESTAMP and LOCALTIMESTAMP

Oracle9i supports the datetime value functions **CURRENT_DATE**, and **CURRENT_TIMESTAMP**, which return the current date, and current timestamp respectively. The data type of **CURRENT_DATE** is **DATE**. The data type of **CURRENT_TIMESTAMP** is **TIMESTAMP WITH TIME ZONE**. The time and timestamp values are returned with time zone displacement equal to the current time zone displacement of the SQL session. If a SQL statement causes the evaluation of one or more datetime value functions, then all such evaluations are effectively performed simultaneously. These functions are NLS sensitive, that is, the results will be in the current NLS calendar and datetime formats.

CURRENT_TIMESTAMP has an optional argument to specify the precision of the time and timestamp value returned.

The **LOCALTIMESTAMP** function returns the current date and time in the session time zone in a value of datatype **TIMESTAMP**. The difference between this function and **CURRENT_TIMESTAMP** is that **LOCALTIMESTAMP** returns a **TIMESTAMP** value while **CURRENT_TIMESTAMP** returns a **TIMESTAMP WITH TIME ZONE** value.

CURRENT_DATE: Example

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY  
HH24:MI:SS';
```

```
ALTER SESSION SET TIME_ZONE = '-5:0';
```

```
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

```
SESSION CURRENT_DATE
```

```
-----
```

```
-05:00 04-APR-2000 13:14:03
```

```
ALTER SESSION SET TIME_ZONE = '-8:0';
```

```
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

```
SESSION CURRENT_DATE
```

```
-----
```

```
-08:00 04-APR-2000 10:14:33
```

ORACLE

CURRENT_DATE: Example

CURRENT_DATE returns the current date in the session time zone, in a value in the Gregorian calendar of data type DATE. CURRENT_DATE is very similar to the sysdate function available in earlier releases. However changes to time zones are reflected in the CURRENT_DATE. In the above examples the value of current_date is retrieved from dual for two different timezones by issuing the ALTER SESSION command. The outputs show the change reflected in the time value.

SYSTIMESTAMP

SYSTIMESTAMP returns date and time of the server in TIMESTAMP format. By contrast, the SYSDATE function is the server system date and time, without time zone adjustment. For example:

```
SELECT SYSTIMESTAMP  
FROM DUAL;
```

```
SYSTIMESTAMP
```

```
-----  
20-SEP-00 05.51.54.056056 -07:00
```

CURRENT_TIMESTAMP

CURRENT_TIMESTAMP returns the current date and time in the session time zone, in a value of data type TIMESTAMP WITH TIME ZONE. The time zone displacement reflects the current local time of the SQL session. The difference between this function and LOCAL_TIMESTAMP is that CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME

CURRENT_TIMESTAMP (continued)

ZONE value while LOCALTIMESTAMP returns a TIMESTAMP value. In the following examples you use the ALTER SESSION command to set the time zone value. You then retrieve the CURRENT_TIMESTAMP at two different time zones. The output shows that the change in time zones is reflected in the returned values.

```
ALTER SESSION SET TIME_ZONE = '-07:00';
```

```
SELECT SESSIONTIMEZONE,CURRENT_TIMESTAMP FROM DUAL;
```

```
SESSION CURRENT_TIMESTAMP
```

```
-----  
-07:00 19-SEP-2000 15:38:10.331911 -07:00
```

```
ALTER SESSION SET TIME_ZONE = '-04:00';
```

```
SELECT SESSIONTIMEZONE,CURRENT_TIMESTAMP FROM DUAL;
```

```
SESSION CURRENT_TIMESTAMP
```

```
-----  
-04:00 19-SEP-2000 18:38:10.331911 -04:00
```

LOCALTIMESTAMP

The LOCALTIMESTAMP function returns the current date and time in the session time zone in a value of TIMESTAMP data type. The difference between this function and CURRENT_TIMESTAMP is that LOCALTIMESTAMP returns a TIMESTAMP value while CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value.

In the following examples, you use the ALTER SESSION command to set the time zone value. You then retrieve the CURRENT_TIMESTAMP and LOCAL_TIMESTAMP at two different time zones. The output shows that the change in time zones is reflected in the returned values.

```
ALTER SESSION SET TIME_ZONE = '-8:00';
```

```
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;
```

```
CURRENT_TIMESTAMP
```

```
LOCALTIMESTAMP
```

```
-----  
19-SEP-00 11:54:25.655551 -08:00 19-SEP-00 11:54:25.655551
```

```
ALTER SESSION SET TIME_ZONE = '-5:00';
```

```
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;
```

```
CURRENT_TIMESTAMP
```

```
LOCALTIMESTAMP
```

```
-----  
19-SEP-00 14:54:25.676207 -05:00 19-SEP-00 14:54:25.676207
```

DBTIMEZONE: Example

**DBTIMEZONE: Returns the value
of the database time zone**

```
SELECT DBTIMEZONE FROM DUAL;  
  
DBTIMEZ  
-----  
+00:00
```

ORACLE

6-28

Copyright © Oracle Corporation, 2001. All rights reserved.

DBTIMEZONE

The DBTIMEZONE function returns the value of the database time zone. The return type is a time zone offset or a time zone region name, depending on how the user specified the database time zone value in the most recent CREATE DATABASE or ALTER DATABASE statement.

SESSIONTIMEZONE

The SESSIONTIMEZONE function returns the value of the current session's time zone. The return type is a time zone offset or a time zone region name, depending on how the user specified the session time zone value in the most recent ALTER SESSION statement. For example:

```
SELECT SESSIONTIMEZONE FROM DUAL;  
  
SESSION  
-----  
-05:00
```

Using FROM_TZ: Example

FROM_TZ: Converts a timestamp value at a time zone
to a **TIMESTAMP WITH TIME ZONE** value

```
SELECT FROM_TZ (
  TIMESTAMP '2000-03-28 08:00:00', '3:00')time
FROM   DUAL;

TIME
-----
28-MAR-00 08.00.00 AM +03:00
```

ORACLE

FROM_TZ

The **FROM_TZ** function converts a timestamp value at a time zone to a **TIMESTAMP WITH TIME ZONE** value. In the above example the **FROM_TZ** function accepts a literal and converts it to a **TIMESTAMP** value. The first argument is the **TIMESTAMP** value and the second argument is the time zone offset.

Using TO_DSINTERVAL: Example

**TO_DSINTERVAL: Converts a character string to an
INTERVAL DAY TO SECOND data type**

```
SELECT last_name,  
       TO_CHAR(hire_date, 'mm-dd-yy:hh:mi:ss') hire_date,  
       TO_CHAR(hire_date +  
               TO_DSINTERVAL('100 10:00:00'),  
               'mm-dd-yy:hh:mi:ss') hiredate2  
FROM employees;
```

LAST_NAME	HIRE_DATE	HIREDATE2
-----	-----	-----
King	06-17-87:12:00:00	09-25-87:10:00:00
Kochhar	09-21-89:12:00:00	12-30-89:10:00:00
De Haan	01-13-93:12:00:00	04-23-93:10:00:00
...		

ORACLE

6-30

Copyright © Oracle Corporation, 2001. All rights reserved.

TO_DSINTERVAL

TO_DSINTERVAL converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL DAY TO SECOND type.

In the above example, the date 100 days and 10 hours after the hire date is obtained.

TO_YMINTERVAL

The TO_YMINTERVAL function converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL YEAR TO MONTH type.

In the following example, the date one year and two months after the hire date is obtained.

```
SELECT hire_date, hire_date + TO_YMINTERVAL('01-02') ytm  
FROM employees;
```

HIRE_DATE	YTM
-----	-----
17-JUN-87	17-AUG-88
21-SEP-89	21-NOV-90
13-JAN-93	13-MAR-94
03-JAN-90	03-MAR-91
21-MAY-91	21-JUL-92
...	

TO_TIMESTAMP: Example

TO_TIMESTAMP: Converts a value of CHAR or VARCHAR2 data type to a value of TIMESTAMP data type

```
SELECT TO_TIMESTAMP('1999-12-01 11:00:00',
    'YYYY-MM-DD HH:MI:SS') Timestamp
FROM DUAL;

TIMESTAMP
-----
01-DEC-99 11.00.00.000000000 AM
```

ORACLE

6-31

Copyright © Oracle Corporation, 2001. All rights reserved.

TO_TIMESTAMP

TO_TIMESTAMP converts a value of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of TIMESTAMP data type. It is very similar to the TO_DATE function in that it accepts a literal as the first argument and the input format as the second argument. It returns the value converted to a TIMESTAMP data type.

TO_TIMESTAMP_TZ

TO_TIMESTAMP_TZ converts a value of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of TIMESTAMP WITH TIME ZONE data type. It is very similar to the TO_DATE function in that it accepts a literal as the first argument and the input format as the second argument. It returns the value converted to a TIMESTAMP data type. For example:

```
SELECT TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00',
    'YYYY-MM-DD HH:MI:SS TZH:TZM') TIMESTAMPTZ
FROM DUAL;

TIMESTAMPTZ
-----
01-DEC-99 11.00.00.000000000 AM -08:00
```

TZ_OFFSET: Example

TZ_OFFSET: Returns the time zone offset

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;  
TZ_OFFS  
-----  
-04:00
```

ORACLE

TZ_OFFSET

TZ_OFFSET returns the time zone offset corresponding to the value entered based on the date the statement is executed. In the above example the offset for US/Eastern from the UTC is returned.

Using EXTRACT: Example

EXTRACT

- Extracts values from datetime and interval
- Extracting from datetime with time zone returns value in UTC

```
SELECT EXTRACT(YEAR FROM DATE '1998-09-20') year
FROM   DUAL;

YEAR
-----
1998
```

ORACLE

6-33

Copyright © Oracle Corporation, 2001. All rights reserved.

EXTRACT Datetime Expression

An EXTRACT expression extracts and returns the value of a specified datetime field from a datetime or interval value expression. When you extract a TIMEZONE_REGION or TIMEZONE_ABBR (abbreviation), the value returned is a string containing the appropriate time zone name or abbreviation. When you extract any of the other values, the value returned is in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC.

You can extract YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, and TIMEZONE_ABBR (abbreviation). The value must contain the field, for example, you cannot extract SECOND from a INTERVAL YEAR TO MONTH.

When you extract a TIMEZONE_REGION or TIMEZONE_ABBR, the value returned is a string containing the appropriate time zone name or abbreviation. When you extract any of the other values, the value returned is in the Gregorian calendar format. When extracting from a datetime with a time zone value, the value returned is in UTC.

```
SELECT EXTRACT(YEAR FROM DATE '1998-03-07') FROM DUAL;
```

Using TO_CHAR with format mask to achieve the same result returns the value in VARCHAR2

```
SELECT TO_CHAR(DATE '1998-03-07', 'YYYY') FROM DUAL;
```

Summary

In this lesson, you should have learned how to:

- **Describe the datetime data types**
- **Use the new data types for datetime and interval**
- **Implement time zone with offsets and named regions**
- **Discuss daylight savings time (DST) with the datetime data types**
- **Use the new built-in SQL functions for datetime and interval**

ORACLE

6-34

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson you should have learned about the different datetime functions and data types and their usage. You have also learnt about the enhancement to Unicode in Oracle9i.

Practice 6 Overview

This practice covers the following topics:

- **Using the datetime data types**
- **Retrieving intervals**
- **Manipulating dates using the datetime functions**

ORACLE

6-35

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing This Practice

To perform this practice go to Appendix A, “Practices.”



Migrating LONGs to LOBs

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Convert a table with a LONG column to a large object (LOB) data type**
- **Use LOB enhancements in SQL and PL/SQL**

ORACLE

7-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

The CLOB and BLOB internal LOB data types introduced in Oracle8 provide many advantages over the older LONG and LONG RAW data types. The Oracle9i database includes several features that facilitate the migration of data and applications to the LOB data types.

For more information, see *Oracle9i Application Developer's Guide - Large Objects (LOBs)*. Specialized subsets of this manual are available in Portable Document Format for the following language interfaces:

- PL/SQL
- C (OCI)
- Pro*COBOL
- Visual Basic (Oracle Objects for OLE)
- Java

Overview

- **Large object (LOB) data types, were introduced in Oracle8.**
- **They provide better functionality than the LONG data type.**
- **It is recommended that existing applications using the LONG data type be converted to use LOBs because of the added functionality.**
- **To assist in the migration process, the LONG API for LOBs has been introduced in Oracle9i.**

ORACLE

7-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview

The following are some of the advantages of LOB data types over LONG data types:

- Up to 4 Gigabytes in size for LOBs versus 2 Gigabytes for LONG columns
- Random piecewise access for LOBs versus sequential access for LONG columns
- Multiple LOB columns versus a single LONG column per table
- Attributes of user defined object types can be LOBs; they cannot be LONG
- Storage flexibility with separate LOB segments versus the entire LONG being stored in-line, in the table data segment itself

The LONG API is an umbrella term for a group of features, some introduced in versions prior to Oracle9i, that facilitate the manipulation of large character strings and raw binary data. This API ensures that when you change your LONG columns to LOBs, your existing applications will require few changes, if any. When possible, change your existing applications to use LOBs instead of the LONG data type because of the added benefits that LOBs provide.

Applications of LOB enhancements:

- They facilitate migration from LONG data types to LOB data types
- They encourage customers to use LOBs in new applications
- They use LOBs in place of the LONG data type in existing applications.

Oracle9i LONG to LOB Migration

- The **ALTER TABLE** statement has been enhanced to allow the data type of a **LONG** column to be modified to **CLOB**, and that of a **LONG RAW** column to be modified to **BLOB**.
- The conversion can be completed using the **ALTER TABLE . . . MODIFY** statement.
- During the conversion the space required for both the **LONG** and the **LOB** data must be available.

ORACLE

7-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i LONG to LOB Migration

The **ALTER TABLE** command can be used to change a **LONG** to a **LOB**. Default values can be specified for the **LOB** column and a **LOB** storage clause can be specified for storing the **LOB** segment. Other **ALTER TABLE** clauses are not allowed within the same statement. Any triggers and constraints on the old **LONG** column are maintained for the new **LOB** column. The default value is also maintained, unless you specify a new default in the **ALTER TABLE** statement.

The new **ALTER TABLE . . . MODIFY** command only modifies a **LONG** or a **LONG RAW** column to a **LOB** column. It does not modify a **VARCHAR2** or a **RAW** column.

LONG to LOB Migration: Example

```
ALTER TABLE print_media
  MODIFY (press_release CLOB);

Table altered.

DESCRIBE print_media
Name                               ... Type
-----
...
PRESS_RELEASE                      CLOB
```

ORACLE

7-5

Copyright © Oracle Corporation, 2001. All rights reserved.

LONG to LOB Migration: Example

Logging During Migration

During the migration, the redo changes for the table are logged only if the table has logging on. The redo changes for the column being converted from LONG to LOB are logged only if the storage characteristics of the LOB indicate logging.

Partial Syntax

```
ALTER TABLE [<schema>.<table_name>
MODIFY ( <long_column_name> { CLOB | BLOB | NCLOB }
[DEFAULT <default_value> ] [LOB_storage_clause];
```

LONG to LOB Migration: Example (continued)

Example

```
SQL> describe print_media
```

Name	Null?	Type
PRODUCT_ID		NUMBER(6)
AD_ID		NUMBER(6)
AD_COMPOSITE		BLOB
AD_SOURCETEXT		CLOB
AD_FINALTEXT		CLOB
AD_FLTEXTN		NCLOB
AD_TEXTDOCS_NTAB		TEXTDOC_TAB
AD_PHOTO		BLOB
AD_GRAPHIC		BINARY FILE LOB
AD_HEADER		ADHEADER_TYP
PRESS_RELEASE		LONG

```
SQL> ALTER TABLE print_media  
2 MODIFY (press_release CLOB);
```

Table altered.

```
SQL> describe print_media
```

Name	Null?	Type
PRODUCT_ID		NUMBER(6)
AD_ID		NUMBER(6)
AD_COMPOSITE		BLOB
AD_SOURCETEXT		CLOB
AD_FINALTEXT		CLOB
AD_FLTEXTN		NCLOB
AD_TEXTDOCS_NTAB		TEXTDOC_TAB
AD_PHOTO		BLOB
AD_GRAPHIC		BINARY FILE LOB
AD_HEADER		ADHEADER_TYP
PRESS_RELEASE		CLOB

Restrictions on LOB Migration

- **LOBs are not allowed in clustered tables.**
- **The migration of LONG columns for a replicated table cannot itself be replicated.**
- **Materialized views on the table being migrated will have to be rebuilt manually.**
- **Triggers do not support LOBs in the UPDATE OF clause.**
- **If a view has an INSTEAD OF trigger then you cannot specify strings for insert or update of LOB columns.**

ORACLE

7-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Restrictions on LOB Migration

LOB columns are not allowed in clustered tables, but a LONG or a LONG RAW column is allowed. So if a table is a part of a cluster, its LONG or LONG RAW column cannot be changed to CLOB or BLOB.

If a table is replicated or has materialized views, and its LONG column is changed to LOB, the replica tables need to be rebuilt manually.

LOB columns are not allowed in the UPDATE OF list in of an update trigger. So a trigger that was intended to fire on the update of a LONG column becomes invalid (and does not recompile) after the column is migrated to a LOB.

The implicit conversion from character data to LOB is not allowed in INSTEAD OF triggers. If a view with a LOB column has an INSTEAD OF trigger, then INSERT and UPDATE statements on the view that supply character values for the LOB will fail.

Prior to changing a LONG column to a LOB, you should investigate any triggers on the table.

Restrictions on LOB Migration

- All indexes on all columns in the table being migrated need to be rebuilt manually.
- Domain indexes on the LONG columns have to be dropped before migration.

ORACLE

7-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Restrictions on LOB Migration (continued)

All indexes on all columns of the table being altered (including function-based and domain indexes) need to be rebuilt manually after changing a LONG column to LOB.

Domain indexes on the LONG or LONG RAW column have to be dropped before altering a LONG column to LOB.

SQL Support for LOB Migration

- **SQL functions and operators that take VARCHAR2 parameters accept CLOB parameters as well.**
- **SQL functions that take RAW parameters accept BLOB parameters as well.**

ORACLE

Using SQL Functions on LOBs: Example

```
SELECT SUBSTR(press_release, 1, 200)
FROM print_media;
```

ORACLE

7-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Using SQL Functions on LOBs: Example

In this example, the PRESS_RELEASE column of the print_media table has been converted to the CLOB data type.

Example

```
SQL> SELECT SUBSTR(press_release, 1, 20)
      2  FROM print_media;
SELECT SUBSTR(press_release, 1, 20)
      *
```

ERROR at line 1:

ORA-00932: inconsistent datatypes

```
SQL> ALTER TABLE print_media
      2  MODIFY (press_release CLOB);
```

Table altered.

```
SQL> SELECT SUBSTR(press_release, 1, 20)
      2  FROM print_media;
```

PL/SQL Support for LOB Migration

- All predefined functions defined in the **STANDARD** package, such as **SUBSTR**, **INSTR**, and so on, that accept **LONG** parameters now accept **CLOB** parameters as well.
- All predefined functions accepting **LONG RAW** parameters now accept **BLOB** parameters.
- Implicit conversion from **LOB** data types to **VARCHAR2** and **RAW** data types and vice versa is done in assignments and when passing parameters.
- A user can define and bind **LOB** columns as **VARCHAR** and **RAW** external data types.

ORACLE

7-11

Copyright © Oracle Corporation, 2001. All rights reserved.

PL/SQL Support for LOB Migration

All functions defined in the **STANDARD** PL/SQL package that previously took **LONG** or **LONG RAW** parameters have been overloaded so that they also accept **CLOB** or **BLOB** parameters.

Overloaded Functions in **STANDARD** Package

LENGTH and **LENGTHB**

INSTR and **INSTRB**

SUBSTR and **SUBSTRB**

CONCAT and **||**

LPAD and **RPAD**

LTRIM, **RTRIM** and **TRIM**

LIKE

REPLACE

LOWER and **UPPER**

NLS_LOWER and **NLS_UPPER**

NVL

Also overloaded are the comparison operators **>**, **=**, **<**, and **!=**.

Implicit Assignment and Parameter Passing to LOBS

- Variables declared using %TYPE based on LONG columns are implicitly converted to accept LOB values after the LONG columns are modified to LOB data types.
- LOB, VARCHAR2, and RAW values can be passed to parameters defined using %TYPE based on LONG columns after the columns are modified to LOB data types.
- Users can fetch a CLOB column into a VARCHAR2 variable, or a BLOB column into a RAW variable.
- This allows built-ins to accept CLOB and BLOB data in addition to VARCHAR2 and RAW data.

ORACLE

7-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Implicit Assignment and Parameter Passing to LOBS

Example

```
CREATE TABLE lobtab (lcol LONG);

/* create an anonymous block to demonstrate */

DECLARE
  var1 VARCHAR2(100);
  var2 lobtab.lcol%type;
BEGIN
  SELECT * INTO var2 FROM lobtab;
  var1 := var2; -- This will change from VARCHAR2 <- LONG to
                --                               VARCHAR2 <- CLOB
  var2 := var1; -- This will change from LONG <- VARCHAR2 to
                --                               CLOB <- VARCHAR2
END;

/*change LONG to LOB */
ALTER TABLE lobtab MODIFY lcol CLOB;
```

If the anonymous block is executed again, it still compiles because of the implicit conversions of the variables.

Implicit Conversion of LOBs: Example

```
DECLARE
  v_clob  CLOB;
  v_char  VARCHAR2(32500);
BEGIN
  SELECT ad_sourcetext INTO v_clob
  FROM print_media
  WHERE product_id = 2056;
  v_char := rtrim(v_clob);
END;
```

ORACLE

Support for LOB Migration in OCI

You can:

- Define a CLOB column as VARCHAR2 or LONG to select up to 4 gigabyte (GB) of LOB data directly into a buffer
- Define a BLOB column as RAW or LONG RAW to select up to 4 gigabyte (GB) of LOB data directly into a buffer
- Select CLOB and BLOB columns in a single piece, piecewise, or in array mode

ORACLE

7-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Support for LOB Migration in OCI

In Oracle Call Interface (OCI) define operations on CLOB columns, you can specify the VARCHAR2 or LONG external types to select or fetch up to 4 GB of LOB data directly into a buffer. Likewise, in define operations on BLOB columns, you can specify the RAW or LONG RAW external types to select or fetch up to 4 GB of LOB data directly into a buffer. You can select or fetch data from CLOB and BLOB columns in one piece, piecewise in multiple pieces, or in array mode.

Benefits

Many OCI applications written to access LONG columns will not have to be rewritten if the LONG columns are migrated to LOBs.

Other LOB Enhancements

- **LOB columns are allowed in partitioned index-organized tables.**
- **Function-based indexes are allowed on LOB columns.**

ORACLE

7-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Other LOB Enhancements

It is now possible to create partitioned index-organized tables (IOTs) with LOB columns. Function-based indexes can be created on LOB columns.

Summary

In this lesson, you should have learned how to:

- **Migrate LONGs to LOBs**
- **Describe SQL support for migration**
- **Describe PL/SQL support for migration**
- **Use the LONG API**

ORACLE

7-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

New features make it simple to migrate data and applications so that they can take advantage of the CLOB and BLOB data types.

Practice 7 Overview

This practice covers migrating a LONG column to the CLOB data type.

ORACLE

7-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 7 Overview

In this practice you will alter a table that contains a LONG column, converting the column to the CLOB data type.

To perform this practice go to Appendix A, “Practices.”

8

Object and Collection Type Enhancements

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Implement object type inheritance**
- **Evolve the definition of an object type**
- **Create and use multilevel collection types**

ORACLE

8-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

The Oracle8 and Oracle8i models did not directly support inheritance. In Oracle9i, object type definitions can be organized into inheritance hierarchies.

In prior releases an object type referenced by a table or user-defined type could only be changed by adding methods to it. In Oracle9i objects can evolve more freely and any changes made to a type can propagate to schema objects that reference it. Oracle9i provides the ability to add, drop, and modify attributes of an object type.

Collections were introduced in Oracle8. They provide an alternate way of modeling data in one-to-many relationships. There are two kinds of collection types: varrays and nested tables. User-defined collection types can be used to specify columns in database tables, attributes of object types, and PL/SQL variables. Oracle9i allows collections to be nested within other collections.

For more information, see *Oracle9i Application Developer's Guide - Object-Relational Features*.

Overview

- **Inheritance**
- **Type evolution**
- **Multilevel collections**

ORACLE

8-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview

Type inheritance provides a way to organize types analogous to the way types can be used for organizing objects. In Oracle9i the functionality of types has been extended to support inheritance, thus completing the object functionality of the Oracle database. You will learn the terminology, syntax extensions, SQL operators, and object functionality in Oracle9i.

Type evolution allows you to evolve the definition of types over time to respond to changing application requirements.

Multilevel collections let you model multiple one-to-many relationships more naturally.

Type Inheritance

- **Allows types to share similar characteristics as well as extend their characteristics**
- **Consists of supertypes and subtypes**
- **Subtypes inherit the characteristics of supertypes.**
- **Subtypes can also have their own attributes and methods.**
- **Subtypes can be substituted for supertypes in code.**
- **Based on the SQL: 1999 inheritance model**
- **A single inheritance model**

ORACLE

8-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Type Inheritance

Type inheritance allows sharing similarities between types as well as extending their characteristics. Object applications typically are organized into types and type hierarchies consisting of supertypes and subtypes. A supertype is an object and a subtype is another object that has inherited from (extended) one supertype. A subtype inherits all its supertype's attributes and methods. A subtype can also add new attributes and methods of its own. All attributes and methods of a type are accessible by other methods in the same type and its subtypes, or to methods that have access to that type.

A subtype can override any method in its supertype chain. The subtype value appears to the surrounding code just as the value of the supertype would, even if it uses separate mechanisms within its specialization methods. Instance substitutability refers to the ability to use an object value of a subtype in a context declared in terms of a supertype. REF substitutability refers to the ability to use a REF to a subtype in a context declared in terms of a REF to a supertype.

Applications

Inheritance is a crucial feature that will promote a wider adoption of object technology. Type inheritance enables the mapping of object models completely within the database. It allows for the extensibility of object attributes to other objects. Type inheritance provides the ability for users to extend their existing data models and define new attributes. It provides better support for C++ and Java applications.

Type Inheritance (continued)

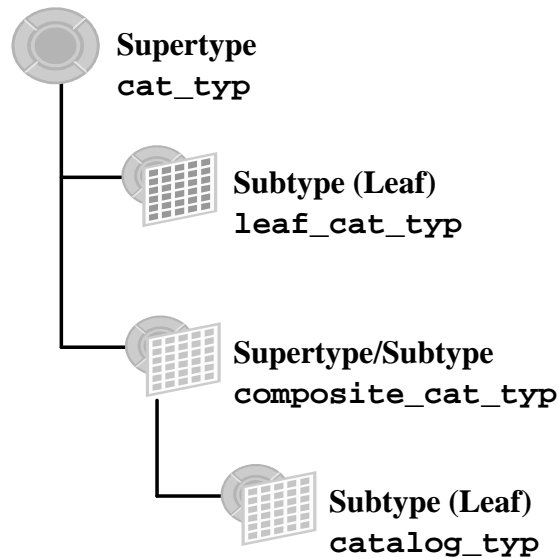
Benefits

Sharing: Provides efficiency in development time and improves manageability of applications.

Extensibility: Provides flexibility and power of expression to solving application problems.

Substitutability: Allows a value of some subtype to be used by code originally written for the supertype, without requiring any advance knowledge of the subtype.

Type Hierarchy



ORACLE

8-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Type Hierarchy

An object type can be created as a subtype of an existing object type. The Oracle9i database implements on a single inheritance model, which means that the subtype can be derived from only one parent type. A type inherits all the attributes and methods of its direct supertype. It can add new attributes and methods and further override any of the inherited methods.

The above figure illustrates two subtypes, `LEAF_CAT_TYP` and `COMPOSITE_CAT_TYP`, created under the `CAT_TYP` object type. A subtype can itself be refined by defining another subtype under it, thus forming type hierarchies. The `catalog_typ` is a subtype of the type hierarchies implement kind-of relationships. That is, a subtype is a kind of its supertype. In the slide example, leaf categories and composite categories are kinds of categories. And a catalog is a kind of composite cat.

A type that has no subtypes is sometimes referred to as a leaf type.

Type Declarations

Type declarations can be:

- **FINAL:** Cannot have subtypes
- **NOT FINAL:** Can have subtypes
- **INSTANTIABLE:** Has constructor
- **NOT INSTANTIABLE:** Has no constructor

ORACLE

FINAL and NOT FINAL Types

- A type declaration must use the NOT FINAL clause to allow subtypes.
- A NOT FINAL type can nest to multiple levels.
- A type declaration is FINAL by default for backward compatibility.

```
CREATE OR REPLACE TYPE cat_typ AS OBJECT
  (cat_name    varchar2(50),
   cat_description varchar2(1000) ,
   cat_id      number(2)
  ) NOT FINAL;
```

ORACLE

FINAL and NOT FINAL Types

In the above example, cat_typ is defined as a NOT FINAL type. This means that we can define subtypes of cat_TYP. FINAL types can be altered and defined as NOT FINAL. A NOT FINAL type (as long as it has no subtypes) can be altered to a FINAL type.

NOT INSTANTIABLE Types

- There is no constructor for a NOT INSTANTIABLE type.
- It is not possible to create instances of this type.
- The typical usage would be to define instantiable subtypes, which are used to populate the type.

```
CREATE OR REPLACE TYPE cat_typ AS OBJECT
( cat_name          varchar2(50) ,
  cat_description   varchar2(1000),
  cat_id            number(2)
) NOT INSTANTIABLE NOT FINAL;
```

ORACLE

8-9

Copyright © Oracle Corporation, 2001. All rights reserved.

NOT INSTANTIABLE Types

In the above example, `cat_typ` has been created as NOT INSTANTIABLE. A method of a type can also be defined to be NOT INSTANTIABLE. Declaring a method as NOT INSTANTIABLE means that the type is not providing an implementation for that method. Further, a type that contains any noninstantiable methods must necessarily be declared as NOT INSTANTIABLE.

Example

```
CREATE OR REPLACE TYPE cat_typ AS OBJECT
( cat_name          VARCHAR2(50)
, cat_description   VARCHAR2(1000)
, cat_id            NUMBER(2)
, NOT INSTANTIABLE MEMBER FUNCTION f1 RETURN NUMBER )
NOT INSTANTIABLE NOT FINAL;
```

A subtype of a noninstantiable type can override any of the noninstantiable methods of the supertype and provide concrete implementations. If there are any noninstantiable methods remaining, the subtype must also necessarily be declared as NOT INSTANTIABLE. A noninstantiable subtype can be defined under an instantiable supertype. Declaring a noninstantiable type to be FINAL is not useful and is not allowed.

Substitutability

- **Substitutability refers to the ability of a supertype slot to hold subtype instances**
 - Instance substitutability
 - REF substitutability
- **Examples**
 - Object and REF columns
 - Collection elements
 - Method parameters
 - PL/SQL variables
 - Bind variables

ORACLE

8-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Substitutability

When you work with types in a type hierarchy, sometimes you want to work at the most general level and, for example, select or update all categories. But other times you want to select or update only leaf categories, or only composite categories.

The ability to select all categories and get back not only objects whose declared type is `cat_typ`, but also objects whose declared type is `leaf_cat_typ`, or `composite_cat_typ`, or `catalog_typ` is called substitutability. A supertype is substitutable if one of its subtypes can substitute or stand in for it in a slot (a variable, column, and so on) whose declared type is the supertype. This is otherwise known as polymorphism.

In general, types are substitutable. This is what you would expect, given that a subtype is just a specialized kind of any of its supertypes. Formally, though, a subtype is a type in its own right, and is not the same type as its supertype. A column that holds all categories, including leaf categories, composite categories, and catalogs, actually holds data of multiple types. Substitutability applies in attributes, columns, and rows (namely, of an object view or object table) declared to be an object type, a REF to an object type, or a collection type.

In principle, object attributes, collection elements and REFs are always substitutable, that is there is no syntax at the level of the type definition to constrain their substitutability to some subtype. You can, however, turn off or constrain substitutability at the storage level, for specific tables and columns by using the `NOT SUBSTITUTABLE AT ALL LEVELS` clause.

Attribute Inheritance

- **Subtypes automatically inherit all attributes declared in their supertypes.**
- **New attributes can be declared in subtypes, but their names must be different from the names of attributes and methods declared in the supertype chain.**

ORACLE

8-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Attribute Inheritance

Subtypes inherit all attributes declared in their supertypes automatically. New attributes can be declared in subtypes, but their names must be different from the names of attributes and methods declared in the supertype chain.

Example

```
CREATE OR REPLACE TYPE leaf_cat_typ UNDER cat_typ
( product_ref_list varchar2(30)
);
```

The `leaf_cat_typ` type is created as a subtype of `cat_typ`. Therefore an instance of `leaf_cat_typ` has all the attributes of `cat_typ` in addition to the new attributes that are declared within `leaf_cat_typ`. Similarly, the statement below creates another subtype, `composite_cat_typ`, under `CAT_TYP`.

```
CREATE TYPE composite_cat_typ UNDER cat_typ
( subcat_ref_list varchar2(30)
) NOT FINAL;
```

Method Inheritance

- **A subtype automatically inherits all methods declared in its supertype.**
- **It can also declare new methods or declare new overloads for the methods inherited from its supertype.**

ORACLE

8-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Method Inheritance

A subtype inherits all methods declared in its supertype automatically. It can also declare new methods or declare new overloads for the methods inherited from its supertype.

Example

```
CREATE TYPE cat_typ AS OBJECT
( cat_name          VARCHAR2(50)
, cat_description   VARCHAR2(1000)
, cat_id            NUMBER(2),
  MEMBER PROCEDURE cat_proc (id NUMBER), ...)
  NOT FINAL;
CREATE TYPE leaf_cat_typ UNDER cat_typ
( product_ref_list  varchar2(30),
  MEMBER PROCEDURE cat_proc (name VARCHAR2),
  MEMBER PROCEDURE cat_proc2 (expire DATE) );
```

For an object of the leaf_cat_typ type, an invocation of the cat_proc procedure will invoke the subtype method, or the supertype method, depending of the data type of the actual parameter.

```
c leaf_cat_typ := leaf_cat_typ(...);
c.cat_proc(1);           -- invokes supertype method
c.cat_proc('X');        -- invokes subtype method
c.cat_proc2(SYSDATE);    -- invokes subtype method
```


Method Override

- A subtype can redefine methods that are defined in the supertype.
- Methods declared as `FINAL` in the supertype cannot be overridden in the subtype.
- All methods of a `FINAL` type are `FINAL`.
- Methods of a `NOT FINAL` type are `NOT FINAL` by default.
- Virtual (dynamic) method dispatch occurs at run time.

ORACLE

8-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Method Override

When the signature of a method is the same as the signature of a method in a supertype, the supertype method does not overload the supertype method. Instead it *overrides* the supertype method and you must use the `OVERRIDING` keyword.

```
CREATE TYPE cat_typ AS OBJECT
( cat_name          VARCHAR2(50)
, cat_description   VARCHAR2(1000)
, cat_id            NUMBER(2),
MEMBER PROCEDURE cat_proc (id NUMBER) )
NOT FINAL;
CREATE TYPE leaf_cat_typ UNDER cat_typ
( product_ref_list  varchar2(30),
OVERRIDING MEMBER PROCEDURE cat_proc (id NUMBER) );
```

Dynamic Method Dispatch

- A method invoked on an object is dispatched to the specific implementation based on the run-time type, also called most specific type (MST), of the object (an example of polymorphism).
- For example:

```
DECLARE
  v_cat_t cat_typ;
BEGIN
  v_cat_t := cat_typ(...); -- MST is cat_typ
  v_cat_t.cat_proc(); -- invokes cat_typ
  v_cat_t := leaf_cat_typ(); -- MST is subtype
  v_cat_t.cat_proc(); -- invokes leaf_cat_typ
END;
```

ORACLE

Dynamic Method Dispatch

In the above example, the method being invoked is based on the run-time value of the VAR_T variable. The run-time type is otherwise known as the most specific type.

Rights Model

- **A subtype has the same rights model as its supertype.**
- **A subtype defined to use definer's rights must be in the same schema as its supertype.**
- **Types hierarchies defined to use invoker's rights can span schemas.**

ORACLE

8-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Rights Model

In a type hierarchy, a subtype has the same rights model as its immediate supertype. That is, it implicitly inherits the rights model of the supertype and cannot explicitly specify one. Furthermore, if the supertype was declared with definer's rights, the subtype must reside in the same schema as the supertype. These rules allow invoker rights type hierarchies to span schemas. However, type hierarchies that use a definer rights model must reside within a single schema.

To create generic object types that can more easily be used in any schema, you should define the type to use invoker's rights. The `AUTHID CURRENT_USER` clause of the `CREATE TYPE` statement is used to specify invoker's rights.

In general, use invoker's rights when both of the following conditions are true:

- There are type methods that access and manipulate data.
- Users who did not define these type methods must use them.

Example

```
CREATE TYPE cat_typ
  AS OBJECT (...);           --definers rights type
CREATE TYPE leaf_cat_typ -- subtype in same schemas
  UNDER cat_typ (...) ;     -- as supertype is OK
CREATE TYPE schema1.composite_cat_typ
  UNDER cat_typ(...);       -- causes error
```

Object and REF Assignment

An object or REF value may be assigned to an object or REF container when:

- The source and the target types are identical
- The source type is a subtype of the target type
 - Called widening
 - Occurs implicitly
- The source type is a supertype of the target type
 - Called narrowing
 - Requires the use of the TREAT operator

ORACLE

Widening: Example

```
CREATE TABLE cat_tab (  
  catmain cat_typ,  
  catleaf leaf_cat_typ  
  );  
  
UPDATE cat_tab  
SET catmain = catleaf; -- assigning sub type to  
                        -- supertype  
  
DECLARE  
var1 cat_typ;  
var2 leaf_cat_typ;  
BEGIN                                -- assigning sub type  
var1 := var2;                        -- to supertype  
END;
```

ORACLE

8-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Widening: Example

In the above slide you can see how a subtype can be assigned instead of a supertype. This is an example of substitutability (polymorphism), and is known as widening.

Narrowing by Using TREAT: Example

```
-- updating supertype with subtype  
UPDATE cat_tab set catleaf =  
TREAT (catmain AS leaf_cat_typ);
```

ORACLE

8-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Narrowing: Example

In the above slide you can see how a supertype can be used instead of a subtype. This is an example of substitutability (polymorphism), and is known as narrowing. In order for narrowing to be possible, the TREAT SQL function must be used.

Inheritance Support in Data Dictionary

The following views have been enhanced to support inheritance, substitutability and type evolution:

- {USER | ALL | DBA} _TYPES
- {USER | ALL | DBA} _TYPE_ATTRIBUTES
- {USER | ALL | DBA} _TYPE_METHODS

ORACLE

Inheritance Support in Data Dictionary

The following are descriptions of these views:

USER_TYPES includes the following columns ATTRIBUTES, METHODS, PREDEFINED, INCOMPLETE, FINAL, INSTANTIABLE, SUPERTYPE_OWNER, SUPERTYPE_NAME, LOCAL_ATTRIBUTES and LOCAL_METHODS which help to track the inheritance hierarchy.

USER_TYPE_METHODS includes FINAL, INSTANTIABLE, OVERRIDING, INHERITED which help track method inheritance.

USER_TYPE_ATTRS includes ATTR_TYPE_OWNER and INHERITED which help track attribute inheritance.

Object Type Evolution

- In prior releases an object type referenced by a table or user-defined type could only be changed by adding methods to it.
- In Oracle9i objects can evolve more freely and any changes made to a type can propagate to schema objects that reference it.
- In Oracle9i attributes and methods of an object type can be:
 - Added
 - Dropped
 - Modified

ORACLE

Type Dependencies

- **A user-defined type can be referenced by:**
 - Tables or views
 - Types or subtypes
 - Program units
 - Index types
 - Function-based indexes
 - Operators
- **Any schema object that references a user-defined type is a dependent object.**

ORACLE

8-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Type Dependencies

Any object that references a type is considered a dependent object. When a type is modified, all dependent program units, views, operators, and index types are marked invalid. Some objects are revalidated automatically the next time that they are accessed. If the recompilation is successful, then the objects become valid again. However, when a type has either type or table dependents, altering the type definition becomes more complicated because of existing persistent data, which relies on the current type definition. To support this functionality, the `ALTER TYPE` statement has additional options to allow changes to an object type to be propagated to its dependent types and tables.

You can track type dependencies in the `USER_DEPENDENCIES` data dictionary view.

Propagating Type Changes

Type changes are divided into two categories:

- **Nonstructural changes:** Changes that do not affect the table data such as adding or dropping a method
- **Structural changes:** Changes that affect the structure of tables and the data such as adding, modifying, or dropping attributes from a type

ORACLE

Propagating Nonstructural Changes

- **Nonstructural changes do not create a new version of the type.**
- **All dependent objects immediately become invalid.**
- **Dependent objects can be validated by recompilation.**
- **Dependent table data is unaffected.**

ORACLE

8-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Propagating Nonstructural Changes

When a nonstructural change such as adding or dropping a method is made to a parent type. The all dependent objects immediately become invalid. each dependent object needs to be recompiled individually. However, since the change to the parent did not affect the attributes of the parent type all dependent types and table objects can be recompiled without causing any change to existing data.

Propagating Structural Changes

Propagating structural changes is a four step process completed in memory a row at a time.

- 1. A new version for each valid dependent type is created.**
- 2. All views, index types , operators, domain indexes and program units that depend on the target type are invalidated.**
- 3. The structure of all dependent tables is upgraded to the latest version of each referenced type.**
- 4. The column data is converted to the format of the latest type version.**

ORACLE

8-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Propagating Structural Changes

A structural type change requires creating a new version of the type and propagating the change to dependent objects. This involves the following steps:

1. A new version for each valid dependent type is created so that each one references the latest version of the altered type. This is done before converting dependent tables to ensure that the latest version of each referenced type is used.
2. All views, index types, operators, domain indexes, and program units that depend directly or indirectly on the target type are invalidated so that they can be recompiled based on the latest version of the type.
3. The structure of all dependent tables are upgraded to the latest version of each referenced type. For each attribute added to a type, one or more internal columns is added to the table depending on the new attribute's type. New attributes are added as null values. For each modified attribute, the type of its associated column is changed accordingly.
4. The data stored in columns that directly or indirectly reference the altered type is converted to the latest version format.

Altering the Attributes of an Object Type

```
ALTER TYPE type_name
    {{ADD | DROP | MODIFY}
     ATTRIBUTE attribute_spec }|
    {ADD | DROP} method_spec}
{INVALIDATE|CASCADE [[NOT]INCLUDING TABLE DATA]}
[FORCE] EXCEPTIONS INTO <table_name>;
```

ORACLE

8-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Altering the Attributes and Methods of an Object Type

New Keywords

ADD:	Adds specified method to type
DROP:	Drops the specified method from the target
ADD ATTRIBUTE:	Adds the specified attribute to the target type
DROP ATTRIBUTE:	Drops the specified attribute from the target type
MODIFY ATTRIBUTE:	Modifies the type of the specified attribute
INVALIDATE:	Alters the target type and invalidates all dependent objects without checking dependent types and tables, whether the type change may cause any problems or not
CASCADE:	Alters the target type and propagates the type change to dependent types and tables
INCLUDING TABLE DATA:	Converts data stored in all user-defined columns to the most recent version of the column's type. This is the default when CASCADE is specified.
NOT INCLUDING TABLE DATA:	Leaves the column data unchanged. FORCE forces errors from dependent tables to be ignored.

Table Validation

```
ALTER TABLE table_name
UPGRADE [[NOT] INCLUDING DATA]
[column_storage_clause];
```

ORACLE

8-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Table Validation

New Keywords and Parameters

UPGRADE: Converts the metadata of the target table to conform with the latest version of each referenced type

INCLUDING DATA: Converts data stored in all user-defined columns to the most recent version of the column's type

NOT INCLUDING DATA: Leaves column data unchanged

COLUMN_STORAGE_CLAUSE: Specifies the storage for the new varray, nested table, or LOB attributes to be added to the table

ALTER TYPE Statement Options

ALTER TYPE

INVALIDATE

1

Target Type

CASCADE

NOT

INCLUDING

TABLE DATA

2

Dependent
Types

CASCADE

INCLUDING

TABLE DATA

3

Dependent
Tables

Other
Dependent
Objects

ALTER TYPE Statement Options

INVALIDATE: All dependent schema objects below line 1 are marked invalid.

CASCADE NOT INCLUDING TABLE DATA: All dependent schema objects below line 2 are marked invalid. All dependent tables are upgraded to the latest type version, but the table data is not converted.

Note: Table data is not invalidated. The invalidation of a table always pertains to its metadata, not its data.

CASCADE INCLUDING TABLE DATA: All dependent schema objects below line 3 are marked invalid. All dependent tables are upgraded to the latest type version including the table data.

Data Dictionary Views

USER_PENDING_CONV_TABLES lists all tables that are in an invalid state because of the alteration of a parent type.

USER_TYPE_VERSIONS lists all versions of types.

Propagating Changes to Dependent Tables

Alter a type without converting column data in dependent tables:

```
ALTER TYPE cat_typ  
ADD ATTRIBUTE cat_created DATE  
CASCADE NOT INCLUDING TABLE DATA;
```

Upgrade a table's data to the latest type version:

```
ALTER TABLE cat_tab UPGRADE;
```

ORACLE

8-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Propagating Changes to Dependent Tables

In the above example, a new attribute is added to `cat_typ`. The data in the table is not converted because the option `NOT INCLUDING TABLE DATA` is specified. However, the metadata that defines the structure of dependent tables is updated. This is a good option when there are a large number of dependent tables because during the upgrade all dependent tables will be locked. Instead, by deferring the upgrade, the data of each independent table can be later converted by using the `ALTER TABLE . . . UPGRADE` statement, locking only one table at a time.

Multilevel Collection Types

- **Multilevel collection types are collection types where the collection element itself is directly or indirectly another collection type.**
- **All permutations are possible:**
 - **Nested table of a nested table or varray type**
 - **Varray of a varray or nested table type**
 - **Nested table of an object type containing a collection**
 - **Varray of an object type containing a collection**

ORACLE

8-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Multilevel Collection Types

The definition of a nested table type can specify a collection as its data type. For example:

```
CREATE TYPE nt AS TABLE OF NUMBER;  
CREATE TYPE va AS VARRAY(10) OF DATE;  
CREATE TYPE nt_nt AS TABLE OF nt;  
CREATE TYPE nt_va AS TABLE OF va;
```

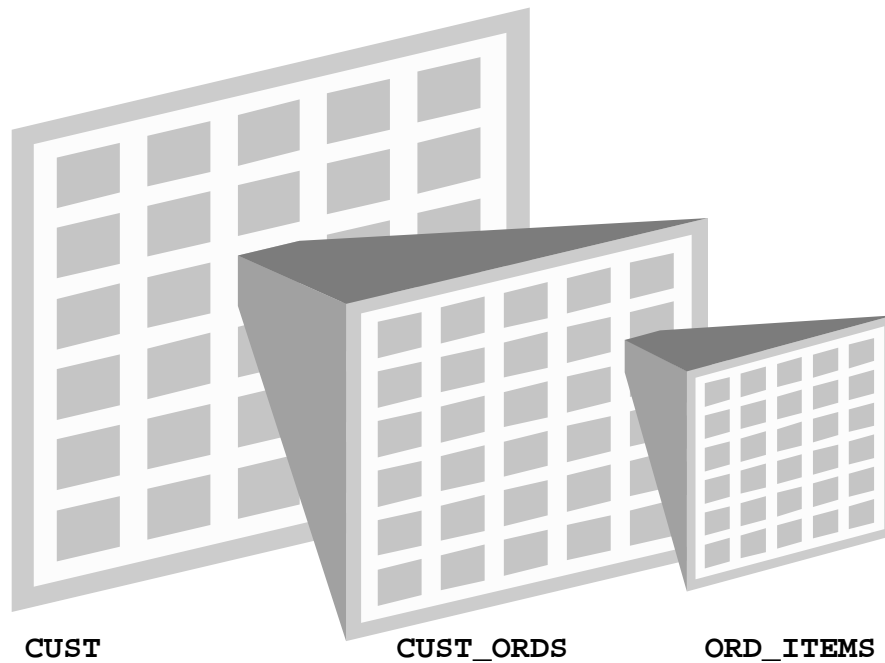
The definition of a varray can specify a collection as its data type. For example:

```
CREATE TYPE va_nt AS VARRAY(10) OF nt;  
CREATE TYPE va_va AS VARRAY(10) OF va;
```

A nested table or varray type can also specify an object type as its data type. The object type can have one or more attributes of (possibly nested) collection types. For example:

```
CREATE TYPE o AS OBJECT  
  ( a number  
    , b date  
    , c nt  
    , d va);  
CREATE TYPE nt_o AS TABLE OF o;  
CREATE TYPE va_o AS VARRAY(10) OF o;
```

Applications of Multilevel Collections



ORACLE

8-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Applications of Multilevel Collections

Multilevel collections can be used to represent nested one-to-many relationships. A one-to-many relationship can be represented by a single collection. If each element of the collection has another one-to-many relationship, then this will require a multilevel collection. This allows the representation of multiple levels of information in one object table.

In the above diagram, CUST is an object table with CUST_ORDS as an attribute. The CUST_ORDS attribute is a nested table of ord objects. ORD_ITEMS is one of the attributes of an ord object. The ORD_ITEMS attribute is itself another nested table of item objects.

In this example, CUST is an object table, but it could also be a relational table. In that case, CUST_ORDS would simply be a nested table column.

Creating Multilevel Collections

```
CREATE TYPE ord_item_typ AS OBJECT
  ( line_item_id      NUMBER(3)
    , unit_price       NUMBER(8,2)
    , quantity         NUMBER(8) ) ;
CREATE TYPE ord_item_list_typ
  AS TABLE OF ord_item_typ;
CREATE TYPE ord_typ AS OBJECT
  ( ord_id            NUMBER(12)
    , ord_item_list    ord_item_list_typ ) ;
CREATE TYPE ord_list_typ
  AS TABLE OF ord_typ;
```

ORACLE

8-31

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Multilevel Collections

Multilevel Collection Constructors

A collection constructor is a function that returns a collection value whose elements are the actual parameters provided to the constructor. Every collection type has a corresponding constructor with the same name called the default constructor. The elements of the constructor of a nested collection are collection constructors as well.

```
DECLARE
  ord_list_typ := ord_list_typ(
    ord_typ (1000, ord_item_list_typ(
      ord_item_typ( 4900, 12, 4),
      ord_item_typ( 2067,  4, 1) ) ),
    ord_typ (1001, ord_item_list_typ(
      ord_item_typ( 3111, 25, 8),
      ord_item_typ( 2309, 10, 4) ) ) );
  ...
```

Assignment and Comparisons

Collection assignment happens during data transfers such as INSERT, UPDATE, and SELECT statements as well as in PL/SQL assignment statements. In collection assignments, both the source and the destination must be of the same data type. Instances of collection types cannot be directly compared in SQL, PL/SQL, or other APIs.

Nested Tables in Multilevel Collections

- The `NESTED TABLE STORAGE` clause must be used to specify the characteristics of the outer and inner collections when they are nested tables.
- A storage table is created for each nested table.
- The physical attributes of the storage table can be specified in the `NESTED TABLE STORAGE` clause.

ORACLE

Varrays in Multilevel Collections

- **A table or a nested table can have columns of a varray type.**
- **A varray column with elements of a collection type is stored in a LOB.**
- **There is no storage table associated with a nested table element within a varray.**

ORACLE

8-33

Copyright © Oracle Corporation, 2001. All rights reserved.

Varrays in Multilevel Collections

A varray collection can be nested in other varray collections or in nested tables. Similarly a nested table can be further nested in a varray. Varrays and nested tables differ in several ways. A table containing a varray column stores the entire varray including any nested collections as a LOB. The individual elements of a varray cannot be updated from SQL. Nested tables, on the other hand, are stored as two separate tables and can be updated easily through SQL DML statements. However a nested table inside a varray column loses some of these characteristics. It cannot be manipulated through DML because the whole varray is stored in a LOB. It also cannot have indexes.

Creating Tables with Multilevel Collections

```
CREATE TABLE cust
  ( customer_id      NUMBER(6)
  ...
  , credit_limit     NUMBER(9,2)
  , cust_ords        ord_list_typ)
NESTED TABLE cust_ords STORE AS ords_tab
  ((PRIMARY KEY (NESTED_TABLE_ID, ord_id))
NESTED TABLE ord_item_list STORE AS
  ord_items_tab );
```

ORACLE

8-34

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Tables with Multilevel Collections

In the above example, ORDS_TAB is a store table that is used to hold the rows corresponding to the ords, while ORD_ITEMS_TAB is used to store the rows for items. A hidden SETID column is created in the CUST table to join the collection elements with the parent row. The ORDS_TAB nested table will include a NESTED_TABLE_ID column that stores the set ID that links its rows to the parent table. Similarly the ORD_ITEMS_TAB nested table will include a NESTED_TABLE_ID column that links its rows to its parent table.

Example

```
CREATE TABLE cust
  ( customer_id      NUMBER(6)
  , cust_first_name  VARCHAR2(20)
  , cust_last_name   VARCHAR2(20)
  , cust_address     VARCHAR2(45)
  , phone_numbers    VARCHAR2(15)
  , credit_limit     NUMBER(9,2)
  , cust_ords        ord_list_typ)
NESTED TABLE cust_ords STORE AS ords_tab
  ((PRIMARY KEY (NESTED_TABLE_ID, ord_id))
  NESTED TABLE ord_item_list STORE AS
    ord_items_tab );
```

Collection Unnesting

- Unnesting is the process of viewing collection data in a flat form
- This can apply to multilevel collections as well as single-level collections
- For example:

```
SELECT *  
FROM cust c,  
      TABLE(c.cust_ords) o,  
      TABLE(o.ord_item_list) i;
```

ORACLE

SYS.AnyType

- The AnyType API defines dynamic types.
- It includes the following program units:
 - BeginCreate() creates a new type of AnyType
 - SetInfo() sets information for transient types
 - AddAttr() adds an attribute to an AnyType
 - EndCreate() ends creation of a transient AnyType
 - GetPersistent() returns a CREATE TYPE type
 - GetInfo() gets type information for the AnyType
 - GetAttrElemInfo() gets type attribute information

ORACLE

SYS.AnyType

BeginCreate() creates a new instance of AnyType which can be used to create a transient type description.

SetInfo() sets any additional information required for constructing a collection or built-in type. It is an error to call this function on an AnyType that represents a persistent user defined type.

AddAttr() adds an attribute to an AnyType.

EndCreate() ends the creation of a transient AnyType. Other creation functions cannot be called after this call.

GetPersistent() returns an AnyType corresponding to a persistent type created earlier using the CREATE TYPE SQL statement.

GetInfo() gets the type information for the AnyType.

GetAttrElemInfo() gets the type information for an attribute of the type TYPECODE_OBJECT or gets the type information for a collection's element type if the SELF parameter is of a collection type.

SYS.AnyData

- **SYS.AnyData defines a single dynamic object.**
- **Two ways to construct:**
 - **Convert from Oracle ORDBMS data type and SYS.AnyData**
 - **Piece by piece construction**
- **Get calls access attributes**

ORACLE

8-37

Copyright © Oracle Corporation, 2001. All rights reserved.

SYS.AnyData

There are two ways to construct an AnyData type:

- **Convert*()** calls enable construction of the AnyData in its entirety with a single call. They serve as explicit CAST functions from any type in the Oracle object relational database management system (ORDBMS) to SYS.AnyData.
- The piece by piece approach uses the following types of calls:
 - **BeginCreate()** begins the construction process
 - **Set*()** calls add attributes
 - **EndCreate()** finishes the construction process

The AnyData has to be constructed or accessed sequentially starting from its first attribute or collection element.

Get*() calls are used to access the attributes of the AnyData object. For piece by piece access of the attributes of objects and elements of collections, the **PieceWise()** call should be invoked prior to **Get*()** calls.

For **Convert***, **Get***, and **Set*** calls, the ***** is replaced with one of the following:

- The name of a built-in data type
- **OBJECT** indicates that an object is being accessed
- **REF** indicates that an object reference is being accessed
- **COLLECTION** indicates that a collection is being accessed

SYS.AnyDataSet

- **SYS.AnyDataSet** defines multiple dynamic objects
- Use **AddInstance()** to add a new instance to an **AnyDataSet**
- Subsequent **Set*()** calls set the current data values
- Similar to **SYS.AnyData.PieceWise()**
- No support for piecewise construction and access of nested objects

ORACLE

8-38

Copyright © Oracle Corporation, 2001. All rights reserved.

SYS.AnyDataSet

The **AnyDataSet** type is constructed value by value sequentially. For each data instance of the type **AnyDataSet**, the **AddInstance()** function need to be invoked to add a new data instance to the **AnyDataSet**. Subsequently, **Set*()** can be called to set each value in its entirety.

The **MODE** of construction or access can be changed to attribute and collection element by making calls to **PieceWise()**:

- If the type of the **AnyDataSet** is **TYPECODE_OBJECT**, individual attributes will be set with subsequent **Set*()** calls, or retrieved with **Get*()** calls.
- If the type of the current data value is a collection type, individual collection elements will be set with subsequent **Set*()** calls, or retrieved with **Get*()** calls.

This call is very similar to **AnyData.PieceWise()** call defined for the type **SYS.AnyData**.

There is no support for piecewise construction and access of nested, that is not top level, attributes that are of object types or collection types.

EndCreate() should be called to finish the construction process. No access calls can be made until construction is complete.

Summary

In this lesson, you should have learned how to:

- **Implement object type inheritance**
- **Evolve the definition of an object type**
- **Create and use multilevel collection types**

ORACLE

8-39

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

With the introduction of inheritance, multilevel collections, type evolution, and their corresponding language APIs, the Oracle9i database provides model completeness, a one-to-one mapping of object model constructs.

Practice 8 Overview

This practice covers the following topics:

- **Evolving the definition of a type**
- **Manipulating data in multilevel collections**
- **Implementing type inheritance**

ORACLE

8-40

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 8 Overview

In this practice you will work with object and collection types in SQL and PL/SQL. You will evolve the definition of an object type, manipulate multilevel collections in SQL and PL/SQL, and learn about type inheritance, substitutability, and dynamic method dispatch by modifying and expanding the implementation of an object type hierarchy.

To perform this practice go to Appendix A, “Practices.”

9

SQL Support in PL/SQL

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

After completing this lesson, you should be able to describe the following PL/SQL enhancements in Oracle9i:

- **CASE statements**
- **Cursor subqueries**
- **Common SQL parser**

ORACLE

9-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

This lesson covers the PL/SQL enhancements which mirror SQL enhancements. You will see how to use some of the new SQL: 1999 syntax in PL/SQL programs. You will learn about cursor subqueries and the advantages of the common SQL parser.

Note: For more information on these topics please refer to *Oracle9i SQL Reference Manual* and *PL/SQL Users Guide*.

Overview

SQL-related enhancements

- **CASE expressions**
- **Cursor subqueries**
- **Common SQL parser**

ORACLE

9-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview

In Oracle9i several enhancements have been made to the SQL language, that are supported in the PL/SQL interface as well. These enhancements include CASE expressions, cursor subqueries, and the common SQL parser.

PL/SQL CASE Statement

- Supports the grammar rules of SQL: 1999
- Allows labels
- Has the following types:
 - CASE expression
 - PL/SQL simple CASE expression
 - PL/SQL searched CASE expression
 - CASE statement
 - PL/SQL simple CASE statement
 - PL/SQL searched CASE statement

ORACLE

9-4

Copyright © Oracle Corporation, 2001. All rights reserved.

PL/SQL CASE Statement

PL/SQL CASE statement supports the grammar rules of SQL CASE statement as specified in SQL: 1999. There is a slight syntax extension to allow PL/SQL CASE statements to be labeled. The label can be optionally used in the final END clause. This is to ensure that they are seamless with other CASE statements.

Types of CASE

- **CASE expression:**
 - Used in a similar way to any SQL expression
 - Used as an assignment variable, within a SQL statement or in a logical operation
 - Starts with **CASE** and terminates with **END**
- **CASE statement:**
 - Used as an independent statement similar to the **IF ... THEN ... ELSE** logical statement
 - Starts with **CASE** and terminates with an **END CASE**

ORACLE

9-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Types of CASE

There are two types of CASE in PL/SQL:

- CASE expression
- CASE statement

CASE expressions can be used just like any other SQL expression. They can be assigned to variables, can be part of SQL statements and can be used in logical expressions. The syntax for the CASE expression starts with **CASE** and terminates with **END**.

CASE statements are independent statements similar to other PL/SQL statements such as **IF...THEN...ELSE** statements. The syntax for this starts with **CASE** and terminates with **END CASE**.

Simple Versus Searched CASE

- **Simple CASE :**
 - Can conditionally evaluate a single variable or expression for multiple values
 - Cannot contain comparison operators in the WHEN clause
- **Searched CASE statement :**
 - Is more flexible
 - Each WHEN clause
 - can have different variables if needed
 - can have comparison operators

ORACLE

9-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Simple Versus Searched CASE

PL/SQL supports both the simple and the searched CASE statements. Simple CASE statements are used to conditionally evaluate a single variable or expression for multiple values. This type of CASE statement does not accept comparison operators in the WHEN clause. Searched CASE statements are more flexible because they can be used for evaluating multiple variables or expressions. Each when clause can evaluate different expressions and can accept comparison operators (>, <, =, BETWEEN, and so on) and logical operators(AND, OR, NOT) .

Simple CASE Expression: Example

```
CREATE OR REPLACE FUNCTION get_comm (p_comm
NUMBER)
RETURN VARCHAR2 IS
  v_comm VARCHAR2(30);
BEGIN
  v_comm :=
    CASE p_comm
      WHEN .15 THEN 'fifteen percent'
      WHEN .2 THEN 'twenty percent'
      ELSE 'thirty percent'
    END;
  RETURN v_comm;
END;
```

ORACLE

9-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Simple CASE Expression: Example

In the above example the CASE expression is being assigned to a variable. In this type of usage, each executable statement following the WHEN clause does not require a semicolon. Also the CASE expression is terminated with just an END statement not an END CASE. The above example evaluates the P_COMM parameter and assigns different values based on the value of the parameter to the V_COMM variable. The above example was written in prior releases as follows:

```
CREATE OR REPLACE FUNCTION get_comm (p_comm NUMBER) RETURN
VARCHAR2 IS
  v_comm VARCHAR2(60);
BEGIN
  IF p_comm = .15 THEN
    v_comm := 'fifteen percent';
  ELSIF p_comm = .2 THEN
    v_comm := 'twenty percent';
  ELSE
    v_comm := 'thirty percent';
  END IF;
  RETURN v_comm;
END;
```

Simple CASE Statement: Example

```
CREATE OR REPLACE FUNCTION get_comm (p_comm
NUMBER) RETURN VARCHAR2 IS
v_comm VARCHAR2(30);
BEGIN
    CASE p_comm
        WHEN .15 THEN
            v_comm := 'fifteen percent';
        WHEN .2 THEN
            v_comm := 'twenty percent';
        ELSE
            v_comm := 'thirty percent';
        END CASE;
    RETURN v_comm;
END;
```

ORACLE

Simple CASE Statement: Example

The above example shows a CASE statement. In this instance the CASE statements end with an END CASE statement and every executable statement following the WHEN clause must be terminated with a semicolon (;). This example is also evaluating the value of P_COMM and assigning different values to the V_COMM variable. In prior releases the above example was coded as:

```
CREATE OR REPLACE FUNCTION get_comm (p_comm NUMBER) RETURN
VARCHAR2 IS
    v_comm VARCHAR2(60);
BEGIN
    IF p_comm = .15 THEN
        v_comm := 'fifteen percent';
    ELSIF p_comm = .2 THEN
        v_comm := 'twenty percent';
    ELSE
        v_comm := 'thirty percent';
    END IF;
    RETURN v_comm;
END;
```

Searched CASE Expression: Example

```
CREATE OR REPLACE FUNCTION get_sal_grade (p_sal
NUMBER)
  RETURN VARCHAR2 IS
  v_grade VARCHAR2(35);

BEGIN
  v_grade :=
    CASE
      WHEN p_sal < 10000 THEN
        'salary is low'
      WHEN p_sal BETWEEN 10000 AND 50000 THEN
        'salary is average'
      ELSE
        'salary is high'
    END;
  RETURN v_grade;
END;
```

ORACLE

Searched CASE Expression: Example

In a searched CASE expression each WHEN clause can have a different conditional expression. The conditions can take the form of <VARIABLE_EXPRESSION> OPERATOR <VALUE_EXPRESSION>. The operator can be any comparison operator that can be used in PL/SQL. The searched CASE expression can be assigned to a variable just like the simple CASE expression. It also terminates with an END keyword and does not take a semicolon (;) after each executable statement. In the above example the function GET_SAL_GRADE accepts the salary and returns the salary grade based on the CASE expression.

Searched CASE Statement: Example

```
CREATE OR REPLACE FUNCTION get_raise (p_sal NUMBER)
RETURN VARCHAR2 is
    v_raise VARCHAR2(30);
BEGIN
    CASE
        WHEN p_sal < 10000 THEN
            v_raise := 'raise is 10%';
        WHEN p_sal >= 10000 THEN
            v_raise := 'raise is 15%';
        END CASE;
    RETURN v_raise;
END;
```

ORACLE

Searched CASE Statement: Example

The searched CASE statement is similar to the searched CASE expression in syntax, except that it is an independent statement and cannot be assigned to variables. In the above example the function GET_RAISE computes the raise based on the CASE statement which evaluates salary values.

NULLIF and COALESCE Expressions

- **PL/SQL NULLIF and COALESCE are forms of shorthand for PL/SQL CASE expressions**
- **The NULLIF behaves like an inverse of the NVL function**
- **The COALESCE behaves similar to the NVL function but can take a list of values**

ORACLE

9-11

Copyright © Oracle Corporation, 2001. All rights reserved.

NULLIF Expression

The semantics of PL/SQL NULLIF expression are defined by rewriting it as:

```
CASE
    WHEN expression_1 = expression_2 THEN NULL;
    ELSE expression_1;
END;
```

COALESCE Expression

When a PL/SQL COALESCE expression has exactly two arguments, the semantics are defined by rewriting it as:

```
CASE
    WHEN expression_1 IS NOT NULL THEN expression_1
    ELSE expression_2
END;
```

When COALESCE has three or more arguments, it has the form COALESCE (expression_1, expression_2, expression 3, . . . , expression_n) the semantics are defined by rewriting it as:

```
CASE
    WHEN expression_1 IS NOT NULL THEN expression_1;
    ELSE COALESCE (expression_2, expression_3,...,
expression_n);
END CASE;
```

Using NULLIF in PL/SQL

```
CREATE OR REPLACE PROCEDURE get_comm_pct (  
  p_comm NUMBER) is  
  v_result VARCHAR2(10);  
BEGIN  
  v_result := NULLIF(p_comm, .2 );  
  
  DBMS_OUTPUT.PUT_LINE ('The commission percent  
is ' || v_result*100 || '%');  
END;  
  
EXEC get_comm_pct(.2);  
  
The commission percent is %  
  
PL/SQL procedure successfully completed.
```

ORACLE

9-12

Copyright © Oracle Corporation, 2001. All rights reserved.

PL/SQL NULLIF Expression

In the above example, you are creating a GET_COMM_PCT procedure which is evaluating the value of the parameter P_COMM. If the value of P_COMM is zero, it returns a NULL; otherwise it returns the current value of P_COMM. The above example can be rewritten as follows:

```
CREATE OR REPLACE PROCEDURE get_comm_pct (p_comm NUMBER) is  
  v_result NUMBER;  
BEGIN  
  v_result :=  
  CASE p_comm  
    WHEN .2 THEN NULL  
    ELSE p_comm  
  END ;  
  DBMS_OUTPUT.PUT_LINE (' v_result is ' || v_result *100 || '%');  
END;  
  
Exec get_comm_pct(.2)  
  
v_result is %  
  
PL/SQL procedure successfully completed.
```


Using COALESCE in PL/SQL

```
CREATE OR REPLACE PROCEDURE get_comm_pct (p_comm
NUMBER) is
v_result number;

BEGIN
    v_result := COALESCE( p_comm, 0);

    DBMS_OUTPUT.PUT_LINE
        ('The commission percent is '
         ||v_result*100||'%');
END;
/
EXEC get_comm_pct(.2);

The commission percent is 20%
PL/SQL procedure successfully completed.
```

ORACLE

9-13

Copyright © Oracle Corporation, 2001. All rights reserved.

PL/SQL COALESCE Expression

The above example can be rewritten as follows:

```
CREATE OR REPLACE PROCEDURE get_comm_pct (p_comm NUMBER) is
    v_result number;

BEGIN
    CASE
        WHEN p_comm IS NOT NULL THEN v_result := p_comm;
        ELSE v_result := 0;
    END CASE ;
    DBMS_OUTPUT.PUT_LINE (' v_result is ' || v_result *100 ||'%');
END;

Exec get_comm_pct(.2)
v_result is 20%
PL/SQL procedure successfully completed.
```

Overview of Cursor Subquery

- **The cursor subquery opens a nested cursor for each evaluation of the cursor expression.**
- **For each row of the parent query, it returns a cursor in the `SELECT` list corresponding to the cursor expression.**
- **Fetches have to be performed to retrieve the results of a subquery.**

ORACLE

9-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Cursor Subquery

The cursor subquery expressions open a nested cursor for each evaluation of the cursor expression. That is, for each row of parent query, it returns a cursor number for the value of the select item corresponding to the cursor expression. The user must perform fetches to retrieve the results of the subquery. A cursor is opened for each evaluation of cursor subquery expression but instead of parsing, executing, and so on, a handle is associated with the cursor. Later, operations such as describe and fetch on nested cursor will do the implicit preparation of the nested cursor. This way, SQL engine minimizes the cost that would have otherwise incurred if all the nested cursors were parsed and executed.

The cursor subqueries are supported by SQL engine since Oracle8.0, but not supported by PL/SQL until Oracle9i. Since PL/SQL allows you to open cursors in many ways and it also allows REF cursors, the PL/SQL implementation also supports cursor using all these possible constructs. These will be supported on both client side and server side in PL/SQL.

In PL/SQL, cursor subquery is allowed in the following contexts:

REF cursor

Explicit cursor

Cursor subqueries are not allowed in implicit cursors.

Note: Cursor subqueries were introduced in SQL in Oracle 8.0 as cursor expressions. Please refer Oracle9i SQL reference manual for more information on this feature.

Cursor Subquery in Ref Cursor: Example

```
DECLARE
    TYPE refcursortype IS REF CURSOR;
    n NUMBER;
    emprow employees%rowtype;
    empcurl refcursortype;
    empcur2 refcursortype;

BEGIN
    OPEN empcurl FOR
        SELECT cursor(SELECT * FROM employees)
            FROM departments;
    FETCH empcurl INTO empcur2;
    FETCH empcur2 INTO emprow;
    ...
    CLOSE empcurl;
END;
```

ORACLE

Cursor Subquery in Ref Cursor: Example

If a REF cursor is bound to a cursor subquery as shown in the above example, a subsequent fetch from this REF cursor is another REF cursor.

In the above example `employees` and `department` are existing tables. The REF cursors, with which the results of the cursor subquery are fetched, should be weak REF cursors. Strong REF cursors are not supported. The above cursor subquery yields a Cartesian product. The fetches are required to bring the data into variables. Because each fetch brings only one row, we still require LOOP statements to process multiple rows. The above code is a partial example.

Note: This feature was included for SQL as cursor expressions in Oracle 8.1.7.

Cursor Subquery in Explicit Cursor: Example

```
DECLARE
    TYPE refcursortype IS REF CURSOR;
    empcurl refcursortype;
    CURSOR c1 is SELECT department_name,
        CURSOR(SELECT employee_id FROM employees e
            WHERE
                e.department_id =d.department_id)
        FROM departments d;
    emprow employees%rowtype;
    v_dname departments.department_name%type;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO v_dname,empcurl;
        FETCH empcurl into emprow;
        ...
    END LOOP;
    CLOSE C1;
END;
```

ORACLE

Cursor Subquery in Explicit Cursor: Example

If an explicit cursor is bound to a cursor subquery, as shown above, then a subsequent fetch returns another REF cursor that can be used to fetch the results from the result of the subquery. In the above example you are using a cursor subquery with correlation variables to get the department names as well as the employee information for employees in each department. The outer query is the only query that needs to be opened explicitly. The cursor subquery can be fetched while the outer query is still open. In the above example the cursor subquery behaves similar to a correlated subquery.

Note: The above code is a partial example.

Cursor Subquery: Nested Cursor Attributes

- **A nested cursor is prepared the first time an operation is performed on it.**
- **Resources are released after all the nested cursors associated with this cursor have been prepared.**
- **The parent is checked at each preparation to verify if this is the last nested cursor so that resources can be released.**

ORACLE

Introducing the Common SQL Parser

The Common SQL Parser:

- Reduces duplication of SQL analysis
- Allows PL/SQL to pick up new SQL features as they are implemented in the RDBMS

ORACLE

9-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Introducing the Common SQL Parser

The common SQL parser replaces PL/SQL's compile-time analysis of a static SQL statement with analysis using a SQL component shared with the RDBMS. This reduces duplication of SQL analysis, allowing PL/SQL to pick up new SQL features as they are implemented in the RDBMS and eliminating bugs due to differences in SQL analysis between SQL and PL/SQL.

Visible Effects of Common SQL Parser

- **Changes PLS-error messages to ORA-error messages**
- **Stricter checking in the common SQL parser may cause compile errors during upgrades from prior releases:**
 - **These errors are indications of poor code and should be fixed.**
 - **If source code is unavailable, common SQL parser should be disabled to compile these programs.**

ORACLE

9-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Support for Common SQL Parser

Error Messages

A number of error messages raised during the analysis of static SQL statements change from PLS_errors to ORA-errors. The common SQL parser raises errors during SQL compilation that PL/SQL analysis did not catch. This allows earlier detection of static SQL errors.

Compatibility and Migration

During an upgrade to Oracle9i from a previous version of Oracle, PL/SQL program units compiled correctly in the previous version may compile with errors in static SQL statements in Oracle9i. These errors are a result of the stricter compile-type checking enabled by the common SQL parser. It is recommended that the user update the PL/SQL code to eliminate any new errors. If the users do not have access to the source code, they can disable the common SQL parser for the program unit containing errors using the compile-time switch. However disabling the common SQL parser will cause all of the new SQL extensions such as SQL:1999 compliant joins to not compile within PL/SQL as these new features are only available with the common SQL parser.

Other PL/SQL Language Enhancements

- **Support for new datetime data types including:**
 - `TIMESTAMP`
 - `TIMESTAMP WITH TIME ZONE`
 - `INTERVAL DAY TO SECOND`
 - `INTERVAL YEAR TO MONTH`
- **Support for SQL Unicode enhancements**
- **Seamless support of BLOB and CLOB data types**
- **Support for object type evolution and single inheritance**
- **Support for multilevel collections**

ORACLE

9-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Other PL/SQL Language Enhancements

Oracle9i SQL and PL/SQL include support for the SQL: 1999 `TIMESTAMP` and `INTERVAL` data types. In addition, Oracle database supports `TIMESTAMP WITH LOCAL TIME ZONE` and daylight savings time. Various operators and functions have been added to support these new data types. Oracle9i SQL and PL/SQL provide support for implicit Unicode conversion.

Note: For more information on these features, refer to the lesson on Globalization Enhancements in this course.

Support for LOBs in the language is seamless. String built-ins (such as `SUBSTR` and so on) now work on LOB arguments also. `CHAR` and `VARCHAR` variables can be assigned back and forth to LOB variables.

Note: For more information on these features, refer to the lesson on LOB migration in this course.

Oracle9i includes support for the creation of subtypes of object types and the creation of type hierarchies based on a single inheritance model. Subtypes can be used in PL/SQL wherever object types can appear. PL/SQL also supports substitutability and dynamic dispatch. Oracle9i includes support for type evolution, that is, object types can be evolved by adding and modifying attributes and methods. Persistent data evolves to match the updated type structure.

Note: For more information on these features, refer to the Objects lesson in this course.

Transparent Performance Enhancements

- There is a 60% plus reduction in overhead of calling PL/SQL procedures from SQL statements.
- For instance, the `BONUS_FN` function listed below executes much faster in Oracle9i.

```
CREATE FUNCTION bonus_fn(p_salary NUMBER)
  RETURN NUMBER is
BEGIN
  RETURN p_salary * (.15);
END;

SELECT salary, bonus_fn(salary) from employees;
```

ORACLE

Summary

In this lesson, you should have learned about:

- **CASE Statements**
- **Cursor subqueries**
- **Common SQL parser**

ORACLE

9-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned about the SQL support in PL/SQL. You have seen how to use CASE expressions and CASE statements in PL/SQL programs. You have seen the usage and functionality of NULLIF and COALESCE expressions. You have learned the advantages of the common SQL parser. You have seen how to use cursor subqueries in PL/SQL.

Practice 9 Overview

This practice covers the following topics:

- Using PL/SQL `CASE` statements
- Using cursor subqueries

ORACLE

9-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing This Practice

To perform this practice go to Appendix A, “Practices.”

10

Performance Enhancements in PL/SQL

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Understand the PL/SQL performance enhancements in Oracle9i**
- **Describe native compilation**
- **Use bulk binds in native dynamic SQL**
- **Create table functions**
- **Describe the transparent performance improvements**

ORACLE

10-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

In this lesson you will learn about the benefits of native compilation. You will also see the enhancements to bulk binding and native dynamic SQL. Additionally, you will be introduced to the functionality and benefits of PL/SQL table functions.

Note: For more information on these topics please refer to the Oracle9i PL/SQL user guide and reference and data warehousing reference manuals.

Overview of Native Compilation of PL/SQL

- In Oracle9i, a PL/SQL library unit can be compiled as native code.
- It is then stored as a shared library in the file system.
- When a user session references a natively compiled PL/SQL library unit, the compiled program is located in the Process Global Area (PGA).
- The instantiation of natively compiled PL/SQL packages is handled in exactly the same manner as the packages compiled to bytecode.

ORACLE

10-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Native Compilation of PL/SQL

A PL/SQL library unit is compiled into bytecode or bytecode (machine dependent code) and the bytecode is loaded into the library cache part of the shared pool in the System Global Area (SGA). When multiple users connected to an Oracle instance execute the same PL/SQL program, they share a single copy of the compiled PL/SQL program. Each user session gets its own private area to store values specific to the session such as the values of package variables (the package instantiation). The location of this session-specific private area will be in the Process Global Area (PGA) of the user if the session is established through a dedicated server. It is in the SGA in the case of a multithreaded server.

When a PL/SQL library unit is compiled native, it is stored as a shared library in the file system. When a user session references a natively compiled PL/SQL library unit, the corresponding shared library is mapped to the virtual address space of the Oracle process. That means the compiled program is located in the PGA. When multiple users reference the same natively compiled PL/SQL library unit, the shared library is mapped to the virtual address space of each Oracle process. But by virtue of being a shared library, there is a single copy of the shared library in the physical memory. The instantiation of natively compiled PL/SQL packages is handled in exactly the same manner as the packages compiled to bytecode.

This only works if a C compiler is available on the database server.

New Parameters for Native Compilation

- **PLSQL_COMPILER_FLAGS**
- **PLSQL_NATIVE_LIBRARY_DIR**
- **PLSQL_NATIVE_MAKE_UTILITY**
- **PLSQL_NATIVE_MAKE_FILE_NAME**

ORACLE

10-4

Copyright © Oracle Corporation, 2001. All rights reserved.

New Parameters for Native Compilation

The new `INIT.ORA` parameters in Oracle9i are:

PLSQL_COMPILER_FLAGS is a parameter used by the PL/SQL compiler. It specifies a list of compiler flags as a comma separated list of strings. The parameter can be set at a system wide level using the `ALTER SYSTEM` command. The `ALTER SESSION` command can be used to change the value of this parameter for a specific session. If `INTERPRETED` is specified as one of the flags, then PL/SQL modules will be compiled to PL/SQL bytecode format. Such modules are executed by the PL/SQL interpreter engine. This is the default PL/SQL compilation mode in releases prior to Oracle9i. If `NATIVE` is specified as one of the flags, then PL/SQL modules (with the exception of top-level anonymous PL/SQL blocks) will be compiled to native (machine) code. Such modules will be executed natively without incurring any interpreter overhead. If `DEBUG` is specified as one of the flags, then PL/SQL modules will be compiled with `PROBE` debug symbols.

PLSQL_NATIVE_LIBRARY_DIR is a parameter used by the PL/SQL compiler. It specifies the name of a directory where the shared objects produced by the native compiler are stored. A parameter must specify such a directory so that the database administrator (DBA) can choose the location of the shared objects produced by the native compiler for each module as the need arises. This parameter can only be modified by the DBA through an `ALTER SYSTEM` command or specifying a value in the `init.ora` file before the instance is started. These directories are created by the DBA/system administrator on the server.

New Parameters for Native Compilation (continued)

PLSQL_NATIVE_MAKE_UTILITY is a parameter that specifies the full path name of a make utility such as make in UNIX or gmake (GNU make). The make utility is needed to generate the shared object or dynamic-link library (DLL) from the generated C source. This parameter can only be modified by the DBA through an ALTER SYSTEM command or specifying a value in the `init.ora` file before the instance is started.

PLSQL_NATIVE_MAKE_FILE_NAME is parameter that specifies the full path name of a make file. The make utility (specified by the `PLSQL_NATIVE_MAKE_UTILITY`) uses this make file to generate the shared object or DLL. A port specific make file is shipped for each platform which contains the rules for the make utility to generate DLLs on that platform. This parameter can only be modified by the DBA through an ALTER SYSTEM command or specifying a value in the `init.ora` file before the instance is started. The make file is usually found in the `$ORACLE_HOME/plsql` directory. The name of the file is `spnc_makefile.mk`.

Steps for Enabling Native Compilation

```
1. ALTER SYSTEM SET PLSQL_NATIVE_LIBRARY_DIR =  
   '/orahome/dbs/native_plsql';  
  
2. ALTER SYSTEM SET PLSQL_NATIVE_MAKE_UTILITY =  
   '/usr/local/bin/make';  
   ALTER SYSTEM SET PLSQL_NATIVE_MAKE_FILE_NAME  
   = '/orahome/admin/spnc_makefile.mk';  
  
--turn the feature on  
3. ALTER SESSION SET  
   plsql_compiler_flags = 'NATIVE';  
  
--recompile programs  
4. ALTER FUNCTION emp_upd COMPILE;  
  
--turn off the feature  
5. ALTER SESSION SET  
   plsql_compiler_flags = 'INTERPRETED';
```

ORACLE

Enabling Native Compilation

1. The first parameter is required:
`ALTER SYSTEM SET PLSQL_NATIVE_LIBRARY_DIR=
'/orahome/dbs/native_plsql';`
2. This set of parameters is mandatory:
`ALTER SYSTEM SET PLSQL_NATIVE_MAKE_UTILITY= '/usr/local/bin/make';
ALTER SYSTEM SET PLSQL_NATIVE_MAKE_FILE_NAME=
'/orahome/admin/spnc_makefile.mk';`
3. To turn the feature on:
`ALTER SESSION SET PLSQL_COMPILER_FLAGS= 'NATIVE' ;`
4. To turn the feature off:
`ALTER SESSION SET PLSQL_COMPILER_FLAGS= ' INTERPRETED' ;`

Subsequent compiles in the same session after one of these ALTER SESSIONs is executed picks up the last value of PLSQL_COMPILER_FLAGS. Implicit recompiles of the same program are done with whatever setting was in effect when it was last compiled. To be sure that the process worked, you can query the data dictionary to see that a procedure is compiled for native execution. To check whether an existing procedure is compiled for native execution or not, you can query the data dictionary views USER[DBA/ALL]_STORED_SETTINGS. For example, to check the status of the MY_PROC procedure, you could enter:

```
SELECT param_value FROM user_stored_settings WHERE  
param_name = 'PLSQL_COMPILER_FLAGS'  
and object_name = 'MY_PROC';
```

The PARAM_VALUE column has a value of NATIVE for procedures that are compiled for native execution, and INTERPRETED otherwise.

Performance

Function	Interpreted	Native
Java call	0.9	0.1
Towers of Hanoi	45.26	40.28
Fibonacci	18.97	8.32
Native Dynamic SQL	2.93	2.83

ORACLE

10-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Performance

The above slide shows the improvement in performance between native and interpreted compilation for different types of programs. The numbers represents the number of seconds or fractions of seconds that it took to complete these programs.

Benefits of Native Compilation of PL/SQL

- **Faster execution of PL/SQL programs by generating native C code instead of bytecode**
- **Improvement in execution speed due to the following factors:**
 - **Eliminating the overhead associated with interpreting byte code**
 - **Control flow is much faster in native code than in interpreted code**
 - **Compiled code corresponding to a PL/SQL program is mapped to a PGA as opposed to SGA**
 - **PL/SQL that does not contain SQL references is 2-10 times faster**

ORACLE

10-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Benefits of Native Compilation of PL/SQL

A PL/SQL program is compiled to native code in two phases: the program is translated to C code which is subsequently compiled to native code. For Oracle supplied packages, the compiled code is statically linked to the Oracle executable whereas for end-user PL/SQL programs, the compiled code is dynamically linked to an Oracle process.

A C compiler is required for the feature to be implemented. Native compilation provides an improvement in execution speed due to the following factors:

- The overhead associated with interpreting the bytecode is eliminated.
- Control flow is much faster in native code than in interpreted code. This is because jumps are label-based on C code. Function calls to targets in the same compilation unit are mapped to C function calls. The cost of setting up a frame is also cheaper because the primary memory for the frame is allocated on the C stack, as opposed to the PGA in the case of bytecode. Exception handling is also much faster in native code because exceptions are implemented as jumps to switch statements containing the exception handler code. There is no run-time overhead associated with looking up an exception handler table as in the case of bytecode based exception handling.

Benefits of Native Compilation of PL/SQL (continued)

- The compiled code corresponding to a PL/SQL program is mapped into the PGA, as opposed to the SGA into which the bytecode is loaded. This should result in less contention for SGA and thus better scalability. Also in the case of large PL/SQL programs, the startup time should be faster on demand paged systems. The bytecode is loaded in its entirety into SGA. The constant pool is also mapped into the PGA as constant data.
- Constant pool items, such as string literals, can be arbitrarily long as the PGA is paged by the operating system. Similarly, the handle segment need not be paged in the case of native compilation. Although, PL/SQL execution is 2 to 10 times faster, it does not speed up SQL execution.

Restrictions to Native Compilation of PL/SQL

- **Package and type specification and body must be compiled in the same mode.**
- **Individual functions or procedures in a package cannot be compiled native.**
- **Anonymous blocks are always compiled for interpreted execution.**

ORACLE

10-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Restrictions to Native Compilation of PL/SQL

Granularity of Native Compilation

The granularity of native compilations is a library unit or compilation unit, that is, a package specification, a package body, a top-level procedure or function. Individual functions or procedures in a package cannot be compiled native. Package and type specification and body must be compiled in the same mode. If the specification of a PL/SQL package is compiled for native (interpreted) execution, then the body must also be compiled for native (interpreted) execution. This restriction enables faster procedure calls between natively compiled libunits.

Note: This restriction can be relaxed, if it does not produce a significant performance advantage.

Anonymous Blocks

Anonymous blocks are always compiled for interpreted execution. You do not want to leave disk footprints, namely DLLs, for anonymous blocks. This creates additional complications in terms of naming the anonymous DLLs, and in terms of clean up of these DLLs in case of a server crash, and so on.

Overview of Oracle9i Bulk Bind Enhancements

- Bulk bind features have been enhanced in Oracle9i to support more efficient and convenient bulk bind operations.
- Restrictions on usage of collections in `SELECT` and `FETCH` clauses have been removed.
- Error handling for failure in bulk binds has been provided.

ORACLE

10-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Oracle9i Bulk Bind Enhancements

Bulk bind features of PL/SQL in Oracle9i have been enhanced to support more convenient and efficient bulk bind operations and provide an error handling mechanism. Previously bulk bind operations, such as `FORALL INSERT/UPDATE/DELETE`, stopped immediately whenever there was an error during its execution and an exception would then be raised. In certain applications it is better to handle the exception and continue processing. An error handling mechanism has been incorporated in Oracle 9i so that errors during a bulk bind operation are collected and returned together when the operation completes.

FORALL Statement Enhancements

```
FORALL index IN lowerbound .. upperbound  
  [SAVE EXCEPTIONS]  
  DELETE/UPDATE/INSERT statement
```

ORACLE

10-13

Copyright © Oracle Corporation, 2001. All rights reserved.

FORALL Statement Enhancements

The keywords `SAVE EXCEPTIONS`, in the `FORALL DELETE/UPDATE/INSERT` statements, are required if the users want the bulk bind operation to be completed regardless of the occurrences of errors. Errors occurring during the execution are saved in the new cursor attribute, `%BULK_EXCEPTIONS`. `%BULK_EXCEPTIONS` is a bulk attribute of a collection of records which have two integer attributes. The first attribute, `index` is used to store the corresponding SQL error code, `SQLCODE`. The user can get the corresponding SQL error message by calling `SQLERRM` with the SQL error code as the parameter. The number of errors is saved in the count attribute of `%BULK_EXCEPTIONS`, that is, `%BULK_EXCEPTIONS.COUNT`. The subscripts of `%BULK_EXCEPTIONS` are from 1 to `%BULK_EXCEPTIONS.COUNT`.

Without the keywords, `SAVE EXCEPTIONS` in a `FORALL` statement, `DELETE/UPDATE/INSERT` statement inside the `FORALL` stops whenever an error occurs. In this situation, `SQL%BULK_EXCEPTIONS.COUNT` is one and `SQL%BULK_EXCEPTIONS` contains one record.

If there is no error at all during the execution of a `FORALL` statement, `%BULK_EXCEPTIONS.COUNT` returns zero.

The values of this new cursor attribute always refer to the most recently executed `FORALL` statement. `%BULK_EXCEPTIONS` cannot be assigned to other collections. Also, it cannot be passed as a parameter to subprograms.

Error Handling for Bulk Binds

- In prior releases, exceptions occurring during the `FORALL` statement stopped the program execution immediately and raised an exception.
- The new error handling mechanism saves the error information and continues execution.
- All the errors occurring during the execution are returned together after the program completes.

ORACLE

Example of Exception Handling

```
DECLARE

TYPE numtab IS table of number;
v_discount numtab:=
    numtab(10,0,15,20,25,0,60,80,34,0);
BEGIN

    FORALL i IN v_discount.first .. v_discount.last
        SAVE EXCEPTIONS
            UPDATE disc_products
                set prod_min_price = prod_list_price -
                    prod_list_price/v_discount(i);
```

ORACLE

10-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Exception Handling

In the above example, the FORALL statement is used to update products and set prices to the discounted price. Errors will occur in this program because of the presence of zeros in the table variable that is being used for computing the discounts. In earlier releases the occurrence of the first error would terminate the program. In Oracle9i, all exceptions are saved using the SAVE EXCEPTIONS statement and the program is allowed to complete. The following slide shows how the errors and error messages can be output.

Example of Exception Handling

```
EXCEPTION
WHEN OTHERS THEN

    dbms_output.put_line('Total number of errors
                          is ' || SQL%BULK_EXCEPTIONS.COUNT);

    For i in 1 .. SQL%BULK_EXCEPTIONS.COUNT LOOP

        dbms_output.put_line('Error ' || i || 'occurred at
                              iteration' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX
                              );
        dbms_output.put_line('SQL Error Code is ' ||
                              SQL%BULK_EXCEPTIONS(i).error_code ||
                              '[Error: ' || SQLERRM
                              (SQL%BULK_EXCEPTIONS(i).error_code) || ']' );
    END LOOP;
END;
```

ORACLE

10-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Exception Handling (continued)

In the example above, errors, whose SQL error codes are 1476 (zero_divide), had occurred when $i = 0$. The bulk bind operation would finish the whole operation and then return all of the errors together into the new cursor attribute, %BULK_EXCEPTIONS.ERROR.

The output of the above program is the following:

Total number of errors is 3

Error 1 occurred at iteration 2

SQL Error Code is 1476 [Error: ORA-01476:divisor is equal to zero]

Error 2 occurred at iteration 6

SQL Error Code is 1476 [Error: ORA-01476:divisor is equal to zero]

Error 3 occurred at iteration 10

SQL Error Code is 1476 [Error: ORA-01476:divisor is equal to zero]

Benefits of Bulk Bind Enhancement

- **Allows manipulation of an entire collection of rows in a single DML statement**
- **Reduces the number of database calls**
- **Allows applications with long transactions to execute with fewer iterations**
- **Facilitates usage of PL/SQL records and tables and improves performance**

ORACLE

Overview of Bulk Dynamic SQL

Native dynamic SQL which was introduced in Oracle8i, now supports:

- **BULK FETCH**
- **BULK EXECUTE IMMEDIATE and BULK EXECUTE**
- **FORALL, COLLECT INTO, and RETURNING INTO**

ORACLE

10-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Bulk Dynamic SQL

The following functionality has been introduced for Bulk SQL in embedded dynamic SQL:

- BULK FETCH support for cursors opened on dynamic SQL statements
- BULK EXECUTE IMMEDIATE and EXECUTE for dynamic SQL
- FORALL statement and COLLECT INTO and RETURNING INTO clauses are extended to support dynamic SQL statements

Bulk Bind with Dynamic SQL: Example

```
DECLARE
    TYPE num_tab IS TABLE OF NUMBER;
    ids num_tab;
BEGIN
    EXECUTE IMMEDIATE 'SELECT employee_id FROM
employees WHERE ROWNUM <=10'
        BULK COLLECT INTO ids;
END;
```

ORACLE

10-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Bulk Bind for Defining Variables: Example

In the above example you are using `EXECUTE IMMEDIATE` to execute a query to retrieve several employee IDs with one call to the database.

Note: In the above example `num_tab` is an object type created as follows:

```
CREATE TYPE num_tab AS TABLE OF NUMBER;
```

Bulk Bind for Input Variables: Example

```
DECLARE
TYPE num_tab IS TABLE OF NUMBER;
TYPE char_tab is TABLE OF VARCHAR2(30);
  ids num_tab := num_tab (12,13,14,15);
  names char_tab := char_tab('R/D','IT','GL','PR');
BEGIN
  FORALL iter IN 1..4
    EXECUTE IMMEDIATE
      'INSERT INTO departments(department_id,
                             department_name)VALUES(:1, :2)'
    USING ids(iter), names(iter);
END;
```

ORACLE

10-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Bulk Bind for Input Variables: Examples

Input bind variables of a SQL statement can be bound with the FORALL control statement and USING clause. In the above example, bulk binding is used to execute an INSERT statement. You are using EXECUTE IMMEDIATE to execute an INSERT statement passing two collection variables IDs and names containing all the department ID and department name values. This method creates several rows using one INSERT statement with one call to the database.

Bulk Binding In Output Variables: Example

```
DECLARE
    TYPE num_tab IS TABLE OF NUMBER;
    sql_str VARCHAR2(200);
    v_sal NUMBER := 10000;
    saltab num_tab;
BEGIN
    sql_str := 'UPDATE employees SET salary = :1
                WHERE department_id = 10
                RETURNING salary INTO :2';
    EXECUTE IMMEDIATE sql_str
        USING v_sal RETURNING BULK COLLECT INTO
saltab;
END;
```

ORACLE

10-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Bulk Bind for Output Variables: Examples

Only DML statements have out binds. In bulk bind output bind variables must be bound through the RETURNING BULK COLLECT INTO clause. In the above example you are using native dynamic SQL to execute an UPDATE statement passing a variable V_SAL for the value to be used for the update. All the updated salaries are then returned using a collection saltab variable.

Benefits of Bulk Dynamic SQL

- **The main benefit of bulk dynamic SQL is the improvement in performance.**
- **Multiple rows can be processed in a single DML statement by using bulk SQL.**
- **The number of context switches between SQL statement executor and the PL/SQL engine is reduced.**
- **This allows for faster execution of PL/SQL programs.**

ORACLE

Overview of Table Functions

- **Oracle9i supports pipelined and parallelizable table functions:**
 - Inputs can be a stream of rows
 - The output can be pipelined
 - Evaluation of the table function can be parallelized
- **Table functions can be defined in PL/SQL using a native PL/SQL interface or in Java or C using the Oracle Data Cartridge Interface (ODCI).**

ORACLE

PL/SQL Table Functions

- New `PIPE` instruction in PL/SQL
- Returns a single result row
- Suspends the execution of the function
- Function is restarted when the caller requests the next row

ORACLE

Example of Creating Table Functions

```
CREATE OR REPLACE FUNCTION emp_stage_fn
  (p ref_cur_type) RETURN emp_tab_typ
  PIPELINED
  PARALLEL_ENABLE( PARTITION p BY ANY) IS
  PRAGMA AUTONOMOUS_TRANSACTION;

BEGIN
  FOR rec IN p LOOP
    ...
    PIPE ROW (rec);
  END LOOP;
  RETURN;

END;
```

ORACLE

10-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Creating Table Functions

In the above example REF_CUR_TYPE and EMP_TAB_TYP are user-defined types. REF_CUR_TYPE is a ref cursor type. EMP_TAB_TYPE is an object table. Pipelined functions must have return statements that do not return any values.

The idea is to transform each record corresponding to the cursor passed as the parameter for the function and return the corresponding row to the caller. Note that one record is returned as soon as one row is processed.

The new PIPELINED instruction in PL/SQL:

- Returns a single result row
- Suspends the execution of the function
- Restarts the function when the caller requests the next row

PARALLEL_ENABLE is an optimization hint indicating that the function can be executed in parallel. The function should not use session state, such as package variables, as those variables cannot be shared among the parallel execution servers.

The optional PARTITION argument BY clause can be used with functions that have a REF CURSOR argument type. It lets you define the partitioning of the inputs to the function from the REF CURSOR argument. Partitioning the inputs to the function affects the way the query is parallelized when the function is used as a table function (that is, in the FROM clause of the query). ANY indicates that the data can be partitioned randomly among the parallel execution servers. Alternatively, you can specify RANGE or HASH partitioning on a specified column list.

Using Table Functions

```
INSERT INTO EMPLOYEES
SELECT * FROM
    TABLE (emp_stage_fn
            (cursor(SELECT * FROM employees_src)));
```

ORACLE

10-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Table Functions

Pipelined functions can be used in the FROM clause of SELECT statements. The result rows are retrieved by the Oracle database iteratively from the table function implementation. Multiple invocations of a table function, either within the same query or in separate queries result in multiple executions of the underlying implementation, that is, there is no buffering and reuse of rows. In the above example `employees_src` represents the source table for the employees table. The data from `EMPLOYEES_SRC` is retrieved and formatted for insert, and then inserted into `EMPLOYEES` using the above statement. This can be combined with other data warehousing features such as multitable inserts.

Note: For more information on multitable inserts please refer to the lesson *DML Features for Data Warehousing* in this course.

Advantages of PL/SQL Table Functions

- **Table functions reduce response time by piping the results as soon as they are produced.**
- **Pipelining eliminates the need for buffering the produced rows.**
- **Table functions can return multiple rows during each invocation.**
- **The number of invocations are reduced, thereby improving performance.**

ORACLE

Transparent Performance Enhancements

- **Faster record construction and copying:**
 - **Benchmarks designed to stress the feature improved up to five times.**
 - **Less focused benchmarks improved by five to ten percent.**
- **General improvement to cross-package references:**
 - **Considerable improvement in PL/SQL programs with many dependencies on other PL/SQL programs.**
 - **Benchmarks specifically designed to test this feature showed a speedup of over 50 percent.**
 - **In more generic benchmarks a five percent improvement was seen.**

ORACLE

Summary

In this lesson, you should have learned how to:

- **Use bulk bind enhancements**
- **Describe native compilation of PL/SQL**
- **Create table functions**
- **Describe the transparent performance improvements**

ORACLE

10-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned the benefits of compiling PL/SQL programs using native compilation. You have seen the new syntax additions to bulk binding and native dynamic SQL. You have learned how to create and use table functions.

Practice 10 Overview

This practice covers the following topics:

- **Compiling programs for native compilation**
- **Using `BULK COLLECT` and `SAVE EXCEPTIONS`**
- **Creating table functions**

ORACLE

10-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing This Practice

To perform this practice go to Appendix A, “Practice.”

11

Globalization Support

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Differentiate usage of Unicode database and Unicode data type**
- **Recognize implicit and explicit interoperability between Unicode and nonUnicode data types**
- **Describe byte and character semantics**
- **Describe the multilingual sorting capabilities of an Oracle9i database**

ORACLE

11-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

In this lesson you will learn about Unicode support in Oracle9i and multilingual collation.

Note: For more information on globalization features in Oracle9i please refer to the Oracle9i Globalization Support Guide.

Note: The new date and timezone data types are covered in lesson 6.

Overview of Unicode

- **Contains all major living scripts and supports legacy data**
- **Develops, deploys, and hosts multiple languages in a single instance**
- **Enables world-wide interchange of data**
- **Conforms to international standards**

ORACLE

11-3

Copyright © Oracle Corporation, 2001. All rights reserved.

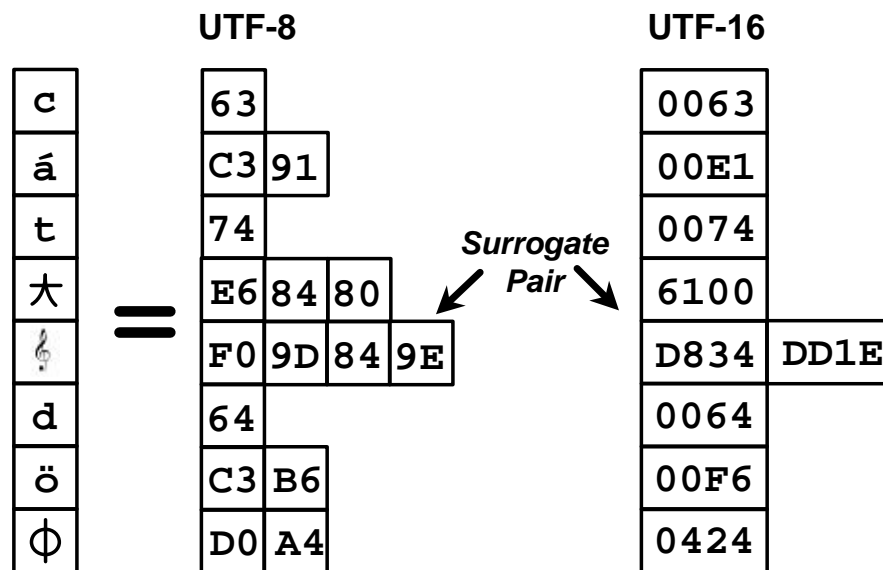
What is Unicode?

Unicode is a universal character encoding scheme that allows you to store information from any major language using a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language.

Oracle started supporting Unicode as a database character set in Oracle7. In Oracle9i, Unicode support has been expanded so that you can find the right solution for your globalization needs. Oracle9i supports Unicode 3.0, the third version of the Unicode Standard. For more information about the Unicode Standard Version 3.0, see *The Unicode Standard Version 3.0*, published by Addison-Wesley, or go to <http://www.unicode.com>.

Unicode specifies which characters are encoded, and several different methods to represent these in binary. The list of defined characters is independent of the method to encode the binary numbers, which is only of interest when exchanging data. This is why there are several Unicode character sets in Oracle9i, one for each binary encoding format, although the list of which characters are represented is the same. There is the varying width byte method, and the fixed-length two byte and four byte methods. Lastly there is a method to extend the first two methods with *surrogate pairs*.

Unicode Encoding



ORACLE

11-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Unicode Encoding

UTF-16 Encoding

This is the 16-bit encoding of Unicode. It is a fixed-width multibyte encoding in which the character codes 0x00 through 0x7F have the same meaning as ASCII. One Unicode character is 2-bytes in this encoding. Characters from both European and Asian scripts are represented in two bytes.



UTF-8 Encoding

This is the 8-bit encoding of Unicode. It is a variable-width multibyte encoding in which the character codes 0x00 through 0x7F have the same meaning as ASCII. One Unicode character can be 1-byte, 2-bytes, or 3-bytes in this encoding. Characters from the European scripts are represented in either one or two bytes, while characters from most Asian scripts are represented in three bytes.

Space Tradeoffs

Your choice of different Unicode character sets will affect the space usage. Storing nonEnglish European strings occupy two bytes. Far East characters occupy three bytes in the varying width character set. In the fixed width every character occupies two bytes. So the proportion of Asian characters is influential. Using a nonUnicode 8 bit character set limits the repertoire of characters, but all characters occupy one byte.

Overview of Oracle9i Unicode Support

- **Extended Unicode Enablement** 
 - Unicode 3.0 support including surrogates
 - Character semantics
 - Unicode Data type
- **Two new Unicode character sets** 
 - AL16UTF16 (Supported through NCHAR data type only)
 - AL32UTF8 (Supported as database character set only)
- **Existing character set**
 - UTF8
 - UTFE
- **Desupported character set**
 - AL24UTFFSS

ORACLE

11-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Oracle9i Unicode Support

Oracle began support for Unicode in Oracle7. Oracle9i supports Unicode 3.0, the third and most recent version of the Unicode standard.

Surrogate Pairs

You can extend Unicode to encode more than one million characters. These extended characters are called surrogate pairs. Surrogate pairs are designed to allow representation of characters in future extensions of the Unicode standard. Surrogate pairs require four bytes in UTF-8 and UTF-16. There are no such assigned characters in the current version of the standard, but it is widely expected that they will be introduced in the near future. One current proposal would assign Western Musical Symbols, like the treble clef shown above, in the surrogate area. UCS-2 is a subset of UTF-16. UTF-16 differs from UCS-2 in that it can access additional encodings with the use of surrogate pairs.

Unicode Character Sets

AL16UTF16 is the fixed width two-byte character set of Unicode. Surrogate pairs will occupy four bytes. It can only be used in the NATIONAL CHARACTER SET. This is also sometimes referred to as UTF16 or UCS-2. Even though this uses more storage for English text compared to the UTF8-based sets, processing will be faster because it is fixed size.

Overview of Oracle9i Unicode Support (continued)

AL32UTF8 is an enhanced version of UTF8. It strictly implements the UTF-8 standard. When defining new databases AL32UTF8 should be used instead of UTF8 if this is required as the database character set.

UTF8 is still supported, to allow migration of databases from Oracle8i or earlier versions. UTFE is still supported, as it is an EBCDIC based variant of UTF8 for use on platforms where EBCDIC is the operating system character set.

AL24UTF8SS is desupported in Oracle9i. This is similar to UTF8 but contains fewer characters and is based on an old Unicode standard.

Unicode Storage

- **You can store Unicode characters in an Oracle9i database in two ways:**
 - **A Unicode database**
 - **Unicode data type**
- **A Unicode database allows you to store Unicode characters as CHAR data types (CHAR, VARCHAR2, CLOB, and LONG).**
- **The Unicode data type allow you to store Unicode characters in columns of NCHAR data types (NCHAR, NVARCHAR2, and NCLOB), irrespective of the database character set.**

ORACLE

11-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Unicode Storage

You can store Unicode characters in an Oracle9i database in two ways:

- Unicode database
- Unicode data types

A Unicode database allows you to store Unicode characters as CHAR data types (CHAR, VARCHAR2, CLOB, and LONG).

The alternative Unicode data types allow you to store Unicode characters in columns of NCHAR data types (NCHAR, NVARCHAR2, and NCLOB), irrespective of the database character set. The NCHAR data type has been redefined in Oracle9i to be a Unicode data type exclusively. In other words, it stores data in the Unicode encoding only. You can use the NCHAR data types in the same way you use the CHAR data types. The Unicode data type (NCHAR) can be used with a nonUnicode database.

The national character set determines the character encoding of the NCHAR data types. In Oracle9i, both UTF8 (UTF-8) and UTF16 (AL16UTF16) Unicode encoding forms are supported as the national character set.

Unicode Solutions: Database

Unicode database

- **Database Character Set**
 - UTF8
 - AL32UTF8
 - UTFE
- **Stores UTF-8 encoded characters in SQL CHAR data types (CHAR, VARCHAR2, CLOB, LONG)**

ORACLE

11-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Unicode Database

The Oracle9i database has the concept of a database character set, which specifies the encoding to be used in the SQL CHAR data types (CHAR, VARCHAR2, CLOB, LONG) as well as the metadata such as table names, column names, and SQL statements. A Unicode database is a database with the UTF-8 encoding scheme as the database character set. There are three Oracle character sets that implement the UTF-8 encoding. The first two are designed for ASCII-based platforms while the third one should be used on EBCDIC platforms: AL32UTF8, UTF8, and UTFE.

Unicode Solutions: Data Type

Unicode data type

- **National Character Set**
 - AL16UTF16 (default)
 - UTF8
- **Stores UTF-16 or UTF-8 encoded characters in SQL NCHAR data types (NCHAR, NVARCHAR2, and NCLOB)**
- **NCHAR length semantics: Character semantics only**

ORACLE

11-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Unicode Data Types

An alternative to storing Unicode data in the database is to use the SQL NCHAR data types (NCHAR, NVARCHAR2, and NCLOB). You can store Unicode characters in columns of these data types irrespective of the database character set. The SQL NCHAR data type has been redefined in Oracle9i to be a Unicode data type exclusively. In other words, it stores data in the Unicode encoding only. You can use the SQL NCHAR data types in the same way that you use the SQL CHAR data types. In other words, it stores data in the Unicode encoding only. NCHAR data types can be used in the same way as the CHAR data types. This allows the inclusion of Unicode data in a nonUnicode database which allows for a phased approach to migration to a full Unicode database.

The NCHAR data type is also suitable for packaged applications because it is a reliable Unicode data type in which the data is always stored in Unicode, and the length of the data is always specified in UTF-16 code units. As a result, the application need only be tested once, and will run on databases of any database character set.

Choosing a Unicode Solution: Unicode Database

Advantages to using a Unicode database:

- Easy code migration for existing Java or PL/SQL applications
- Easy data migration from existing US7ASCII database
- Evenly distributed multilingual data
- Oracle Text Search

ORACLE

11-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Unicode Database

Easy code migration for Java or PL/SQL

A Unicode Database minimizes code changes when implementing multiple languages by storing multilingual data in existing SQL CHAR columns. It is not necessary to recode for the SQL NCHAR data type.

Easy data migration from ASCII-based data

If the current database character set and data are strict US7ASCII, the database can be migrated with a simple ALTER DATABASE statement.

Evenly distributed multilingual data

If multilingual data is distributed throughout the database, choose a Unicode database solution because it does not require you to identify which columns store multilingual data.

Oracle Text

To use multilingual documents as BLOBs and use content searching with Oracle Text, a Unicode database solution is required. You must use a Unicode database in this case. The BLOB data is converted to the database character set before being indexed by Oracle Text. If your database character set is not UTF8, data will be lost when the documents contain characters that cannot be converted to the database character set.

Choosing a Unicode Solution: Unicode Data Type

Advantages to using the Unicode data type:

- **Adding multilingual support incrementally**
- **Packaged application requires reliable Unicode data type**
- **Performance: Single byte database character set with a fixed width national character set**
- **Native support for UTF-16 with windows clients**

ORACLE

11-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Unicode Data Type

Add Multilingual Support Incrementally

To add Unicode support without migrating the database you can add SQL NCHAR data types to new and existing tables. Another option is to convert current SQL CHAR columns to SQL NCHAR columns.

Package Application

Use the SQL NCHAR data type for packaged applications because it is a reliable Unicode data type in which the data is always stored in Unicode, and the length of the data is always specified in UTF-16 code units. As a result, you need only test the application once and your application will run on customer databases of any database character set.

Performance

For performance concerns consider using a single-byte character set for the database character set and SQL NCHAR data types using AL16UTF16 for multilingual data. There is a performance overhead for using a UTF-8 encoding, which is a variable width format; fixed-width single byte and multibyte character sets perform more efficiently.

Better support for UTF-16 with windows clients

If your applications are written in Visual C/C++ or Visual Basic running on Windows, you may want to use the SQL NCHAR data types because you can store UTF-16 data in these data types in the same way you store it in the wchar_t buffer in Visual C/C++ and string buffer in Visual Basic. You can avoid buffer overflow in your client applications because the length of the wchar_t and string data types match the length of the SQL NCHAR data types in the database.

NCHAR Interoperability

- The interoperability for NCHAR is greatly improved in Oracle9i.
- Using implicit and explicit conversions, users can store, retrieve, and process NCHAR data the same way as CHAR data.
- Explicit conversions are more controllable.
- Oracle9i provides a wide range of conversion functions to meet customers requirements.

ORACLE

11-12

Copyright © Oracle Corporation, 2001. All rights reserved.

NCHAR Interoperability

The interoperability for NCHAR is greatly improved in Oracle9i. When interoperations between NCHAR and other data types are necessary, users can either apply explicit or implicit conversions. By implicit and explicit conversions, users can store, retrieve and process NCHAR data the same way as CHAR data.

The explicit conversions are more controllable. Users can decide which direction to convert. This may benefit performance in some cases where one conversion direction is faster than the other one, such as a join operation between CHAR and NCHAR columns. Oracle9i provides a wide range of conversion functions to meet customers requirements, (such as: TO_NCHAR(), TO_CHAR(), ROWIDTONCHAR(), CHARTOROWID(), TO_CLOB(), TO_NCLOB(), TO_NUMBER(), TO_DATE(), UNISTR(), ASCIIISTR())

NCHAR Interoperability (continued)

UNISTR takes as its argument a string in any character set and returns it in unicode in the database unicode character set. In the string you can specify the hexadecimal character number (not the byte representation) of difficult characters. To include UCS2 codepoint characters in the string, use the escape backslash (\) followed by the next number. To include the backslash itself, precede it with another backslash (\\).

Example:

```
SELECT UNISTR( 'M\00b2' ) FROM DUAL ;
```

ASCIISTR takes as its argument a string in any character set and returns an ASCII string in the database character set. The value returned contains only characters that appear in SQL, plus the forward slash (/).

NCHAR Interoperability

- **Implicit conversion**
 - Between NCHAR and CHAR types
 - Between NCHAR and NUMBER, DATE, ROWID, RAW, CLOBs, and so on
- **Conversion direction:**
 - INSERT/SELECT ... INTO/UPDATE/assignment operations: convert to target
 - Comparison, concatenation: CHAR → NCHAR to avoid any data loss
 - SQL function: convert to the first string parameter
- **Makes migration to NCHAR much easier**

ORACLE

NCHAR Interoperability (continued)

Oracle9i supports implicit conversions between NCHAR and CHAR data types, as well as between NCHAR and other data types such as DATE, NUMBER, ROWID, RAW, and CLOBs. The implicit conversion is incurred whenever any interoperation happens between different data types or the type of argument is different from the formal definition. Such operations include INSERT, UPDATE, and SELECT into statements, assignment in PL/SQL, comparison, concatenation in SQL or PL/SQL statements, SQL functions, and so on. At the same time, a set of rules are also defined to determine the conversion direction. For SELECT...INTO/ INSERT/ UPDATE/ assignment operations, they are converted to the target data types. For arithmetic operations between NCHAR types and other data types such as DATE, NUMBER and so on, NCHAR types are converted to the other data type. For comparison or concatenation operations between CHAR and NCHAR data types, CHAR types can be converted to NCHAR types, the reverse conversion can result in data loss if the database character set does not support the characters that are stored in the NCHAR field. The conversion direction to NCHAR can avoid data loss during character set conversion from CHAR. The conversion between NCHAR and CHAR possibly involves character set conversions. The implicit conversion makes migration to NCHAR much easier. After some NCHAR columns are added or converted from existing CHAR columns, application only needs to make minimum changes to support them. In most cases, application doesn't need to care too much about whether the data is from NCHAR column or CHAR column and whether the character data are encoded in the same character set. They will be converted when necessary.

Exception Handling for Data Loss

- **Database or session parameter:**
 - `NLS_NCHAR_CONV_EXCP` `TRUE` | `FALSE`
 - **Dynamically changed in each session**
 - **Effective for both explicit and implicit conversions**
- **Smoothness of operation versus accuracy of operation**

ORACLE

Exception Handling for Data Loss

With the Unicode data types you can store Unicode in nonUnicode database. In some cases, database character set is only a subset of NCHAR character set. Therefore, data loss could occur when NCHAR types are converted to CHAR types such as in explicit conversion function `TO_CHAR (NCHAR)` or when NCHAR data is inserted into CHAR columns. A replacement character is used when there is no corresponding mapping character in the target character set. NLS session parameter `NLS_NCHAR_CONV_EXCP` is used to control the behavior when this kind of data loss happens. Error will be returned if this parameter is set to `TRUE`. Its default value is false and data loss is not reported. This session parameter can be specified for the whole RDBMS system or the current user session. It can be dynamically changed during a session. Users can choose the default false value when they care more about the smoothness of the operation and allow some replacement characters in user data. In some cases, such replacement characters are unavoidable. It is not uncommon to see some question marks in a multilingual Web page.

Byte and Character Semantics for CHAR and VARCHAR2

- **Length specification can be byte or character:**
 - When defining columns
 - When defining parameters for SQL functions
- **Byte or character semantics is an attribute of the column, stored in data dictionary.**
- **Can be specified:**
 - At the column level
 - As a session or system default through `NLS_LENGTH_SEMANTICS`
 - As an initialization parameter
 - `NCHAR` and `NVARCHAR` are always character semantics

ORACLE

11-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Byte and Character Semantics

The default sizing of character data types (`CHAR`, `VARCHAR2`, and `LONG`) are in bytes by default. `CHAR(10)` in a table definition means 10 bytes not 10 characters. For a singlebyte character set encoding the character and byte length are the same. However, multibyte character set encoding does not correspond to the bytes, making sizing the column more difficult.

Explicitly setting `NLS_LENGTH_SEMANTICS` to `CHAR`, either as an environment variable for the client, or in an `ALTER SESSION` statement, enables the new character length semantics. Alternatively you can specify the semantic mode in the column definition, for example:

```
CREATE TABLE semantic
  (bytewise CHAR(10 BYTE), charwise CHAR(10 CHAR) ) ;
```

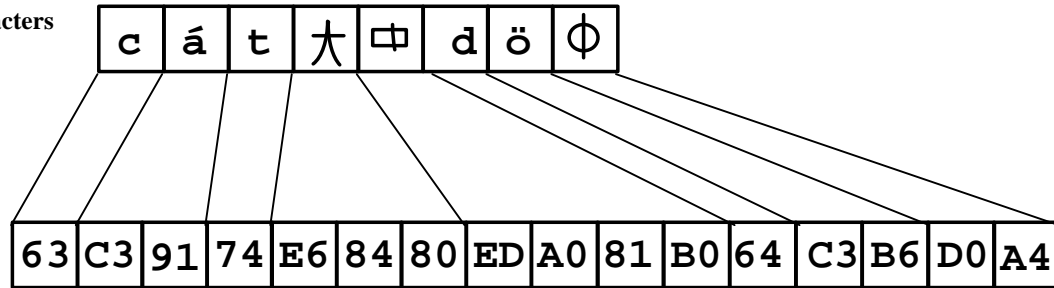
`NLS_LENGTH_SEMANTICS` has no effect on tables owned by `SYS` or `SYSTEM`. These will always be treated with byte semantics.

```
DESC semantic
```

Name	Null?	Type
-----	-----	-----
BYTEWIDE		CHAR(10)
CHARWIDE		CHAR(10 CHAR)

Character Semantics

8 visual characters



Stored in 16 UTF8 bytes

- 1 byte per ASCII Character
- 2 bytes per accented Latin, Greek, Cyrillic, Arabic, and Hebrew characters
- 3 bytes for Asian characters
- 2 byte pairs for surrogates

ORACLE

Character Semantics Support in Oracle9i

- It meets ANSI SQL standard. The size is defined in character in the standard, but most vendors implement in bytes.
- Character semantics column:
 - Implicit with `NLS_LENGTH_SEMANTICS` (session or database) setting to 'CHAR' for `VARCHAR2(30)`
 - Explicit with quantifier: `varchar2(30 char)`
 - `CHAR (size [BYTE | CHAR])`
 - `VARCHAR2 (size [BYTE | CHAR])`
- SQL functions for different length semantics in Unicode:
 - `LIKE/LIKE2/LIKE4/LIKEC`
 - `LENGTHB/LENGTH/LENGTH2/LENGTH4/LENGTHC`
 - `SUBSTRB/SUBSTR/SUBSTR2/SUBSTR4/SUBSTRC`

ORACLE

Character Semantics Support in Oracle9i

Character semantics support across server and client will make column definition close to the visual length. As the number of character does not change through character set conversion, it is easy for customers to program in Unicode.

Length Semantics

```
/* An 'a' with a unicode combining diacritic character */  
SELECT LENGTHB(UNISTR('A\0300')), LENGTH(UNISTR('A\0300')),  
LENGTH2(UNISTR('A\0300')), LENGTH4(UNISTR('A\0300')),  
LENGTHC(UNISTR('A\0300')) FROM DUAL;
```

DATA	LENGTHB	LENGTH	LENGTH2	LENGTH4	LENGTHC
á	4	2	2	2	1

```
/* A surrogate character */  
SELECT LENGTHB(UNISTR('\D801\DC01')),  
LENGTH(UNISTR('\D801\DC01')), LENGTH2(UNISTR('\D801\DC01')),  
LENGTH4(UNISTR('\D801\DC01')), LENGTHC(UNISTR('\D801\DC01'))  
FROM DUAL;
```

DATA	LENGTHB	LENGTH	LENGTH2	LENGTH4	LENGTHC
☐	4	2	2	1	1

ORACLE

Length Semantics

As discussed in the previous slides, byte and length semantics can be defined at the database, table, column, or session level. In addition, SQL functions such as `length`, `like`, and `substr` can be utilized to work with different semantics.

In this slide you are using the length SQL function to demonstrate the versatility of semantics. The first example demonstrates a base letter with a combining character. In storage they are treated as two code points but rendered as one. The length returns four, that is two bytes for each code point. Length looks at character code points based on the character set used, so it returns two. Length2 and length4 use UCS2 and UCS4 code points respectively and are mostly used for surrogates. Length uses Unicode complete characters so it returns one.

The second example demonstrates code point in the surrogate range, stored as four bytes and two Unicode characters. In length4 (UCS4) there is one surrogate character, so one is returned. Likewise length returns one complete Unicode character.

Multilingual Linguistic Sorts

- Based on the new ISO 14651 and Unicode 3.0 standard for multilingual collation
- New predefined linguistic sorts, including new Asian sorts
- Multilingual linguistic sorts:
 - GENERIC_M CANADIAN_M
 - DANISH_M FRENCH_M
 - JAPANESE_M KOREAN_M
 - SPANISH_M THAI_M
 - SCHINESE_STROKE_M SCHINESE_PINYIN_M
 - TCHINESE_RADICAL_M TCHINESE_STROKE_M
- Customizable linguistic sorts

ORACLE

11-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Multilingual Linguistic Sorts

Oracle9i extends monolingual linguistic sorts with multilingual linguistic sorts so that you can now sort additional languages as part of one sort. For multilingual data, Oracle provides a sorting mechanism based on an ISO standard (ISO14651 - International String Ordering) and the Unicode 3.0 standard. This is useful for certain regions or languages that have complex sorting rules or global multilingual databases. Multilingual linguistic sorts also work for Asian language sorts ordered by the number of strokes, PinYin, or radicals. Additionally, Oracle9i still supports all the sort orders defined by previous releases.

Example:

```
CREATE INDEX JA_INDXX ON CUSTOMERS
(NLSSORT(cust_last_NAME, 'NLS_SORT=JAPANESE_M' ) ) ;
```

Binary and Multilingual Sort

Sample Binary Sort

Ac ...
Ev ...
af ...
c ...
Ân ...
Ãd ...
Ëb ...
âx ...
åb ...

Sample Multilingual Sort

åb ...
Ac ...
Ãd ...
af ...
Ân ...
âx ...
c ...
Ëb ...
Ev ...

ORACLE

11-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Binary and Multilingual Sort

Oracle provides linguistic sort capabilities that handle the complex sorting requirements of different languages and cultures. Prior to Oracle9i, only monolingual and binary sorting were available. The following are available in Oracle9i:

JAPANESE_M: Japanese sort supports Japanese Industrial Standard (SJIS) character set order and Extended Unix Code (EUC) characters which are not included in SJIS.

KOREAN_M: Korean sort: Hangul characters are based on Unicode binary order. Hanja characters are based on pronunciation order. All Hangul characters are before Hanja characters.

SPANISH_M: Traditional Spanish sort supports special contracting characters.

THAI_M: Thai sort supports swap characters for some vowels and consonants.

SCHINESE_STROKE_M: Simplified Chinese sort uses number of strokes as primary order and radical as secondary order.

Binary and Multilingual Sort (continued)

SCHINESE_PINYIN_M: Simplified Chinese PinYin sorting order.

TCHINESE_RADICAL_M : Traditional Chinese sort based on radical as primary order and number of strokes order as secondary order.

TCHINESE_STROKE_M: Traditional Chinese sort uses number of strokes as primary order and radical as secondary order.

SCHINESE_RADICAL_M: Simplified Chinese sort based on radical as primary order and number of strokes order as secondary order.

SQL Collation Functions

- **Normalization functions:**
 - **Composition function**
 - **Decomposition function**
- **Collation Key functions: NLSSORT ()**

ORACLE

11-23

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL Collation Functions

In Oracle9i, there are three collation functions in the SQL level that users can access directly.

COMPOSE () , DECOMPOSE () and NLSSORT ()

For example:

```
SELECT COMPOSE ( 'o' || UNISTR('\0308') ) FROM DUAL;
```

CO

--

Ö

```
SELECT DECOMPOSE ( 'Châteaux' ) FROM DUAL;
```

DECOMPOSE

Cha^teaux

SQL Collation Functions (continued)

Composed Versus Decomposed

Characters which have diacritical marks, such as 'ö', can either be represented as two characters, or the base character followed by the mark. For example, as an 'o' and '¨', or as a single character. The visual output and the intended lexical meaning is the same character, but they store differently. This is due to the legacy character sets all being presentable in Unicode. This is true in all Unicode character sets. Unicode character sets only differ in which binary bytes are used; the repertoire of characters is the same. Comparisons should use the decomposed format consistently. Either strings are always store in decomposed format, or an extra conversion must be done.

According to the Unicode standard, to compose a sequence of characters, we are suppose to first canonically decompose the sequence, then canonically sort it and then look up the canonical mapping table to find the pre-composed form. For this example, the string 'o Ä ^' will be composed into '?'.

The reason we have to do so much preparation before we can begin look up is because these other sequences can also composed into '?'.

'o Ä ^', 'o ^ Ä ', 'ö Ä ', '? ^'

After canonical decomposition and canonical sort, all the strings will become 'o Ä ^'

The compose() function in Oracle9i is highly optimized. Oracle generates a mapping table at compile time. This table has all the possible decomposition sequences. It also includes the same sequence in different orders and sequences with partially composed characters. Because this table contains every possible valid decomposition sequences, Oracle does not need to first decompose the original string and then recompose to achieve Normalization Form C. It only needs to compose the original string using this mapping table. Using special mapping technology Oracle has created, this mapping table not only has a small footprint, but can also be retrieved using direct access methods, therefore eliminating any time spent on data searching.

Oracle does not have a specific SQL function to convert to Normalization Form KC, but it can be done by applying two SQL functions. First, decompose using the compatibility flag and then compose the resulting string.

Example:

```
SELECT COMPOSE(DECOMPOSE('ABC', 'COMPATIBILITY')) FROM DUAL;
```

NLSSORT Function: Example

REGULAR SORT

```
SELECT language_id,  
       translated_name  
FROM product_descriptions  
WHERE language_id = 'PTB' AND  
       SUBSTR(translated_name,1,1) IN  
       ('A', 'C', 'M')  
ORDER BY 2;
```

```
LAN TRANSLATED_NAME  
-----  
...  
PTB Monitor de Plasma 10/LE/VGA  
PTB Monitor de Plasma 10/TFT/XGA  
PTB Monitor de Plasma 10/XGA  
PTB Montagem de Unidade - A  
PTB Montagem de Unidade - A/T  
PTB Montagem de Unidade - D  
PTB Mouse +WP  
PTB Mouse +WP/CL  
PTB Mouse C/E  
PTB Mouse Pad /CL
```

NLSSORT

```
SELECT language_id,  
       translated_name  
FROM product_descriptions  
WHERE language_id = 'PTB' AND  
       SUBSTR(translated_name,1,1) IN  
       ('A', 'C', 'M')  
ORDER BY  
       NLSSORT(2,'NLS_SORT=GENERIC_M');
```

```
LAN TRANSLATED_NAME  
-----  
...  
PTB Material de FG - H  
PTB Material de SS - 3mm  
PTB Material de Plastico - Y  
PTB Material de Plastico - R  
PTB Material de FG - L  
PTB Material de SS - 1 mm  
PTB Material de Plastico - B/HD  
PTB Material de Plastico - G  
PTB Material de Plastico - O  
PTB Material de Plastico - W/HD
```

ORACLE

NLSSORT: Example

In multilingual web applications, it is very important to synchronize web content with the user's locale setting. Therefore, we offer a SQL level function that can switch to a different linguistic preference without the need to starting up another session or reset the client environment. In this example, the client linguistic setting is `Generic_M` sort. Users can dynamically switch to `Japanese_M` sort by calling `NLSSORT` function instead of changing the client environment variables.

An alternative to the sample above is:

```
ALTER SESSION SET NLS_SORT='GENERIC_M';
```

```
SELECT language_id, translated_name  
FROM product_descriptions  
WHERE language_id = 'PTB'  
AND SUBSTR(translated_name,1,1) IN ('A', 'C', 'M')
```

Unicode Support in Oracle9i APIs

Interface	Programming Env.	UTF-8	UTF16	NCHAR
JDBC	JSP and Servlet	N/A	Yes	Yes
SQL	PSP and PL/SQL	Yes	Yes	Yes
OCI	C/C++ programs	Yes	Yes	Yes
ProC/C++	C/C++ programs	Yes	Yes	Yes
ODBC	MS ASP	N/A	Yes	Yes
OLEDB	MS ASP	N/A	Yes	Yes

ORACLE

Unicode Support in Oracle9i APIs

Besides the extensive Unicode support on database server, Oracle9i also provides a comprehensive set of database access interfaces that can be deployed by applications in middle tier or client. All access programming interfaces to Oracle9i are enabled for both UTF-16 and UTF-8, providing excellent native integration for applications written in these Unicode forms. Client side interfaces are extended to have full UTF-16 support for both character semantics and NCHAR.

Programming Interfaces

- **OCI Unicode Support**
 - **Support UTF-16 bind/define buffers**
 - **Unicode meta data, `SQL_TEXT`, error messages through mode parameter**
 - **Unicode interface support is independent of server or client character set**
- **PL/SQL - Reference `VARCHAR` and `NVARCHAR` data type directly**
- **Pro*C/C++ - Unicode support through `UCHAR`, `UVARCHAR`**
- **JDBC - Transparently converts UTF-8 and `NCHAR` data to UTF-16 encoded Java Strings**
- **ODBC/OLEDB - Unicode support through `SQL_WCHAR`**

ORACLE

11-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Programming Interfaces

The Oracle Call Interface (OCI) supports Unicode at different levels. First, it supports Unicode on the data level. UTF-16 data can be retrieved or inserted through bind and define buffers which were explicitly specified as UTF-16. The second level is much wider. User data, meta data, SQL statement text and error messages can all be in UTF-16 encoding. This is achieved through a mode parameter which is specified at the time when OCI environment is initialized. This mode parameter can be dynamically switched during a session. These two levels of Unicode support are independent of the server or client character set. They can be used to access both SQL `CHAR` and SQL `NCHAR` data types, so client or middle-tier applications can run in Unicode UTF-16 environment while the server can still run in native character set. They do not necessarily migrate at the same time. Character length semantics is also supported in OCI. For the define and bind buffers, users can specify the number of character constraints in addition to the regular byte length. This guarantees that the result returned does not exceed a certain number of characters.

Other database access interfaces that support Unicode include PL/SQL, Pro*C/C++, JDBC and ODBC/OLEDB.

Summary

In this lesson, you should have learned how to:

- Differentiate usage of Unicode database and Unicode data type
- Recognize implicit and explicit interoperability between Unicode and nonUnicode data types
- Describe byte and character semantics
- Describe the multilingual sorting capabilities of an Oracle9i database

ORACLE

11-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned about the enhancements to Unicode in Oracle9i .

12

Performance and Availability Enhancements

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Take advantage of index-organized table (IOT) and index enhancements**
- **Create bitmap join indexes**
- **Identify unused indexes**
- **Use new cost-based optimizer features**
- **Know what kinds of indexes can be rebuilt online**
- **Plan for a quiescent database**
- **Resume space allocation statements**
- **Describe automatic undo management**
- **Unload metadata**

ORACLE

12-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

In Oracle9i a number of new features help to reduce planned down time, thus making the database more available.

In this lesson, you will learn about new features that help your applications perform better use memory more efficiently, and reduce down time for applications.

For more information, see *Oracle9i Application Developer's Guide - Fundamentals*, and *Oracle9i Database Performance Guide and Reference*.

Overview of Performance Enhancements

- **Enhancements to indexing:**
 - IOT enhancements
 - Bitmap join indexes
 - Skip scanning of indexes
 - Identifying unused indexes
- **SQL execution memory management**
- **Cost-based optimizer enhancements**
- **Literal replacement of bind variables**
- **Enhancements to stored outlines**

ORACLE

12-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview

Application developers have a responsibility to write SQL statements with good performance characteristics. They also are most familiar with the data access patterns exhibited by applications. DBAs set system parameters and usually have the responsibility of creating indexes. Developers and DBAs should work together to ensure that appropriate indexes are in place and system parameters are set correctly to meet the performance requirements of an application.

New index types in the Oracle9i database mean that you should reexamine existing transactions and queries with a history of performance problems to see if they could benefit from these and other new features. Likewise, indexes should be dropped if they do not provide a benefit; the Oracle9i database provides a tool that can identify indexes that are not used by your applications.

With more and more applications using cost based optimization, it is especially important to strike a balance between accuracy and efficiency when collecting optimizer statistics. Application developers can sometimes benefit from the ability to edit stored outlines of execution plans. New optimizer features address both these requirements.

Index-Organized Table Enhancements

- **Use of bitmap indexes for secondary indexes on IOTs**
- **Online CREATE, REBUILD, and COALESCE commands can be applied to secondary B*-tree indexes**
- **Online MOVE of IOT with overflow segment**
- **Parallel data manipulation language (DML) on index-organized tables**

ORACLE

12-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Index-Organized Table Enhancements

Index-organized tables (IOTs) are typically accessed by the primary key. This provides the fastest retrieval path and is the reason IOTs are used in place of heap tables. However, for retrieval that uses columns other than the primary key, secondary indexes are useful. Until Oracle9i, these indexes had to be normal B*-tree indexes. Now they can be either B*-tree or bitmap indexes. This allows you to take advantage of the benefits of a bitmap index for a secondary index on an IOT. These benefits include:

- Choice of index retrieval by the optimizer when there are few distinct values in the indexed column
- More compact index for lower cardinality columns
- Fast bitwise operations when two or more bitmap indexes are available for AND or OR operations

Another new capability introduced in Oracle9i to support secondary indexes on IOTs allows B*-tree indexes to be built, rebuilt, or coalesced online. This means that DML can continue on the underlying IOT while secondary indexes are being maintained. In previous releases, such activity would lock the table against DML.

Similarly, an IOT with an overflow segment can now be moved. In previous releases, the `ALTER TABLE . . . MOVE` command was not allowed.

Another Oracle9i enhancement is that parallel DML can now be used on IOTs. This removes a previous restriction that limited you to using serial DML on your IOTs.

Mapping Tables

- **A mapping table is needed when you use a bitmap index as a secondary index on an IOT.**
- **The mapping table contains the ROWID in the IOT associated with each bit in the bitmap index.**
- **When you access the bitmap index, the mapping table translates the bit position to the actual location of the row.**

ORACLE

12-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Mapping Tables

A bitmap index contains a series of bitmaps, each representing a potential range of row locations in the underlying table for the given key value. The bitmap can locate a row because it assumes that data blocks are contiguous within an extent. In an IOT, the blocks in an extent can be either branch blocks or leaf blocks; there is not a correspondence between the location of an entry and its contents based on the block offset from the start of the extent.

A mapping table provides the translation from a bit position in a secondary bitmap index to a location in an IOT. The bitmap table contains a column of modified ROWIDs, each pointing to a specific entry. The order of the rows in the mapping table corresponds to the bit positions in the related bitmap index.

Creating a Mapping Table

```
SQL> CREATE TABLE countries
  2  ( country_id      CHAR(2),
  3    country_name    VARCHAR2(40),
  4    currency_name    VARCHAR2(25),
  5    currency_symbol  VARCHAR2(3),
  6    region          VARCHAR2(15),
  7    CONSTRAINT      country_c_id_pk
  8      PRIMARY KEY (country_id)
  9    ORGANIZATION INDEX
 10  MAPPING TABLE TABLESPACE tbs_1;
```

ORACLE

12-6

Copyright © Oracle Corporation, 2001. All rights reserved.

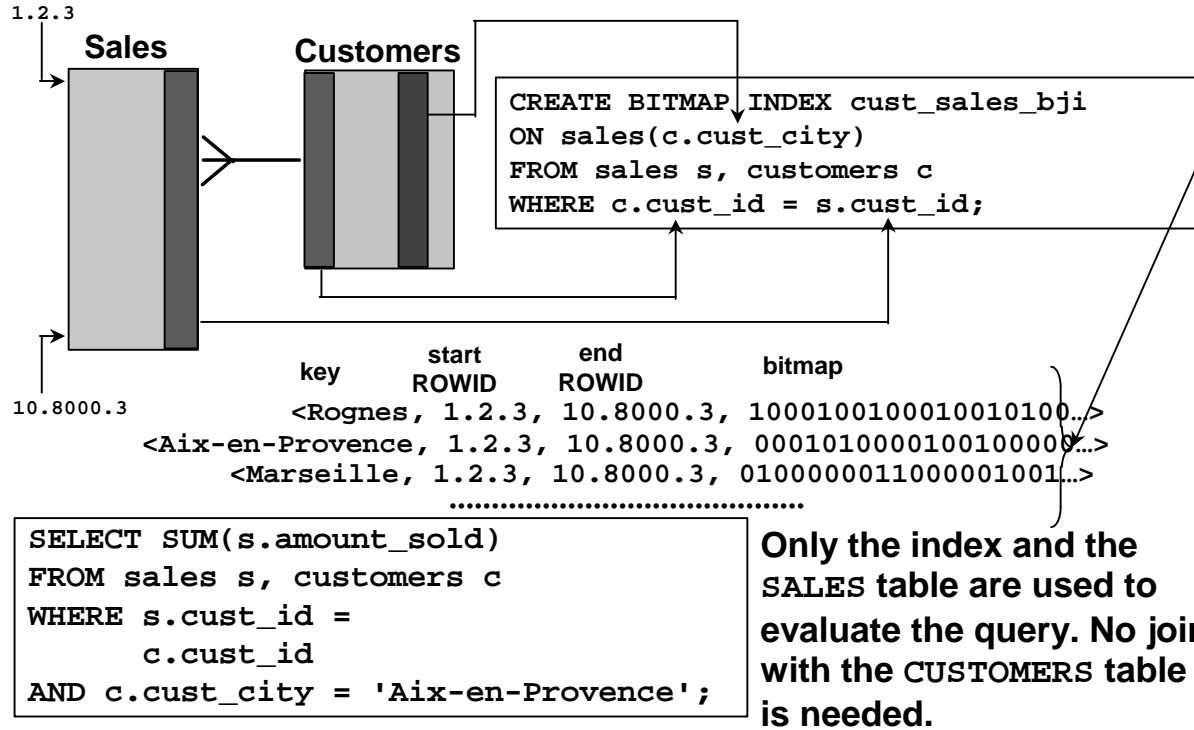
Creating a Mapping Table

If you intend to use bitmap indexes as secondary indexes on an IOT, you must create a mapping table as part of your `CREATE TABLE ... ORGANIZATION INDEX` command. You do this with the `MAPPING TABLE TABLESPACE` clause as shown in the example. The mapping table's default name is `SYS_IOT_MAP_sequence`, where *sequence* is a generated value that guarantees a unique name.

Once a mapping table is created for an IOT, each secondary bitmap index can use the same table because the structure of the bitmaps is the same in each index.

If you create an IOT with a mapping table, the mapping table is updated each time you perform DML on the IOT. The mapping table is also automatically updated with new entries when the IOT leaf blocks split or coalesce. You should therefore only create a mapping table if you expect to use it; otherwise you incur unnecessary overhead associated with maintaining the mapping table's contents.

What Is a Bitmap Join Index?



ORACLE

What Is a Bitmap Join Index?

In addition to a bitmap index on a single table, you can create a bitmap join index in Oracle9i. 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 shown in the slide, Oracle9i introduces a new CREATE BITMAP INDEX syntax, allowing you to specify a FROM and a WHERE clause.

You create a new bitmap join index named CUST_SALES_BJI on the SALES table. The key of this index is the CUST_CITY column of the CUSTOMERS table.

This example assumes that there is an enforced relationship between primary keys and foreign keys in the SALES and CUSTOMERS tables in order 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 table and also the foreign key from the SALES table to the CUSTOMERS table.

What Is a Bitmap Join Index? (continued)

The FROM and WHERE clause in the CREATE statement allow Oracle9i to make the link between the two tables. They represent the natural join condition between the two tables.

The middle part of the diagram in the 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 quite obvious as it is the same representation as for traditional bitmap indexes. Each bit in a bitmap corresponds to one row in the SALES table. If you take the first key above (Rognes), you can see that the first row in the SALES table corresponds to a product sold to a Rognes customer, while the second bit is not a product sold to a Rognes customer.

The interest of this structure becomes clear with the last part of the slide. Indeed, when the user tries to find what the total cost of all sales for the Aix-en-Provence customers is, Oracle9i can use the above 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 in this case is much faster than computing the join at query time.

Advantages and Disadvantages of Bitmap Join Indexes

- **Advantages:**
 - **Good performance for join queries and are 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**
 - **Maintenance costs are higher: Building or refreshing a bitmap join index requires a join**

ORACLE

12-9

Copyright © Oracle Corporation, 2001. All rights reserved.

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 already 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 as it compresses ROWIDs of the fact tables.

Queries using bitmap join indexes can also 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 implication of having many indexes on one table is to have higher maintenance costs, especially when the fact table is updated.

Advantages and Disadvantages (continued)

Because you must store the results of join results, bitmap join indexes have the following restrictions:

- Parallel DML is currently only supported on the fact table. Parallel DML on one of the participating dimension tables marks the index as unusable.
- Only one table can be updated concurrently by different transactions when using the bitmap join index.
- No table can appear twice in the join.
- You cannot create a bitmap join index on an index-organized table or a temporary table.
- The columns in the index must all be columns of the dimension tables.
- The dimension table join columns must be either 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.

Skip Scanning of Indexes

Index skip scanning:

- **Enables access through a composite index when the prefix column is an unknown value**
- **Supported by:**
 - **Cluster indexes**
 - **Descending scans**
 - **CONNECT BY clauses**
- **Is not supported by reverse key or bitmap indexes**

ORACLE

12-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Skip Scanning Definition

In prior releases, a composite index would be used only if the index prefix (leading) column was included in the predicate of the statement. With Oracle9i, the 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 ROWIDs 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. This can be useful when high levels of DML activity would be 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. Sometimes, these two requirements are met by two different columns in a composite index, forcing a compromise or the use of multiple indexes.

Skip Scanning Definition (continued)

During a skip scan, the B*-tree is probed for each distinct value in the prefix column. Under each prefix column value, the normal search algorithm takes over. The result is a series of searches through subsets of the index, each of which appears to result from a query using a specific value of the prefix column. However, with the skip scan, the value of the prefix column in each subset is obtained from the initial index probe rather than from the command predicate.

In addition to standard B*-tree indexes, the optimizer can use skip scans for processing:

- Cluster indexes
- Descending scans
- Statements with `CONNECT BY` clauses

Reverse key and bitmap indexes do not support the skip scan algorithm.

Skip Scanning: Example

LANGUAGE and TERRITORY combinations:

LANGUAGE	TERRITORY
ENGLISH	AMERICA
ENGLISH	CANADA
ENGLISH	UK
FRENCH	CANADA
FRENCH	FRANCE
FRENCH	SWITZERLAND
GERMAN	GERMANY
GERMAN	SWITZERLAND
PORTUGUESE	BRAZIL
PORTUGUESE	PORTUGAL

ORACLE

12-13

Copyright © Oracle Corporation, 2001. All rights reserved.

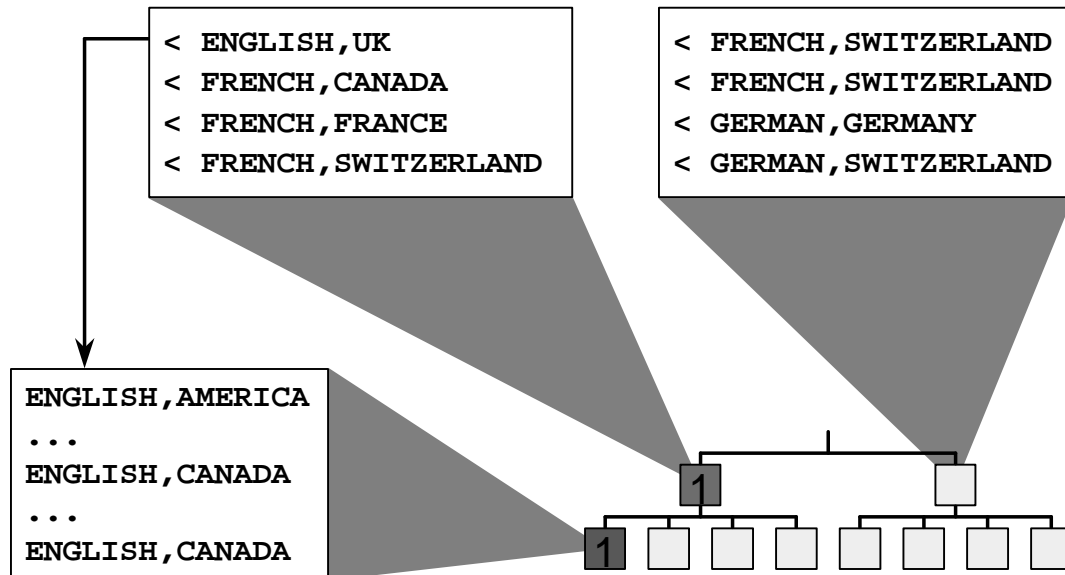
Skip Scanning: Example

In the example, suppose a composite index exists on two columns, LANGUAGE and TERRITORY, with LANGUAGE as the prefix column. The data values stored in the underlying table result in the combinations of values shown in the table. Each combination can occur multiple times in the table and the resulting index.

In previous releases without the skip scan algorithm, a query on a value in the TERRITORY column would be forced to execute a full table scan or a fast full index scan. If such a query were infrequent, this might be acceptable. If the query were more common, then you might have to add a new index on the TERRITORY column alone. This new index, however, could negatively impact the performance of inserts, updates and deletes on the table.

The skip scan solution provides an improvement without the need for the second index. While not as fast as a direct index lookup, the skip scan algorithm is faster than a full table scan in cases where the number of distinct values in the prefix column is relatively low.

Skip Scanning: Search for Switzerland



ORACLE

12-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Skip Scanning: Example

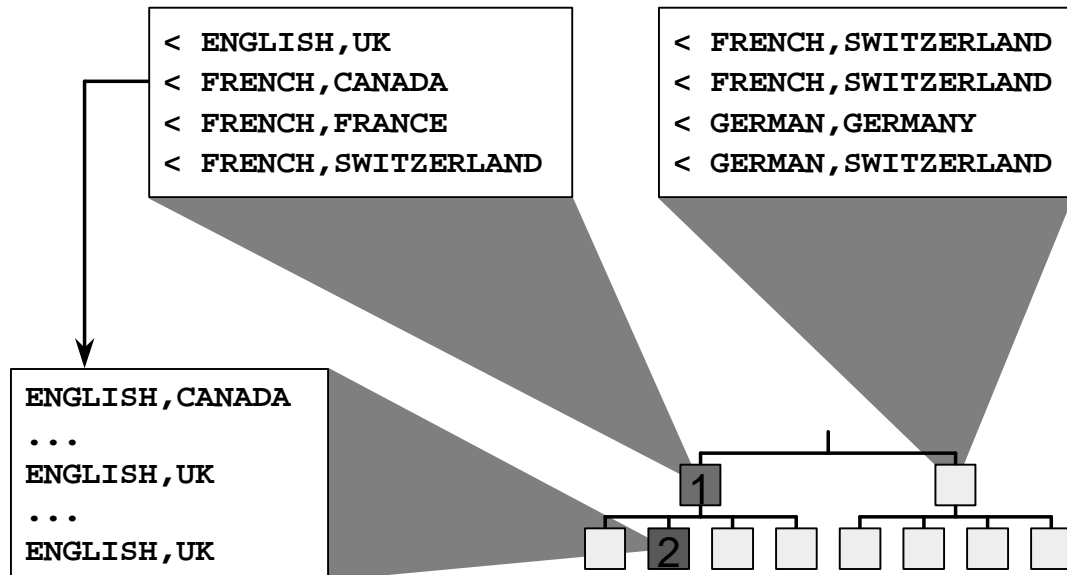
Here is part of the index built on the LANGUAGE and TERRITORY columns as shown on the previous slide. This portion of the index contains two branch blocks, each of which points to four leaf blocks. The highest value on the leaf block determines the boundary condition for the branch pointers. For example, the first leaf block referenced by the first branch block contains a number of entries but with only two distinct values, “ENGLISH,AMERICA” and “ENGLISH,CANADA,” both less than “ENGLISH,UK.” The pointer from the branch block entry to this leaf block is shown in the slide.

In the next few slides, you will see how the skip scanning algorithm bypasses unneeded leaf blocks when searching for the value “SWITZERLAND” in the TERRITORY column.

First Leaf Block

The index scan begins with the first branch block that indicates that the first leaf block has values less than “ENGLISH,UK.” The specific values on this leaf block are unknown at this time, so all of the entries are scanned. At the end of the scan of the first leaf block, the potential value “ENGLISH,SWITZERLAND” has not been reached.

Skip Scanning: Search for Switzerland



ORACLE

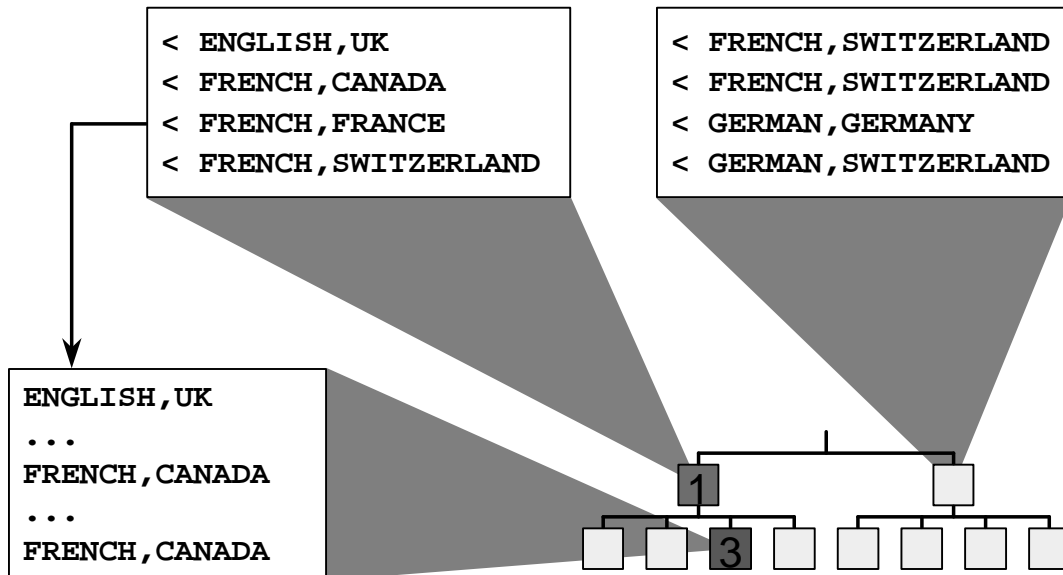
12-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Second Leaf Block

The second entry in the first branch block, combined with the last value read on the first leaf block, implies that the entries on the second leaf block are bounded by the values “ENGLISH,CANADA” and “FRENCH,CANADA.” Because there could be a number of LANGUAGE values that include “ENGLISH” and “FRENCH”, in this range, the second leaf block is also scanned for any entries with a TERRITORY value of “SWITZERLAND.”

Skip Scanning: Search for Switzerland



ORACLE

12-16

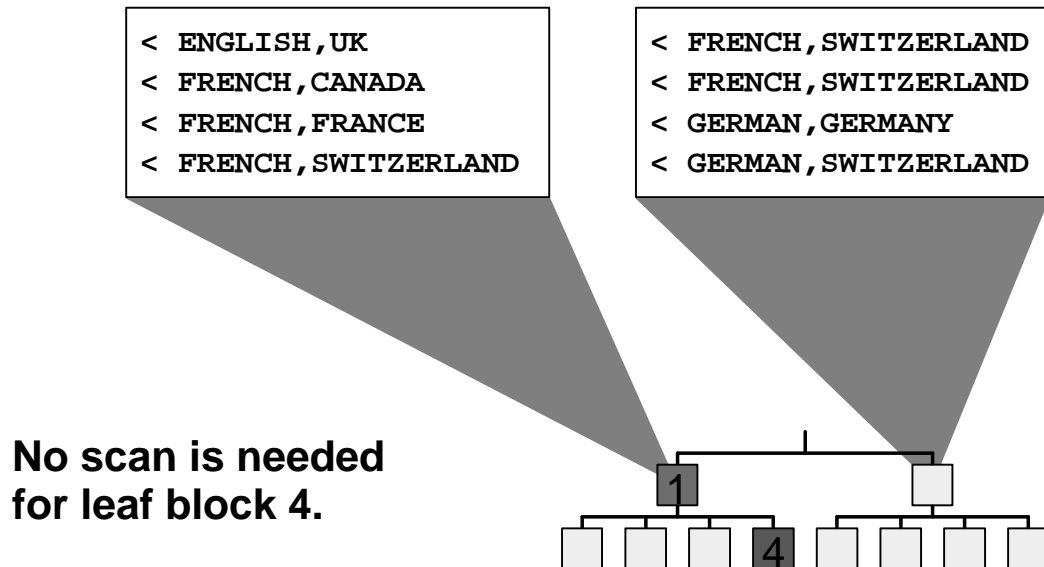
Copyright © Oracle Corporation, 2001. All rights reserved.

Third Leaf Block

The third entry in the first branch block, combined with the last value read on the second leaf block, implies that entries on the third leaf block are bounded by the values “ENGLISH,UK” and “FRENCH,FRANCE.” Although the value “ENGLISH,SWITZERLAND” would not be in this range, there is still the possibility of values such as “FRENCH, SWITZERLAND,” for example, appearing on this leaf block.

However, as the third leaf block is scanned, the value “FRENCH,CANADA” is encountered. Because the block does not contain values greater than “FRENCH,FRANCE,” the value “FRENCH,SWITZERLAND” cannot be in this leaf block. Further, there can be no other LANGUAGE values besides “FRENCH,” and so the remainder of the third leaf block can be skipped.

Skip Scanning: Search for Switzerland



ORACLE

12-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Fourth Leaf Block

The fourth entry in the first branch block indicates that the final value on the fourth leaf block must be less than “FRENCH,SWITZERLAND.” The last value on the third leaf block also has a LANGUAGE value of “FRENCH,” a fact that can be determined from the previous two branch entries. This implies that the entire block must consist of entries with a LANGUAGE of “FRENCH,” none of which has a TERRITORY value of “SWITZERLAND.” The scan therefore skips this entire block.

Identifying Unused Indexes

- **Oracle9i 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

12-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Identifying Unused Indexes

Oracle9i provides the capability to gather statistics within the database about the use of an index.

Benefits

If you find an index that is never used, you can drop the index and free the space it used.

Indexes must be maintained during most insertions, deletions, 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 Use

- To start monitoring the use of an index:

```
ALTER INDEX EMPLOYEES_idx MONITORING USAGE;
```

- To stop monitoring the usage of an index:

```
ALTER INDEX EMPLOYEES_idx NOMONITORING USAGE;
```

- **V\$OBJECT_USAGE** and **V\$OBJECT_STATS** contain information about the use of an index.

ORACLE

12-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Enabling and Disabling the Monitoring of Index Use

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 new V\$OBJECT_USAGE view supports the identification of unused indexes with the following columns:

- INDEX_NAME: The index name
- TABLE_NAME: The corresponding table
- MONITORING: Indicates whether monitoring is on or off
- USED: Indicates whether an index has been used during the monitoring time
- START_MONITORING: Time monitoring began on an index
- STOP_MONITORING: Time monitoring stopped on an index

New First Rows Optimization

- **OPTIMIZER_MODE =**
 - **FIRST_ROWS_1**
 - **FIRST_ROWS_10**
 - **FIRST_ROWS_100**
 - **FIRST_ROWS_1000**
- **ALTER SESSION**
SET OPTIMIZER_GOAL = FIRST_ROWS_n
- **/*+ FIRST_ROWS(x) */**

ORACLE

12-20

Copyright © Oracle Corporation, 2001. All rights reserved.

New First Rows Optimization

Oracle9i introduces a new way of doing first rows optimization. The old mode was partially based on heuristics, such as always using an index if possible. These heuristics could sometimes lead to bad plans.

The new approach is completely cost based and allows optimizing for a particular number of first rows, for example, the first 10 rows.

It is invoked as an argument to the `optimizer_mode` initialization parameter or the `optimizer_goal` session parameter, which allows the following values:

- **FIRST_ROWS_1**
- **FIRST_ROWS_10**
- **FIRST_ROWS_100**
- **FIRST_ROWS_1000**

In addition, the **FIRST_ROWS** hint now takes a numeric argument that is not limited to the values for the parameter. For instance, you could specify `/*+ FIRST_ROWS(20) */` as a hint.

Without a numeric argument, both the parameter and the hint imply the old first rows behavior, which is retained for backwards compatibility and plan stability.

New Statistics-Gathering Estimations

- **New auto-sampling functionality**
- **DBMS_STATS.AUTO_SAMPLE_SIZE: New possible value for ESTIMATE_PERCENT parameter; the Oracle server decides on the percentage necessary to ensure accurate statistics collection**
- **For histograms, you can now specify the following new size options in the METHOD_OPT parameter:**
 - REPEAT
 - AUTO
 - SKEWONLY

ORACLE

12-21

Copyright © Oracle Corporation, 2001. All rights reserved.

New Statistics-Gathering Estimations

Because the cost-based approach relies on statistics, you should generate statistics for all tables, clusters, and all indexes accessed by SQL statements before using the cost-based approach. If the size and data distribution of the tables change frequently, then you should regenerate these statistics regularly to ensure the statistics accurately represent the data in the tables.

Exact statistics computation requires enough space to perform scans and sorts of involved objects. If there is not enough space in memory, then temporary space might be required. Thus, it is also possible to compute only estimations in order to reduce resources needed to gather statistics. The difficulty in computing estimated statistics is to find the best sample size. Some statistics are always computed exactly, such as the number of data blocks currently containing data in a table or the depth of an index from its root block to its leaf blocks. Nevertheless, this is not true for all statistics.

With Oracle9i, it is recommended that you set the ESTIMATE_PERCENT parameter of the DBMS_STATS gathering procedures to the new value DBMS_STATS.AUTO_SAMPLE_SIZE. This is introduced to maximize performance gains while achieving necessary statistical accuracy and avoiding the extremes of collecting inaccurate statistics and wasting valuable time.

New Statistics-Gathering Estimations (continued)

Also, Oracle9i introduces new possible values for the METHOD_OPT parameters of the DBMS_STATS gathering procedures:

- If the size option is set to REPEAT and the column has a histogram with b buckets, the Oracle server attempts to create a new histogram with b buckets. If the column has no histogram, the Oracle server does not create a histogram. This option is used to maintain the same kind of statistics (histogram or no histogram) when looking at new data.
- If the size is set to AUTO, the Oracle server decides to create a histogram based on the data distribution *and* the way the column is being used by the application. This means that Oracle not only looks at nonuniformity in value repetition counts (skew) but also at nonuniformity in range (sparsity). If the application has not been run for a sufficient amount of time to capture the workload involving this column, it is better to use the SKEWONLY option temporarily.
- If the size is set to SKEWONLY, the Oracle server decides to create a histogram based solely on the data distribution (regardless of how the application uses the column). This option is useful when gathering statistics for the first time (before the workload has been captured). Using SKEWONLY can add quite a bit of overhead to statistics collection, so it is recommended that you use AUTO after the application has run for a while.

The example below shows you how to collect all table, column, and index statistics for the OE schema. The Oracle server decides what the sampling percentage should be and whether histograms are necessary (assuming that the workload has run for a while).

Note: The Oracle server captures workload information for a cursor when it is hard parsed. Information is stored in the System Global Area (SGA) and regularly flushed to disk. No access to these memory and disk structures is provided in Oracle9i.

Example

```
DBMS_STATS.GATHER_SCHEMA_STATS(  
  ownname           => 'OE',  
  estimate_percent  => DBMS_STATS.AUTO_SAMPLE_SIZE,  
  method_opt        => 'for all columns size AUTO');
```

Optimizer Cost Model Enhancements

- **The cost model now gives more meaningful cost estimates.**
- **The `PLAN_TABLE` contains three new columns:**
 - **`CPU_COST`: The estimated Central Processing Unit (CPU) cost of the operation**
 - **`IO_COST`: The estimated Input/Output (I/O) cost of the operation**
 - **`TEMP_SPACE`: The estimated temporary space, in bytes, used by the operation**
- **Includes CPU and network usage**
- **Accounts for the effect of caching**
- **Accounts for index prefetching**

ORACLE

12-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Optimizer Cost Model Enhancements

The role of a query optimizer is to produce the best performing execution plan for a given query. This process includes selecting access paths for single tables, for join orders if more than one table is involved in the query, and the implementation of each join operation (that is, join method).

Currently, a user has a choice between using the rule-based optimizer (RBO) and the cost-based optimizer (CBO). The cost-based optimizer uses a cost model (a set of cost functions) to choose between alternative access paths, join order, or join methods. The RBO uses a set of simple rules.

The cost-based optimizer compares the cost of several alternatives and selects the one with the lowest cost. In addition to the cost model, the cost-based optimizer uses a size model in order to derive statistics on intermediate tables, for example, the cardinality of the result of a join operation.

Both the cost and the size model use statistics on the objects manipulated by the query. Those statistics are produced by using the `DBMS_STATS` package, and are stored in the database dictionary.

The quality of the execution plan produced by the optimizer is dependent on the accuracy of both the cost and the size model.

The current version of both models contains several limitation both in terms of accuracy and completeness. For example, the size model assumes independence of columns when computing the selectivity of multiple predicates on different columns, and the cost model accounts only for I/O activities.

Optimizer Cost Model Enhancements (continued)

The cost model is extended to take into account the following:

- It allows users or developers to convert the cost information into more meaningful details. The `PLAN_TABLE` contains three new columns:
 - `CPU_COST`: The CPU cost of the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, this column is null. The value of this column is proportional to the number of machine cycles required for the operation.
 - `IO_COST`: The I/O cost of the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, this column is null. The value of this column is proportional to the number of data blocks read by the operation.
 - `TEMP_SPACE`: The temporary space, in bytes, used by the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, or for operations that do not use any temporary space, this column is null.
- It includes CPU and network usage. CPU usage is estimated for SQL functions and operators. Network usage is estimated when data is shipped between query servers running on different nodes of a cluster.
- It accounts for the effect of caching on the performance of nested-loops joins.
- It accounts for index prefetching. Index prefetching consists of fetching multiple leaf blocks in a single I/O operation.

Gathering System Statistics

- **System statistics enable the cost-based optimizer to use CPU and I/O characteristics.**
- **System statistics must be gathered on a regular basis by analyzing system activity for a specified period of time.**
- **Does not invalidate cached plans**
- **System analysis is provided by the new procedures:**
 - **DBMS_STATS.GATHER_SYSTEM_STATS**
 - **DBMS_STATS.SET_SYSTEM_STATS**
 - **DBMS_STATS.GET_SYSTEM_STATS**

ORACLE

12-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Gathering System Statistics

In a new feature of Oracle9i, system statistics allow the optimizer to consider a system's I/O and CPU performance and utilization. For each candidate plan, the optimizer computes estimates for I/O and CPU costs. It is important to know the system characteristics to pick the most efficient plan with optimal proportion between I/O and CPU cost.

System CPU and I/O characteristics depend on many factors and do not stay constant all the time. Using system statistics management routines, 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 allows the optimizer to generate relevant costs with respect to available system resource plans.

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

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.

Gathering System Statistics: Example

- During the first day:

```
DBMS_STATS.GATHER_SYSTEM_STATS(  
  interval => 120,  
  stattab => 'mystats',  
  statid => 'OLTP');
```

- During the first night:

```
DBMS_STATS.GATHER_SYSTEM_STATS(  
  interval => 120,  
  stattab => 'mystats',  
  statid => 'OLAP');
```

ORACLE

12-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Gathering System Statistics: Example

The example above shows database applications processing OLTP transactions during the day, and running reports at night.

First, system statistics must be collected during the day. Here, gathering ends after 120 minutes and is stored in the MYSTATS table.

Then, system statistics are collected during the night. Gathering ends after 120 minutes and is stored in the MYSTATS table.

Generally, you use the syntax above to gather the system statistics. Before invoking the GATHER_SYSTEM_STATS procedure with the INTERVAL parameter specified, be sure that the DBA has specified at least one job queue processes.

Alternatively, you can also invoke the same procedure with different arguments to enable manual gathering instead of using jobs. For syntax information refer to *Oracle9i Supplied PL/SQL Packages Reference*.

If appropriate, you can switch between the statistics gathered. Note that it is possible to automate this process by submitting a job to update the dictionary with appropriate statistics. During the day, a job can import the OLTP statistics for the daytime run, and during the night another job can import the OLAP statistics for the nighttime run.

Gathering System Statistics: Example (continued)

During subsequent days:

```
DBMS_STATS.IMPORT_SYSTEM_STATS(  
  Stattab => 'mystats', statid => 'OLTP');
```

During subsequent nights:

```
DBMS_STATS.IMPORT_SYSTEM_STATS(  
  Stattab => 'mystats', statid => 'OLAP');
```

Safe and Unsafe Cursor Sharing

- **Cursor sharing occurs when the optimizer replaces a fixed (literal) value with a bind variable before developing the execution plan for the statement.**
- **Safe cursor sharing implies that, regardless of the literal value provided in the statement, the optimizer would develop the same execution plan.**
- **Unsafe cursor sharing implies that, were it not for the substitution of the bind variable, the optimizer could develop different execution plans for different literal values.**

ORACLE

12-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Cursor Sharing

The term *cursor sharing* refers to an operation performed by the optimizer to reduce the overhead of parsing SQL statements containing fixed (literal) values. Rather than developing an execution plan each time a similar statement (but one with a different literal value) is executed, the optimizer substitutes a bind variable for the literal when the statement is first parsed. The execution plan developed using the bind variable is used for all subsequent executions of the statement, a process known as cursor sharing.

Safe Cursor Sharing

Consider a query like this one, where `EMPLOYEE_ID` is the primary key

```
SELECT * FROM employees WHERE employee_id = 153;
```

The substitution of any value would produce exactly the same execution plan. It would, therefore, be safe for the optimizer to use cursor sharing and substitute a bind variable for the value 153 before generating the execution plan for this statement.

Cursor Sharing (continued)

Unsafe Cursor Sharing

Assume the same EMPLOYEES table has a wide range of values in its DEPARTMENT_ID column. For example, department 50 could contain over one third of all employees and department 70 could contain just one or two employees. Given the two queries:

```
SELECT * FROM employees WHERE department_id = 50;
```

```
SELECT * FROM employees WHERE department_id = 70;
```

Replacing either value with a bind variable would not be a safe cursor sharing. Depending on which statement was executed first, the execution plan could contain a full table (or fast full index) scan, or it could use a simple index range scan.

Cursor Sharing

- **CURSOR_SHARING parameter values:**
 - **FORCE**
 - **EXACT (default)**
 - **SIMILAR (new in Oracle9i)**
- **CURSOR_SHARING can be changed using:**
 - **The initialization file (init.ora) parameter**
 - **An ALTER SYSTEM statement**
 - **An ALTER SESSION statement**

ORACLE

12-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Cursor Sharing

The value of the CURSOR_SHARING initialization parameter determines how the optimizer processes statements with bind variables.

- **EXACT:** Cursor sharing disabled completely
- **FORCE:** Causes cursor sharing for all literals
- **SIMILAR:** Causes cursor sharing for safe literals only

In previous releases, you could choose only the EXACT or the FORCE option. The SIMILAR option is new in Oracle9i. It causes the optimizer to examine the statement to ensure that cursor sharing occurs only for safe literal values. In doing this, it can use information about the nature of any available index (unique or nonunique) and statistics collected on the index or underlying table, including histograms.

For example, consider the distribution of values in the DEPARTMENT_ID column and the query on the EMPLOYEES table discussed earlier:

```
SELECT * FROM employees WHERE department_id = value
```

The optimizer would not perform cursor sharing with CURSOR_SHARING set to SIMILAR if you had histogram on the DEPARTMENT_ID column. This is because the histogram would indicate the skew existing for different values in this column.

The value of CURSOR_SHARING in the initialization file can be overridden with an ALTER SYSTEM or an ALTER SESSION command.

Overview of Outline Editing

- **Stored outlines were introduced in Oracle8i as a way to preserve execution plan stability across:**
 - Different releases of the database
 - Different operating environments
- **Stored outline editing is new in Oracle9i. It enables users and third-party vendors to tune execution plans without having to change the application.**
- **This is possible by editing the content of the saved plan.**

ORACLE

Outline Editing: An Overview

Stored outlines were introduced in Oracle8i as a way to preserve execution plan stability across releases of the database and across different operating environments. In Oracle9i, outline editing extends the usefulness of stored outlines by introducing a user-editing interface, which enables users and third-party vendors to tune their execution plans by editing the stored outlines used to influence the optimizer. This might be useful when using the same application in different environments (small versus big databases).

This feature benefits both application developers and customer support personnel. While the optimizer usually chooses optimal plans for queries, there are times when the user knows something about the execution environment that is inconsistent with the heuristics the optimizer follows. Sometimes an execution plan is acceptable in one environment but not in another. By editing the outline directly, the user can tune the query without having to change the application.

The application developer might generate outlines in a staging area and notice that some plan did not take advantage of an index that could improve performance. It might be easier to simply edit the outline to use the index rather than searching through the application code and tuning the SQL till it eventually yields the desired result.

For example, a customer support agent might be at a customer site investigating a problem query. By creating an outline and then editing it, it is likely that with some simple edits, such as changing join order, the problem can be solved quickly. In this way, the customer's problem is solved immediately without having to go through the process of debugging and updating the application itself.

Overview of Outline Editing

- **You clone the outline in a staging area where the outline can be safely edited without impacting the user community.**
- **Once satisfied with the result, the editor can publicize the result to the user community.**

ORACLE

12-32

Copyright © Oracle Corporation, 2001. All rights reserved.

Outline Editing: An Overview (continued)

For the customer whose environment has unique characteristics that might cause an outline to yield a less than optimal execution plan, the ability to make minor adjustments to the outline enhances the ability to support specific customer needs. In this sense, stored outlines are made more adaptive as users can make finely tuned adjustments to the saved plan.

Stored outline metadata is maintained in the OUTLN schema and maintained by the server. You are advised not to update these tables directly in the same way you are advised not to update system tables. Therefore, users need a way to safely edit an outline without compromising the integrity of the outline for the rest of the user community.

To accomplish this, it is recommended that the outline be cloned into the user's schema at the onset of the outline editing session. All subsequent editing operations are performed on that clone until the user is satisfied with his edits and chooses to publicize them. In this way, any editing done by the user does not impact the rest of the user community, which continues to use the public version of the outline until the edits are explicitly saved.

Editable Attributes

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

ORACLE

12-33

Copyright © Oracle Corporation, 2001. All rights reserved.

Editable Attributes

Join order: Join order defines the sequence in which tables are joined during query execution. This includes tables produced by evaluating subqueries and views as well as tables appearing in the FROM clauses of subqueries and views.

Join methods: Join methods define the methods used to join tables during query execution. Examples are nested loops join and sort-merge join.

Access methods: Access methods define the methods used to retrieve table data from the database. Examples are indexed access and full table scan.

Distributed execution plans: Distributed queries have execution plans that are generated for each site at which some portion of the query is executed. The execution plan for the local site at which the query is submitted can be controlled by plan stability and equivalent plans must be produced at that site. In addition, driving site selection can be controlled centrally even though it might normally change when certain schema changes are made.

Distribution methods: For parallel query execution, distribution methods define how the inputs to execution nodes are partitioned.

View and subquery merging and summary rewrite: View and subquery merging and summary rewrite is meant to include all transformations in which objects or operations that occur in one subquery of the original SQL statement are caused to migrate to a different subquery for execution. Summary rewrite can also cause one set of objects or operations to be replaced by another.

Outline Cloning

- **Public outlines:**
 - Provide a default setting when creating outlines
 - Are stored in the OUTLN schema
 - Are used when `USE_STORED_OUTLINES` is set to `TRUE`
- **Private outlines:**
 - Are stored in user's schema
 - Can be edited
 - Are used when `USE_PRIVATE_OUTLINES` is set to `TRUE`
 - Can be used to save changes as public outlines

ORACLE

12-34

Copyright © Oracle Corporation, 2001. All rights reserved.

Outline Cloning

Public Outlines

In Oracle8i, all outlines are public objects indirectly available to all users on the system for whom the `USE_STORED_OUTLINES` configuration parameter setting applies. Outline data resides in the OUTLN schema that can be thought of as an extension to the SYS schema, in the sense that it is maintained by the system only. You are discouraged from manipulating this data directly to avoid security and integrity issues associated with outline data. Outlines continue to be public by default, and only public outlines are generally available to the user community.

Private Outlines

In Oracle9i, the notion of a private outline to aid in outline editing is introduced. A private outline is an outline seen only in the current session and whose data resides in the current parsing schema. By storing the outline data for a private outline directly in the user's schema, users are given the opportunity to manipulate the outline data directly through DML in whatever way they choose. Any changes made to such an outline are not seen by any other session on the system and applying a private outline to the compilation of a statement can only be done in the current session through a new session parameter. Only when a user explicitly chooses to save edits back to the public area do the rest of the users see them.

An outline clone is a private outline that has been created by copying data from an existing outline.

Outline: Administration and Security

- **Privileges required for using CREATE OUTLINE FROM**
 - SELECT_CATALOG_ROLE
 - CREATE ANY OUTLINE
- **DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES**
 - Creates required temporary tables in user's schema for cloning and editing outlines
 - Requires EXECUTE privilege on DBMS_OUTLN_EDIT

ORACLE

12-35

Copyright © Oracle Corporation, 2001. All rights reserved.

Outline: Administration and Security

SELECT_CATALOG_ROLE

This role is required for the CREATE OUTLINE FROM command unless the issuer of the command is also the owner of the outline. Any CREATE OUTLINE statement requires the CREATE ANY OUTLINE privilege. Specification of the FROM clause additionally requires the SELECT_CATALOG_ROLE role because such a command exposes SQL text to different users who might otherwise not be privileged to read the text.

DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES

This is a supporting command procedure that creates the metadata tables in the invoker's schema. The procedure can be called by anyone with EXECUTE privilege on the DBMS_OUTLN_EDIT package. Refer to the *Supplied PL/SQL Packages Reference* for more information. Note that the DBMS_OUTLN package is synonymous with OUTLN_PKG.

Oracle developers chose to use session temporary tables to meet the requirement that private outlines be private to the session editing them. To accommodate the possibility of multiple users editing the same outline at the same time (not a good practice, but possible) the data had to be partitioned by session, which temporary tables do. This is really scratch data not intended for long-term use. Users need not worry about cleaning up when they are finished because the temporary table data is deleted at the end of their session.

Outline: Administration and Security (continued)

V\$SQL

A column is added to the V\$SQL fixed view to help users distinguish whether a shared cursor was compiled while using a private outline or a public outline. OUTLINE_SID is the name of this new column and it identifies the session ID from which the outline was retrieved. The default is 0 which implies a lookup in the OUTLN schema.

Configuration Parameters

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

```
ALTER SESSION SET USE_PRIVATE_OUTLINES =  
TRUE | FALSE | category_name ;
```

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

ORACLE

12-37

Copyright © Oracle Corporation, 2001. All rights reserved.

Configuration Parameters

The **USE_PRIVATE_OUTLINES** session parameter is added to control the use of private outlines instead of public outlines. When an outlined SQL command is issued, this parameter causes outline retrieval to come from the session private area rather than the public area normally consulted as per the setting of **USE_STORED_OUTLINES**. If no outline exists in the session private area, no outline is used for the compilation of the command.

You can specify a value for this session parameter by using the following syntax:

```
ALTER SESSION SET USE_PRIVATE_OUTLINES =  
TRUE | FALSE | category_name ;
```

Where :

- **TRUE** enables use of private outlines and defaults to the **DEFAULT** category
- **FALSE** disables use of private outlines
- *category_name* enables use of private outlines in the named category

When a user begins an outline editing session, the parameter should be set to the category to which the outline being edited belongs. This enables the feedback mechanism in that it allows the private outline to be applied to the compilation process.

Upon completion of outline editing, this parameter should be set to **FALSE** to restore the session to normal outline lookup as dictated through the **USE_STORED_OUTLINES** parameter.

Create Outline Syntax Changes

The CREATE OUTLINE command has been extended to clone outlines:

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

ORACLE

12-38

Copyright © Oracle Corporation, 2001. All rights reserved.

Create Outline Syntax Changes

The new syntax elements in the slide are:

PUBLIC: The outline is to be created for use by PUBLIC. This is the default because outline creation is intended for systemwide use.

PRIVATE: The outline is to be created for private use by the current session only and its data is stored in the current parsing schema. When specified, the prerequisite outline tables and indices must exist in the local schema.

FROM: This construct provides a way to create an outline by copying an existing one.

source_outline_name: This is the name of the outline being cloned. By default, it is found in the public area, but if preceded by the PRIVATE keyword, it is found in the local schema.

The addition of the PRIVATE and FROM keywords enable outline cloning. When you want to edit an outline, you do so on a private copy which is created by specifying the PRIVATE keyword. In the FROM clause, the source outline to be edited is named and is found in the public area unless preceded by the PRIVATE keyword, in which case the you would be copying a private version of the named outline.

When specifying the FROM clause, existing semantics apply to the outline name and category, so if unspecified, an outline name is generated under the DEFAULT category.

When a PRIVATE outline is being created, if the prerequisite outline tables to hold the outline data do not exist in the local schema, an error is returned.

Overview of Availability Enhancements

- **Online index rebuild**
- **Quiesce database**
- **Resumable statements and new system events**
- **Automatic undo management concept and undo tablespaces**
- **Metadata unload**
- **Oracle Change Data Capture**

ORACLE

12-39

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview

Maintaining the availability of a database is primarily a database administrator (DBA) responsibility. Application developers can also have responsibilities in this area because application architecture decisions can effect availability. When the database is unavailable, the application is also unavailable. Application requirements drive database availability requirements. DBAs and developers can take advantage of new features in Oracle9i to minimize application down time.

Applications

When an index can be rebuilt online, applications can continue to make database changes without waiting for the rebuild operation to complete. This feature is available since Oracle8i and has been enhanced in this release.

The ability to quiesce a database makes it possible to suspend applications for maintenance operations that previously would have required a database shutdown. This prevents loss of data due to interrupted transactions

Resumable space allocation statements make batch jobs more likely to complete successfully, minimizing the overhead of restarting the job.

With automatic undo management, your applications should encounter fewer snapshot too old errors. Also, you do not need to assign long-running batch jobs to a special large rollback segment.

Application developers can benefit from tools that allow the capture of schema definitions and data changes.

Oracle9i New Features for Application Developers 12-39

Online Index Rebuild

- **Online index rebuild has now been extended to the following:**
 - Reverse-key indexes
 - Function-based indexes
 - Key compressed indexes on regular and index-organized tables
 - Secondary indexes on index-organized tables (including statistics computation)
- **Updates are tracked in a journal table and later merged with the new index.**
- **There is no support for online build of domain (extensible) indexes or bitmap indexes.**

ORACLE

12-40

Copyright © Oracle Corporation, 2001. All rights reserved.

Online Index Rebuild

Online operations on tables and their indexes allow the table to be available for DML operations while the object redefinition is in progress. This is similar to support currently available for indexes on heap-organized tables.

Existing availability features have been extended to index-organized tables (IOTs):

- Online create and rebuild of secondary indexes on IOTs
- Online coalesce of primary indexes on IOTs
- Online update of logical ROWIDs for secondary indexes on IOTs
- Online move of IOTs along with their overflow segment
- Online update of logical ROWIDs

Because an index-organized table's primary key index cannot be directly altered, an ALTER TABLE <table_name> COALESCE operation has been introduced that coalesces the primary key index for an index-organized table. This coalesce works just like it does for indexes on heap-organized tables.

Secondary indexes on index-organized tables store logical ROWIDs to match the performance of a index on a conventional table. The logical ROWID is a *guess* as to the actual location of the row. These can become less accurate or *stale* over time. In such cases, the ALTER INDEX command can now be used to update the logical ROWIDs online. This operation is performed in parallel if the object has a default parallel clause set.

The Quiesce Database Feature

- The database can be placed in a partially available state.
- During this time, no ongoing non-DBA transactions, queries, or PL/SQL statements are allowed.
- Only users `SYS` and `SYSTEM` are considered DBA users.
- Previously, these restrictions required that a database be shut down and reopened in restricted mode.
- Now, maintenance operations can be performed without forcing a shutdown.
- This is achieved by blocking new transactions.

ORACLE

12-41

Copyright © Oracle Corporation, 2001. All rights reserved.

The Quiesce Database Feature

This feature allows a database to be put into a quiesced state. During that time, no ongoing non-DBA transactions, queries, or PL/SQL statements are allowed. Database administrators are the only users who can proceed when the system is in a quiesced state.

Without this feature a DBA has to shut down the database and reopen it in restricted mode in many cases, which is a serious restriction for 24/7 systems. The quiesce database feature lessens such a restriction by providing database administrators the ability of performing these actions safely without shutting down the database.

All non-DBA actions are blocked by the resource manager. They are allowed to proceed when the system goes back to a normal, unquiesced state. The users do not get any error messages due to the quiesced state.

Actions that cannot be safely done when the system is *not* quiesced include:

- Actions that can possibly fail if there are concurrent user transactions or queries. For example, changing the schema of a database table can fail if a concurrent transaction is accessing the same table.
- Actions whose undesirable intermediate effect can be seen by concurrent user transactions or queries. For example, to change the schema of a database table and update a PL/SQL procedure to a new version that uses this new schema of the database table. The intermediate effect of the first step is an inconsistency between the schema of the table and the implementation of the PL/SQL procedure. This inconsistency can be seen by concurrent users who try to execute the PL/SQL procedure at the same time.

Benefits of the Quiesce Database Feature

- **Users do not lose their sessions.**
- **There is no need for a client or instance cache warm-up, because a shutdown is not required.**
- **Several maintenance operations benefit from this feature:**
 - **ALTER TABLE . . .**
 - **CREATE OR REPLACE PROCEDURE . . .**
 - **Table defragmentation by means of export and import**

ORACLE

12-42

Copyright © Oracle Corporation, 2001. All rights reserved.

Benefits of the Quiesce Database Feature

Any data definition language (DDL) command requires exclusive locks and if there are ongoing transactions on the objects being modified, the DDL statement fails. Also any queries on objects that are being temporarily dropped return errors. Modifications to objects in use by other sessions also cause inconsistencies in queries and stored procedures because DDL statements implicitly commit themselves.

Defragmenting a table involves the following:

- Exporting the table
- Dropping the table
- Importing the table from the export file

After the table is dropped, applications that try to access the table will fail. Additionally, the DROP command can fail if other transactions have placed DML locks. All these operations work properly in a quiesced database.

Limitations

For the quiesce database feature to work, Resource Manager must be turned on without interruption after startup on every instance. The implication is that the ALTER SYSTEM QUIESCE RESTRICTED command is only available in those Oracle versions that have the database resource manager feature.

A recovery scheme brings the system back to a normal state in case of a crash while quiesced.

Resumable Space Allocation

- **Provides the ability to suspend, fix, and resume execution of large operations**
- **Benefits include:**
 - **Support for errors related to space limits and out-of-space conditions**
 - **Opportunity to take the corrective steps to resolve errors**
 - **Automatically continues operation when the problem is fixed**

ORACLE

12-43

Copyright © Oracle Corporation, 2001. All rights reserved.

Resumable Space Allocation

Large operations can run out of space or reach space limits after executing for a long time. When this condition occurs, the operation is rolled back and an error is returned to the user. With the resumable space allocation feature, the Oracle server suspends these large operations that run into correctable problems. Suspending the operation provides an opportunity to take the corrective steps to resolve the error condition. Once the error condition disappears, the suspended operation automatically resumes the operation's execution.

Enabling and Disabling Resumable Space Allocation Mode

- **Enable resumable space allocation:**

```
ALTER SESSION ENABLE RESUMABLE  
[TIMEOUT 60] [NAME RSA20];
```

- **Disable resumable space allocation:**

```
ALTER SESSION DISABLE RESUMABLE;
```

- **New system privilege: RESUMABLE**

ORACLE

12-44

Copyright © Oracle Corporation, 2001. All rights reserved.

Enabling and Disabling Resumable Space Allocation Mode

Enabling Mode

TIMEOUT: A suspension time-out interval is associated with every resumable operation. The time-out period is defined in seconds. A suspended operation is aborted if the error is not fixed within the specified number of seconds. The timeout can be altered using an ALTER SESSION command or the DBMS_RESUMABLE package.

NAME: The clause is a user-defined text string that is visible in the USER_RESUMABLE and DBA_RESUMABLE views.

RESUMABLE System Privilege

You must have the RESUMABLE system privilege in order to enable resumable space allocation operations with the ALTER SESSION ENABLE RESUMABLE statement. Suspending an operation automatically results in suspending the transaction. Therefore, all transactional resources are held through the suspension and resumption of an operation.

Automatic Undo Management

- **Automatic undo management simplifies and automates the management of undo data.**
- **There is no need to CREATE, DROP, or ALTER rollback segments.**
- **You can minimize snapshot too old errors more easily.**
- **You can choose to manage undo segments automatically or manually.**
- **Undo data is managed by a single tablespace defined by the UNDO_TABLESPACE initialization parameter.**

ORACLE

12-45

Copyright © Oracle Corporation, 2001. All rights reserved.

Automatic Undo Management

The automatic undo management feature simplifies and automates the management of rollback segments, to be referred to as undo segments. Using this feature, a DBA no longer is required to CREATE, DROP, or ALTER undo segments. A DBA has the choice of managing undo segments automatically or manually by choosing an undo mode. The undo mode is set by the initialization parameter UNDO_MANAGEMENT with the values of AUTO or MANUAL. If AUTO is selected, the Oracle server automatically maintains undo data within the undo tablespace. If MANUAL is selected, the DBA needs to create and manage undo segments manually, as was done in releases prior to Oracle9i.

Automatic Undo Management requires an undo tablespace. More than one undo tablespace may exist in the database, but only one undo tablespace may be active.

Rollback segments are still used for execution of transactions, but are internally created and maintained as undo segments. A DBA no longer has to decide on the different number and sizes of rollback segments to create, and developers do not have to strategically assign transactions (of different sizes) to individual rollback segments. A DBA also does not have to adjust the attributes of rollback segments in order to juggle between undo block contentions and space utilization issues.

Metadata API: Advantages

- **Centralized, easy, flexible extraction of complete database object definitions as either:**
 - XML for downstream transformation
 - SQL creation DDL
- **Transformations and filters can be applied during extraction.**
- **DBMS_METADATA has two styles of interface:**
 - **Programmatic:** OPEN / set filters / FETCH / CLOSE
 - **Casual browsing:** GET_DDL, GET_XML
- **Extensible Stylesheet Language (XSL) style sheets used for generating DDL are in a kit in the directory ORACLE_HOME/rdbms/xml/xsl.**

ORACLE

12-46

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle8i Metadata API: Issues and Concerns

There are different methods that you can use to extract metadata in an Oracle8i database.

The first method involves querying the data dictionary using SQL statements. This includes high maintenance costs, because new object definitions and DDL changes occur. In many cases more than one select statement has to be written, increasing network traffic.

The second method is to export with ROWS=N and then to import with SHOW=Y. This produces a text file from the binary dump file that can be edited to create SQL scripts. This method requires substantial editing and is not considered a convenient technique. It can also require multiple executions of EXPORT and IMPORT commands to obtain information about objects that do not fit into one of the export categories: entire database, user schema, or table.

A third method involves the OCIDescribeAny interface. This is not widely used because of its drawbacks, such as being incapable of retrieving a complete set of metadata about all database objects. It also does not scale well.

Oracle9i Metadata API: Advantages

To address the metadata unload problem, you can use a new option in an Oracle9i database, the GET_DDL function from the DBMS_METADATA. This command lets you decide what information you want to unload using a WHERE clause on the relevant data dictionary view.

The output is in Extensible Markup Language (XML) format. XML files can be edited, if necessary, after they are created using the SPOOL option of SQL*Plus. The Oracle9i product contains XSL style sheets, originally developed for internal use to generate DDL from the XML. These can be found in a kit in the ORACLE_HOME/rdbms/xml/xsl directory, and can be used or modified by any compliant parser and XSL processor.

Metadata Unload in Oracle9i

- **Casual browsing: Get the DDL for all your tables**
- **Working example of programmatic interface in:**
`$ORACLE_HOME/rdbms/demo/mddemo.sql`
- **Documented in *Oracle9i Supplied PL/SQL Packages Reference***

```
SELECT
  dbms_metadata.get_ddl('TABLE', table_name)
FROM user_tables
WHERE nested = 'NO';
```

ORACLE

12-47

Copyright © Oracle Corporation, 2001. All rights reserved.

Metadata Unload in Oracle9i

The `SELECT` statement in the slide provides, in combination with a `SQL*Plus SPOOL` command, an editable output of the required objects. You can make the query as complex as you need to obtain the DDL you require by including an appropriate `WHERE` clause.

Summary

In this lesson, you should have learned how to:

- **Take advantage of IOT and index enhancements**
- **Create bitmap join indexes**
- **Identify unused indexes**
- **Use new cost-based optimizer features**
- **Rebuild new kinds of indexes online**
- **Quiesce a database**
- **Resume space allocation statements**
- **Describe automatic undo management**
- **Unload metadata**

ORACLE

12-48

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned about new features that help your applications perform better while using indexes and memory more efficiently.

Practice 12 Overview

This practice covers identifying unused indexes.

ORACLE

12-49

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 12 Overview

In this practice you examine an execution plan for a query that skip scans an index. Also, you identify indexes that may be unused. These unused indexes are candidates to be dropped.

To perform this practice, go to Appendix A, “Practices.”

