

# **Oracle Database 10g: SQL Fundamentals II**

**Student Guide • Volume 1**

D17111GC11

Edition 1.1

August 2004

Applied

**ORACLE.**

# Contents

## Preface

### I Introduction

Objectives 1-2

Course Objectives 1-3

Course Overview 1-4

Summary 1-6

### 1 Controlling User Access

Objectives 1-2

Controlling User Access 1-3

Privileges 1-4

System Privileges 1-5

Creating Users 1-6

User System Privileges 1-7

Granting System Privileges 1-8

What Is a Role? 1-9

Creating and Granting Privileges to a Role 1-10

Changing Your Password 1-11

Object Privileges 1-12

Granting Object Privileges 1-14

Passing On Your Privileges 1-15

Confirming Privileges Granted 1-16

Revoking Object Privileges 1-17

Summary 1-19

Practice 1: Overview 1-20

### 2 Manage Schema Objects

Objectives 2-2

The ALTER TABLE Statement 2-3

Adding a Column 2-5

Modifying a Column 2-6

Dropping a Column 2-7

The SET UNUSED Option 2-8

Adding a Constraint Syntax 2-10

Adding a Constraint 2-11

ON DELETE CASCADE 2-12

Deferring Constraints 2-13

Dropping a Constraint 2-14

Disabling Constraints 2-15

Enabling Constraints 2-16

Cascading Constraints 2-18

Overview of Indexes 2-20

- CREATE INDEX with CREATE TABLE Statement 2-21
- Function-Based Indexes 2-23
- Removing an Index 2-25
- DROP TABLE ... PURGE 2-26
- The FLASHBACK TABLE Statement 2-27
- External Tables 2-29
  - Creating a Directory for the External Table 2-31
  - Creating an External Table 2-33
  - Creating an External Table Using ORACLE\_LOADER 2-35
  - Querying External Tables 2-37
- Summary 2-38
- Practice 2: Overview 2-39

### **3 Manipulating Large Data Sets**

- Objectives 3-2
- Using Subqueries to Manipulate Data 3-3
- Copying Rows from Another Table 3-4
- Inserting Using a Subquery as a Target 3-5
- Retrieving Data with a Subquery as Source 3-7
- Updating Two Columns with a Subquery 3-8
- Updating Rows Based on Another Table 3-9
- Deleting Rows Based on Another Table 3-10
- Using the WITH CHECK OPTION Keyword on DML Statements 3-11
- Overview of the Explicit Default Feature 3-12
- Using Explicit Default Values 3-13
- Overview of Multitable INSERT Statements 3-14
- Types of Multitable INSERT Statements 3-16
- Multitable INSERT Statements 3-17
- Unconditional INSERT ALL 3-19
- Conditional INSERT ALL 3-20
- Conditional INSERT FIRST 3-22
- Pivoting INSERT 3-24
- The MERGE Statement 3-27
- The MERGE Statement Syntax 3-28
- Merging Rows 3-29
- Tracking Changes in Data 3-31
- Example of the Flashback Version Query 3-32
- The VERSIONS BETWEEN Clause 3-34
- Summary 3-35
- Practice 3: Overview 3-36

### **4 Generating Reports by Grouping Related Data**

- Objectives 4-2
- Review of Group Functions 4-3

Review of the GROUP BY Clause 4-4  
Review of the HAVING Clause 4-5  
GROUP BY with ROLLUP and CUBE Operators 4-6  
ROLLUP Operator 4-7  
ROLLUP Operator: Example CUBE Operator 4-9  
CUBE Operator: Example 4-10  
GROUPING Function 4-11  
GROUPING Function: Example 4-12  
GROUPING SETS 4-13  
GROUPING SETS: Example 4-15  
Composite Columns 4-17  
Composite Columns: Example 4-19  
Concatenated Groupings 4-21  
Concatenated Groupings: Example 4-22  
Summary 4-23  
Practice 4: Overview 4-24

## **5 Managing Data in Different Time Zones**

Objectives 5-2  
Time Zones 5-3  
TIME\_ZONE Session Parameter 5-4  
CURRENT\_DATE, CURRENT\_TIMESTAMP, and LOCALTIMESTAMP 5-5  
CURRENT\_DATE 5-6  
CURRENT\_TIMESTAMP 5-7  
LOCALTIMESTAMP 5-8  
DBTIMEZONE and SESSIONTIMEZONE 5-9  
TIMESTAMP Data Type 5-10  
TIMESTAMP Data Types 5-11  
TIMESTAMP Fields 5-12  
Difference between DATE and TIMESTAMP 5-13  
TIMESTAMP WITH TIMEZONE Data Type 5-14  
TIMESTAMP WITH TIMEZONE: Example 5-15  
TIMESTAMP WITH LOCAL TIMEZONE 5-16  
TIMESTAMP WITH LOCAL TIMEZONE: Example 5-17  
INTERVAL Data Types 5-18  
INTERVAL Fields 5-20  
INTERVAL YEAR TO MONTH Data Type 5-21  
INTERVAL YEAR TO MONTH: Example 5-22  
INTERVAL DAY TO SECOND Data Type 5-23  
INTERVAL DAY TO SECOND Data Type: Example 5-24  
EXTRACT 5-25

TZ\_OFFSET 5-26  
TIMESTAMP Conversion Using FROM\_TZ 5-28  
Converting to TIMESTAMP Using TO\_TIMESTAMP and TO\_TIMESTAMP\_TZ 5-29  
Time Interval Conversion with TO\_YMINTERVAL 5-30  
Using TO\_DSINTERVAL: Example 5-31  
Daylight Saving Time 5-32  
Summary 5-34  
Practice 5: Overview 5-35

## **6 Retrieving Data Using Subqueries**

Objectives 6-2  
Multiple-Column Subqueries 6-3  
Column Comparisons 6-4  
Pairwise Comparison Subquery 6-5  
Nonpairwise Comparison Subquery 6-6  
Scalar Subquery Expressions 6-7  
Scalar Subqueries: Examples 6-8  
Correlated Subqueries 6-10  
Using Correlated Subqueries 6-12  
Using the EXISTS Operator 6-14  
Find Employees Who Have at Least One Person Reporting to Them 6-15  
Find All Departments That Do Not Have Any Employees 6-16  
Correlated UPDATE 6-17  
Using Correlated UPDATE 6-18  
Correlated DELETE 6-20  
Using Correlated DELETE 6-21  
The WITH Clause 6-22  
WITH Clause: Example 6-23  
Summary 6-25  
Practice 6: Overview 6-27

## **7 Hierarchical Retrieval**

Objectives 7-2  
Sample Data from the EMPLOYEES Table 7-3  
Natural Tree Structure 7-4  
Hierarchical Queries 7-5  
Walking the Tree 7-6  
Walking the Tree: From the Bottom Up 7-8  
Walking the Tree: From the Top Down 7-9  
Ranking Rows with the LEVEL Pseudocolumn 7-10  
Formatting Hierarchical Reports Using LEVEL and LPAD 7-11  
Pruning Branches 7-13  
Summary 7-14  
Practice 7: Overview 7-15

## **8 Regular Expression Support**

Objectives 8-2

Regular Expression Overview 8-3

Meta Characters 8-4

Using Meta Characters 8-5

Regular Expression Functions 8-7

The `REGEXP` Function Syntax 8-8

Performing Basic Searches 8-9

Checking the Presence of a Pattern 8-10

Example of Extracting Substrings 8-11

Replacing Patterns 8-12

Regular Expressions and Check Constraints 8-13

Summary 8-14

Practice 8: Overview 8-15

# I Introduction

ORACLE<sup>®</sup>

Copyright © 2004, Oracle. All rights reserved.

# Objectives

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

- **List the course objectives**
- **Describe the sample tables used in the course**

**ORACLE®**



# Course Objectives

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

- **Use advanced SQL data retrieval techniques to retrieve data from database tables**
- **Apply advanced techniques in a practice that simulates real life**

**ORACLE®**

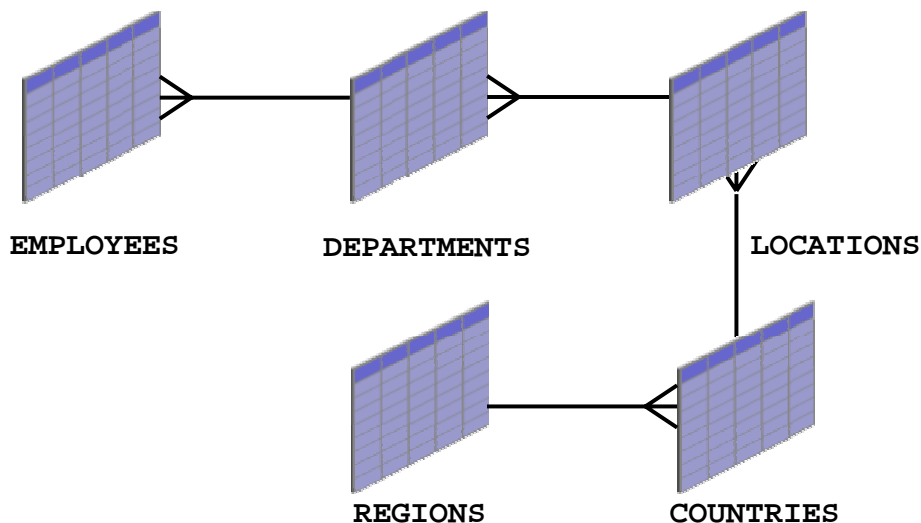
# Course Overview

**In this course, you will use advanced SQL data retrieval techniques such as:**

- **Datetime functions**
- **ROLLUP, CUBE operators, and GROUPING SETS**
- **Hierarchical queries**
- **Correlated subqueries**
- **Multitable inserts**
- **Merge operation**
- **External tables**
- **Regular expression usage**

**ORACLE®**

# Course Application



ORACLE

I-5

Copyright © 2004, Oracle. All rights reserved.

## Tables Used in the Course

The following tables are used in this course:

**EMPLOYEES:** The EMPLOYEES table contains information about all the employees such as their first and last names, job IDs, salaries, hire dates, department IDs, and manager IDs. This table is a child of the DEPARTMENTS table.

**DEPARTMENTS:** The DEPARTMENTS table contains information such as the department ID, department name, manager ID, and location ID. This table is the primary key table to the EMPLOYEES table.

**LOCATIONS:** This table contains department location information. It contains location ID, street address, city, state province, postal code, and country ID information. It is the primary key table to DEPARTMENTS table and is a child of the COUNTRIES table.

**COUNTRIES:** This table contains the country names, country IDs, and region IDs. It is a child of the REGIONS table. This table is the primary key table to the LOCATIONS table.

**REGIONS:** This table contains region IDs and region names of the various countries. It is a primary key table to the COUNTRIES table.

# Summary

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

- **The course objectives**
- **The sample tables used in the course**

**ORACLE®**

# 1

## Controlling User Access

ORACLE®

Copyright © 2004, Oracle. All rights reserved.

# Objectives

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

- **Differentiate system privileges from object privileges**
- **Grant privileges on tables**
- **View privileges in the data dictionary**
- **Grant roles**
- **Distinguish between privileges and roles**

**ORACLE**

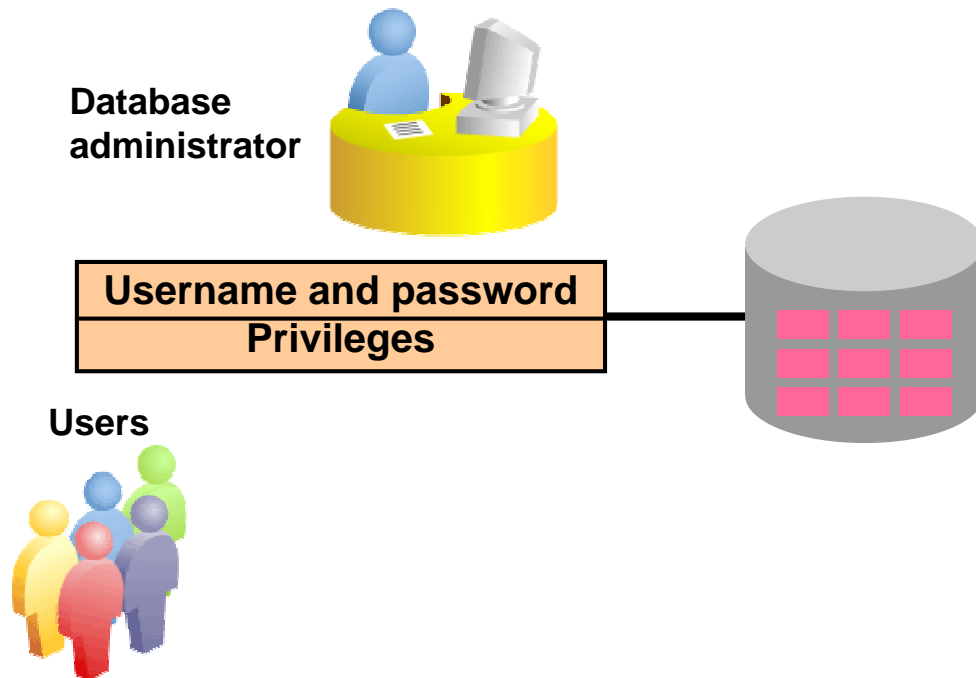
1-2

Copyright © 2004, Oracle. All rights reserved.

## Objectives

In this lesson, you learn how to control database access to specific objects and add new users with different levels of access privileges.

# Controlling User Access



ORACLE®

1-3

Copyright © 2004, Oracle. All rights reserved.

## Controlling User Access

In a multiple-user environment, you want to maintain security of the database access and use. With Oracle server database security, you can do the following:

- Control database access
- Give access to specific objects in the database
- Confirm given and received privileges with the Oracle data dictionary
- Create synonyms for database objects

Database security can be classified into two categories: system security and data security. System security covers access and use of the database at the system level such as the username and password, the disk space allocated to users, and the system operations that users can perform. Database security covers access and use of the database objects and the actions that those users can have on the objects.

# Privileges

- **Database security:**
  - System security
  - Data security
- **System privileges: Gaining access to the database**
- **Object privileges: Manipulating the content of the database objects**
- **Schemas: Collection of objects such as tables, views, and sequences**

ORACLE®

1-4

Copyright © 2004, Oracle. All rights reserved.

## Privileges

Privileges are the right to execute particular SQL statements. The database administrator (DBA) is a high-level user with the ability to create users and grant users access to the database and its objects. Users require *system privileges* to gain access to the database and *object privileges* to manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users or to *roles*, which are named groups of related privileges.

### Schemas

A *schema* is a collection of objects such as tables, views, and sequences. The schema is owned by a database user and has the same name as that user.

For more information, see the *Oracle Database 10g Application Developer's Guide – Fundamentals* reference manual.



# System Privileges

- **More than 100 privileges are available.**
- **The database administrator has high-level system privileges for tasks such as:**
  - **Creating new users**
  - **Removing users**
  - **Removing tables**
  - **Backing up tables**

ORACLE

1-5

Copyright © 2004, Oracle. All rights reserved.

## System Privileges

More than 100 distinct system privileges are available for users and roles. System privileges typically are provided by the database administrator.

### Typical DBA Privileges

System Privilege	Operations Authorized
CREATE USER	Grantee can create other Oracle users.
DROP USER	Grantee can drop another user.
DROP ANY TABLE	Grantee can drop a table in any schema.
BACKUP ANY TABLE	Grantee can back up any table in any schema with the export utility.
SELECT ANY TABLE	Grantee can query tables, views, or materialized views in any schema.
CREATE ANY TABLE	Grantee can create tables in any schema.

# Creating Users

The DBA creates users with the `CREATE USER` statement.

```
CREATE USER user
IDENTIFIED BY password;
```

```
CREATE USER HR
IDENTIFIED BY HR;
User created.
```

ORACLE

1-6

Copyright © 2004, Oracle. All rights reserved.

## Creating a User

The DBA creates the user by executing the `CREATE USER` statement. The user does not have any privileges at this point. The DBA can then grant privileges to that user. These privileges determine what the user can do at the database level.

The slide gives the abridged syntax for creating a user.

In the syntax:

*user* is the name of the user to be created

*Password* specifies that the user must log in with this password

For more information, see *Oracle Database 10g SQL Reference*, “GRANT” and “CREATE USER.”

# User System Privileges

- After a user is created, the DBA can grant specific system privileges to that user.

```
GRANT privilege [, privilege...]  
TO user [, user/ role, PUBLIC...];
```

- An application developer, for example, may have the following system privileges:
  - CREATE SESSION
  - CREATE TABLE
  - CREATE SEQUENCE
  - CREATE VIEW
  - CREATE PROCEDURE

ORACLE®

1-7

Copyright © 2004, Oracle. All rights reserved.

## Typical User Privileges

After the DBA creates a user, the DBA can assign privileges to that user.

System Privilege	Operations Authorized
CREATE SESSION	Connect to the database
CREATE TABLE	Create tables in the user's schema
CREATE SEQUENCE	Create a sequence in the user's schema
CREATE VIEW	Create a view in the user's schema
CREATE PROCEDURE	Create a stored procedure, function, or package in the user's schema

In the syntax:

*privilege* is the system privilege to be granted  
*user* | *role* | *PUBLIC* is the name of the user, the name of the role, or *PUBLIC*  
designates that every user is granted the privilege

**Note:** Current system privileges can be found in the *SESSION\_PRIVS* dictionary view.

# Granting System Privileges

**The DBA can grant specific system privileges to a user.**

```
GRANT  create session, create table,  
       create sequence, create view  
TO      scott;  
Grant succeeded.
```

ORACLE

1-8

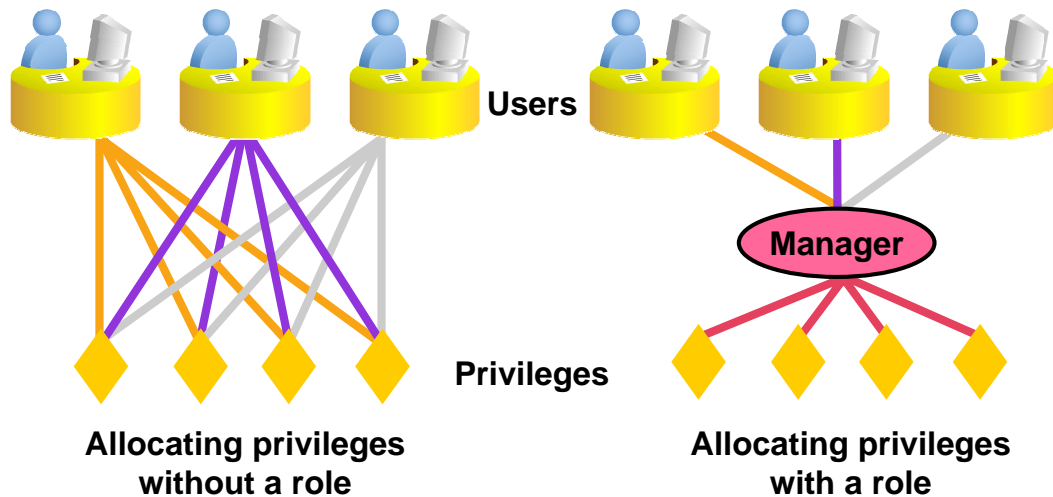
Copyright © 2004, Oracle. All rights reserved.

## Granting System Privileges

The DBA uses the GRANT statement to allocate system privileges to the user. After the user has been granted the privileges, the user can immediately use those privileges.

In the example on the slide, user Scott has been assigned the privileges to create sessions, tables, sequences, and views.

# What Is a Role?



ORACLE

1-9

Copyright © 2004, Oracle. All rights reserved.

## What Is a Role?

A role is a named group of related privileges that can be granted to the user. This method makes it easier to revoke and maintain privileges.

A user can have access to several roles, and several users can be assigned the same role. Roles are typically created for a database application.

## Creating and Assigning a Role

First, the DBA must create the role. Then the DBA can assign privileges to the role and assign the role to users.

### Syntax

```
CREATE    ROLE role;
```

In the syntax:

*role* is the name of the role to be created

After the role is created, the DBA can use the GRANT statement to assign the role to users as well as assign privileges to the role.

# Creating and Granting Privileges to a Role

- **Create a role**

```
CREATE ROLE manager;  
Role created.
```

- **Grant privileges to a role**

```
GRANT create table, create view  
TO manager;  
Grant succeeded.
```

- **Grant a role to users**

```
GRANT manager TO DE HAAN, KOCHHAR;  
Grant succeeded.
```

ORACLE

1-10

Copyright © 2004, Oracle. All rights reserved.

## Creating a Role

The example on the slide creates a manager role and then enables managers to create tables and views. It then grants De Haan and Kochhar the role of managers. Now De Haan and Kochhar can create tables and views.

If users have multiple roles granted to them, they receive all of the privileges associated with all of the roles.

# Changing Your Password

- The DBA creates your user account and initializes your password.
- You can change your password by using the `ALTER USER` statement.

```
ALTER USER HR  
IDENTIFIED BY employ;  
User altered.
```

ORACLE

1-11

Copyright © 2004, Oracle. All rights reserved.

## Changing Your Password

The DBA creates an account and initializes a password for every user. You can change your password by using the `ALTER USER` statement.

### Syntax

```
ALTER USER user IDENTIFIED BY password;
```

In the syntax:

<i>user</i>	is the name of the user
<i>password</i>	specifies the new password

Although this statement can be used to change your password, there are many other options. You must have the `ALTER USER` privilege to change any other option.

For more information, see the *Oracle Database 10g SQL Reference* manual.

**Note:** SQL\*Plus has a `PASSWORD` command (`PASSW`) that can be used to change the password of a user when the user is logged in. This command is not available in *iSQL\*Plus*.

# Object Privileges

Object Privilege	Table	View	Sequence	Procedure
ALTER	√		√	
DELETE	√	√		
EXECUTE				√
INDEX	√			
INSERT	√	√		
REFERENCES	√			
SELECT	√	√	√	
UPDATE	√	√		

ORACLE

1-12

Copyright © 2004, Oracle. All rights reserved.

## Object Privileges

An *object privilege* is a privilege or right to perform a particular action on a specific table, view, sequence, or procedure. Each object has a particular set of grantable privileges. The table on the slide lists the privileges for various objects. Note that the only privileges that apply to a sequence are SELECT and ALTER. UPDATE, REFERENCES, and INSERT can be restricted by specifying a subset of updatable columns. A SELECT privilege can be restricted by creating a view with a subset of columns and granting the SELECT privilege only on the view. A privilege granted on a synonym is converted to a privilege on the base table referenced by the synonym.



# Object Privileges

- **Object privileges vary from object to object.**
- **An owner has all the privileges on the object.**
- **An owner can give specific privileges on that owner's object.**

```
GRANT      object_priv [(columns)]  
ON         object  
TO         {user|role|PUBLIC}  
[WITH GRANT OPTION];
```

ORACLE

1-13

Copyright © 2004, Oracle. All rights reserved.

## Granting Object Privileges

Different object privileges are available for different types of schema objects. A user automatically has all object privileges for schema objects contained in the user's schema. A user can grant any object privilege on any schema object that the user owns to any other user or role. If the grant includes `WITH GRANT OPTION`, then the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

In the syntax:

<i>object_priv</i>	is an object privilege to be granted
ALL	specifies all object privileges
<i>columns</i>	specifies the column from a table or view on which privileges are granted
ON <i>object</i>	is the object on which the privileges are granted
TO	identifies to whom the privilege is granted
PUBLIC	grants object privileges to all users
WITH GRANT OPTION	enables the grantee to grant the object privileges to other users and roles

# Granting Object Privileges

- Grant query privileges on the **EMPLOYEES** table.

```
GRANT  select
ON     employees
TO     sue, rich;
Grant succeeded.
```

- Grant privileges to update specific columns to users and roles.

```
GRANT  update (department_name, location_id)
ON     departments
TO     scott, manager;
Grant succeeded.
```

ORACLE

1-14

Copyright © 2004, Oracle. All rights reserved.

## Guidelines

- To grant privileges on an object, the object must be in your own schema, or you must have been granted the object privileges WITH GRANT OPTION.
- An object owner can grant any object privilege on the object to any other user or role of the database.
- The owner of an object automatically acquires all object privileges on that object.

The first example on the slide grants users Sue and Rich the privilege to query your **EMPLOYEES** table. The second example grants **UPDATE** privileges on specific columns in the **DEPARTMENTS** table to Scott and to the manager role.

If Sue or Rich now want to use a **SELECT** statement to obtain data from the **EMPLOYEES** table, the syntax they must use is:

```
SELECT * FROM HR.employees;
```

Alternatively, they can create a synonym for the table and issue a **SELECT** statement from the synonym:

```
CREATE SYNONYM emp FOR HR.employees;
SELECT * FROM emp;
```

**Note:** DBAs generally allocate system privileges; any user who owns an object can grant object privileges.

# Passing On Your Privileges

- Give a user authority to pass along privileges.

```
GRANT  select, insert
ON      departments
TO      scott
WITH    GRANT OPTION;
Grant succeeded.
```

- Allow all users on the system to query data from Alice's DEPARTMENTS table.

```
GRANT  select
ON      alice.departments
TO      PUBLIC;
Grant succeeded.
```

ORACLE

1-15

Copyright © 2004, Oracle. All rights reserved.

## WITH GRANT OPTION Keyword

A privilege that is granted with the `WITH GRANT OPTION` clause can be passed on to other users and roles by the grantee. Object privileges granted with the `WITH GRANT OPTION` clause are revoked when the grantor's privilege is revoked.

The example on the slide gives user Scott access to your `DEPARTMENTS` table with the privileges to query the table and add rows to the table. The example also shows that Scott can give others these privileges.

## PUBLIC Keyword

An owner of a table can grant access to all users by using the `PUBLIC` keyword.

The second example allows all users on the system to query data from Alice's `DEPARTMENTS` table.

## Confirming Privileges Granted

Data Dictionary View	Description
<code>ROLE_SYS_PRIVS</code>	System privileges granted to roles
<code>ROLE_TAB_PRIVS</code>	Table privileges granted to roles
<code>USER_ROLE_PRIVS</code>	Roles accessible by the user
<code>USER_TAB_PRIVS_MADE</code>	Object privileges granted on the user's objects
<code>USER_TAB_PRIVS_RECD</code>	Object privileges granted to the user
<code>USER_COL_PRIVS_MADE</code>	Object privileges granted on the columns of the user's objects
<code>USER_COL_PRIVS_RECD</code>	Object privileges granted to the user on specific columns
<code>USER_SYS_PRIVS</code>	System privileges granted to the user

ORACLE

1-16

Copyright © 2004, Oracle. All rights reserved.

### Confirming Granted Privileges

If you attempt to perform an unauthorized operation, such as deleting a row from a table for which you do not have the `DELETE` privilege, the Oracle server does not permit the operation to take place.

If you receive the Oracle server error message “table or view does not exist,” then you have done either of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privilege

You can access the data dictionary to view the privileges that you have. The chart on the slide describes various data dictionary views.

## Revoking Object Privileges

- You use the **REVOKE** statement to revoke privileges granted to other users.
- Privileges granted to others through the **WITH GRANT OPTION** clause are also revoked.

```
REVOKE {privilege [, privilege...]|ALL}  
ON      object  
FROM    {user[, user...]|role|PUBLIC}  
[CASCADE CONSTRAINTS];
```

ORACLE

1-17

Copyright © 2004, Oracle. All rights reserved.

### Revoking Object Privileges

You can remove privileges granted to other users by using the **REVOKE** statement. When you use the **REVOKE** statement, the privileges that you specify are revoked from the users you name and from any other users to whom those privileges were granted by the revoked user.

In the syntax:

**CASCADE** is required to remove any referential integrity constraints made to the **CONSTRAINTS** object by means of the **REFERENCES** privilege

For more information, see *Oracle Database 10g SQL Reference*.

**Note:** If a user were to leave the company and you revoke his privileges, you must re-grant any privileges that this user may have granted to other users. If you drop the user account without revoking privileges from it, then the system privileges granted by this user to other users are not affected by this action.

# Revoking Object Privileges

**As user Alice, revoke the `SELECT` and `INSERT` privileges given to user Scott on the `DEPARTMENTS` table.**

```
REVOKE select, insert
ON      departments
FROM    scott;
Revoke succeeded.
```

ORACLE®

1-18

Copyright © 2004, Oracle. All rights reserved.

## Revoking Object Privileges (continued)

The example on the slide revokes `SELECT` and `INSERT` privileges given to user Scott on the `DEPARTMENTS` table.

**Note:** If a user is granted a privilege with the `WITH GRANT OPTION` clause, that user can also grant the privilege with the `WITH GRANT OPTION` clause, so that a long chain of grantees is possible, but no circular grants (granting to a grant ancestor) are permitted. If the owner revokes a privilege from a user who granted the privilege to other users, then the revoking cascades to all privileges granted.

For example, if user A grants a `SELECT` privilege on a table to user B including the `WITH GRANT OPTION` clause, user B can grant to user C the `SELECT` privilege with the `WITH GRANT OPTION` clause as well, and user C can then grant to user D the `SELECT` privilege. If user A revokes privileges from user B, then the privileges granted to users C and D are also revoked.

# Summary

**In this lesson, you should have learned about statements that control access to the database and database objects.**

Statement	Action
CREATE USER	Creates a user (usually performed by a DBA)
GRANT	Gives other users privileges to access the objects
CREATE ROLE	Creates a collection of privileges (usually performed by a DBA)
ALTER USER	Changes a user's password
REVOKE	Removes privileges on an object from users

ORACLE

1-19

Copyright © 2004, Oracle. All rights reserved.

## Summary

DBAs establish initial database security for users by assigning privileges to the users.

- The DBA creates users who must have a password. The DBA is also responsible for establishing the initial system privileges for a user.
- After the user has created an object, the user can pass along any of the available object privileges to other users or to all users by using the GRANT statement.
- A DBA can create roles by using the CREATE ROLE statement to pass along a collection of system or object privileges to multiple users. Roles make granting and revoking privileges easier to maintain.
- Users can change their password by using the ALTER USER statement.
- You can remove privileges from users by using the REVOKE statement.
- With data dictionary views, users can view the privileges granted to them and those that are granted on their objects.
- With database links, you can access data on remote databases. Privileges cannot be granted on remote objects.

# Practice 1: Overview

**This practice covers the following topics:**

- **Granting other users privileges to your table**
- **Modifying another user's table through the privileges granted to you**
- **Creating a synonym**
- **Querying the data dictionary views related to privileges**

**ORACLE**

1-20

Copyright © 2004, Oracle. All rights reserved.

## **Practice 1: Overview**

Team up with other students for this exercise about controlling access to database objects.



## Practice 1

To complete questions 6 and higher, you will need to connect to the database using iSQL\*Plus. To do this, launch the Internet Explorer browser from the desktop of your client. Enter the URL in the *http://machinename:5561/isqlplus/* format and use the *oraxx account* and the corresponding *password* and service identifier (in the *Tx* format) provided by your instructor to log on to the database.

1. What privilege should a user be given to log on to the Oracle server? Is this a system or an object privilege?  
\_\_\_\_\_
2. What privilege should a user be given to create tables?  
\_\_\_\_\_
3. If you create a table, who can pass along privileges to other users on your table?  
\_\_\_\_\_
4. You are the DBA. You are creating many users who require the same system privileges. What should you use to make your job easier?  
\_\_\_\_\_
5. What command do you use to change your password?  
\_\_\_\_\_
6. Grant another user access to your DEPARTMENTS table. Have the user grant you query access to his or her DEPARTMENTS table.
7. Query all the rows in your DEPARTMENTS table.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID	
10	Administration	200	1700	
20	Marketing	201	1800	
30	Purchasing	114	1700	
40	Human Resources	203	2400	
50	Shipping	121	1500	
60	IT	103	1400	
...	70	Public Relations	204	2700
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID	
250	Retail Sales		1700	
260	Recruiting		1700	
270	Payroll		1700	

27 rows selected.

## Practice 1 (continued)

8. Add a new row to your DEPARTMENTS table. Team 1 should add Education as department number 500. Team 2 should add Human Resources as department number 510. Query the other team's table.
9. Create a synonym for the other team's DEPARTMENTS table.
10. Query all the rows in the other team's DEPARTMENTS table by using your synonym.

*Team 1 SELECT statement results:*

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
500	Education		
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400

...

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
240	Government Sales		1700
250	Retail Sales		1700
260	Recruiting		1700
270	Payroll		1700

28 rows selected.

*Team 2 SELECT statement results:*

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700

...

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
250	Retail Sales		1700
260	Recruiting		1700
270	Payroll		1700
510	Human Resources		

28 rows selected.

## Practice 1 (continued)

11. Query the USER\_TABLES data dictionary to see information about the tables that you own.

TABLE_NAME
JOB_HISTORY
EMPLOYEES
JOBS
DEPARTMENTS
LOCATIONS
REGIONS
COUNTRIES

7 rows selected.

12. Query the ALL\_TABLES data dictionary view to see information about all the tables that you can access. Exclude tables that you own.

**Note:** Your list may not exactly match the list shown below.

TABLE_NAME	OWNER
DUAL	SYS
SYSTEM_PRIVILEGE_MAP	SYS
...	
WK\$ACL_SNAPSHOT	WKSYS
DEPARTMENTS	ORA2

13. Revoke the SELECT privilege from the other team.
14. Remove the row you inserted into the DEPARTMENTS table in step 8 and save the changes.



# 2

## Manage Schema Objects

ORACLE®

Copyright © 2004, Oracle. All rights reserved.

# Objectives

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

- **Add constraints**
- **Create indexes**
- **Create indexes using the `CREATE TABLE` statement**
- **Creating function-based indexes**
- **Drop columns and set column `UNUSED`**
- **Perform `FLASHBACK` operations**
- **Create and use external tables**

**ORACLE**

2-2

Copyright © 2004, Oracle. All rights reserved.

## Objectives

This lesson contains information about creating indexes and constraints, and altering existing objects. You also learn about external tables, and the provision to name the index at the time of creating a primary key constraint.

# The ALTER TABLE Statement

Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column
- Define a default value for the new column
- Drop a column

ORACLE

2-3

Copyright © 2004, Oracle. All rights reserved.

## The ALTER TABLE Statement

After you create a table, you may need to change the table structure because you omitted a column, your column definition needs to be changed, or you need to remove columns. You can do this by using the ALTER TABLE statement.

# The ALTER TABLE Statement

Use the ALTER TABLE statement to add, modify, or drop columns.

```
ALTER TABLE table
ADD          (column datatype [DEFAULT expr]
             [, column datatype]...);
```

```
ALTER TABLE table
MODIFY       (column datatype [DEFAULT expr]
             [, column datatype]...);
```

```
ALTER TABLE table
DROP        (column);
```

ORACLE

2-4

Copyright © 2004, Oracle. All rights reserved.

## The ALTER TABLE Statement (continued)

You can add columns to a table, modify columns, and drop columns from a table by using the ALTER TABLE statement.

In the syntax:

<i>table</i>	is the name of the table
ADD   MODIFY   DROP	is the type of modification
<i>column</i>	is the name of the new column
<i>datatype</i>	is the data type and length of the new column
DEFAULT <i>expr</i>	specifies the default value for a new column



# Adding a Column

- You use the ADD clause to add columns.

```
ALTER TABLE dept80
ADD      (job_id VARCHAR2(9));
Table altered.
```

- The new column becomes the last column.

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
145	Russell	14000	01-OCT-96	
146	Partners	13500	05-JAN-97	
147	Errazuriz	12000	10-MAR-97	
148	Cambrault	11000	15-OCT-99	
149	Zlotkey	10500	29-JAN-00	

...

ORACLE

2-5

Copyright © 2004, Oracle. All rights reserved.

## Guidelines for Adding a Column

- You can add or modify columns.
- You cannot specify where the column is to appear. The new column becomes the last column.

The example on the slide adds a column named `JOB_ID` to the `DEPT80` table. The `JOB_ID` column becomes the last column in the table.

**Note:** If a table already contains rows when a column is added, then the new column is initially null for all the rows. You cannot add a mandatory `NOT NULL` column to a table that contains data in the other columns. You can only add a `NOT NULL` column to an empty table.

## Modifying a Column

- You can change a column's data type, size, and default value.

```
ALTER TABLE dept80  
MODIFY      (last_name VARCHAR2(30));  
Table altered.
```

- A change to the default value affects only subsequent insertions to the table.

ORACLE

2-6

Copyright © 2004, Oracle. All rights reserved.

### Modifying a Column

You can modify a column definition by using the `ALTER TABLE` statement with the `MODIFY` clause. Column modification can include changes to a column's data type, size, and default value.

#### Guidelines

- You can increase the width or precision of a numeric column.
- You can increase the width of numeric or character columns.
- You can decrease the width of a column if:
  - The column contains only null values
  - The table has no rows
  - The decrease in column width is not less than the existing values in that column
- You can change the data type if the column contains only null values. The exception to this is `CHAR` to `VARCHAR2` conversions, which can be done with data in the columns.
- You can convert a `CHAR` column to the `VARCHAR2` data type or convert a `VARCHAR2` column to the `CHAR` data type only if the column contains null values or if you do not change the size.
- A change to the default value of a column affects only subsequent insertions to the table.

# Dropping a Column

Use the **DROP COLUMN** clause to drop columns you no longer need from the table.

```
ALTER TABLE dept80
DROP COLUMN job_id;
Table altered.
```

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
145	Russell	14000	01-OCT-96
146	Partners	13500	05-JAN-97
147	Errazuriz	12000	10-MAR-97
148	Cambrault	11000	15-OCT-99
149	Zlotkey	10500	29-JAN-00

ORACLE

2-7

Copyright © 2004, Oracle. All rights reserved.

## Dropping a Column

You can drop a column from a table by using the **ALTER TABLE** statement with the **DROP COLUMN** clause.

### Guidelines

- The column may or may not contain data.
- Using the **ALTER TABLE** statement, only one column can be dropped at a time.
- The table must have at least one column remaining in it after it is altered.
- After a column is dropped, it cannot be recovered.
- A column cannot be dropped if it is part of a constraint or part of an index key unless the cascade option is added.
- Dropping a column can take a while if the column has a large number of values. In this case it may be better to set it to be unused and drop it when the number of users on the system are fewer to avoid extended locks.

**Note:** Certain columns can never be dropped such as columns that form part of the partitioning key of a partitioned table or columns that form part of the primary key of an index-organized table.

## The SET UNUSED Option

- You use the SET UNUSED option to mark one or more columns as unused.
- You use the DROP UNUSED COLUMNS option to remove the columns that are marked as unused.

```
ALTER TABLE <table_name>  
SET UNUSED(<column_name>);  
OR  
ALTER TABLE <table_name>  
SET UNUSED COLUMN <column_name>;  
  
ALTER TABLE <table_name>  
DROP UNUSED COLUMNS;
```

ORACLE

2-8

Copyright © 2004, Oracle. All rights reserved.

### The SET UNUSED Option

The SET UNUSED option marks one or more columns as unused so that they can be dropped when the demand on system resources is lower. Specifying this clause does not actually remove the target columns from each row in the table (that is, it does not restore the disk space used by these columns). Therefore, the response time is faster than if you executed the DROP clause.

Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked as unused, you have no access to that column. A SELECT \* query will not retrieve data from unused columns. In addition, the names and types of columns marked unused will not be displayed during a DESCRIBE statement, and you can add to the table a new column with the same name as an unused column. SET UNUSED information is stored in the USER\_UNUSED\_COL\_TABS dictionary view.

**Note:** The guidelines for setting a column to be UNUSED are similar to those of dropping a column.

## The DROP UNUSED COLUMNS Option

DROP UNUSED COLUMNS removes from the table all columns currently marked as unused. You can use this statement when you want to reclaim the extra disk space from unused columns in the table. If the table contains no unused columns, the statement returns with no errors.

```
ALTER TABLE dept80
SET UNUSED (last_name);
Table altered.
```

```
ALTER TABLE dept80
DROP UNUSED COLUMNS;
Table altered.
```

# Adding a Constraint Syntax

Use the **ALTER TABLE** statement to:

- Add or drop a constraint, but not modify its structure
- Enable or disable constraints
- Add a **NOT NULL** constraint by using the **MODIFY** clause

```
ALTER TABLE <table_name>
ADD [CONSTRAINT <constraint_name>]
type (<column_name>);
```

ORACLE®

2-10

Copyright © 2004, Oracle. All rights reserved.

## Adding a Constraint

You can add a constraint for existing tables by using the **ALTER TABLE** statement with the **ADD** clause.

In the syntax:

<i>table</i>	is the name of the table
<i>constraint</i>	is the name of the constraint
<i>type</i>	is the constraint type
<i>column</i>	is the name of the column affected by the constraint

The constraint name syntax is optional, although recommended. If you do not name your constraints, the system will generate constraint names.

### Guidelines

- You can add, drop, enable, or disable a constraint, but you cannot modify its structure.
- You can add a **NOT NULL** constraint to an existing column by using the **MODIFY** clause of the **ALTER TABLE** statement.

**Note:** You can define a **NOT NULL** column only if the table is empty or if the column has a value for every row.

# Adding a Constraint

**Add a FOREIGN KEY constraint to the EMP2 table indicating that a manager must already exist as a valid employee in the EMP2 table.**

```
ALTER TABLE emp2
modify employee_id Primary Key;
Table altered.
```

```
ALTER TABLE emp2
ADD CONSTRAINT emp_mgr_fk
    FOREIGN KEY(manager_id)
    REFERENCES emp2(employee_id);
Table altered.
```

ORACLE

2-11

Copyright © 2004, Oracle. All rights reserved.

## Adding a Constraint (continued)

The first example on the slide modifies the EMP2 table to add a PRIMARY KEY constraint on the EMPLOYEE\_ID column. Note that because no constraint name is provided, the constraint is automatically named by the Oracle server. The second example on the slide creates a FOREIGN KEY constraint on the EMP2 table. The constraint ensures that a manager exists as a valid employee in the EMP2 table.

## ON DELETE CASCADE

**Delete child rows when a parent key is deleted.**

```
ALTER TABLE Emp2 ADD CONSTRAINT emp_dt_fk  
FOREIGN KEY (Department_id)  
REFERENCES departments ON DELETE CASCADE);  
Table altered.
```

ORACLE

2-12

Copyright © 2004, Oracle. All rights reserved.

### ON DELETE CASCADE

The ON DELETE CASCADE action allows parent key data that is referenced from the child table to be deleted, but not updated. When data in the parent key is deleted, all rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the ON DELETE CASCADE option in the definition of the FOREIGN KEY constraint.



# Deferring Constraints

Constraints can have the following attributes:

- DEFERRABLE or NOT DEFERRABLE
- INITIALLY DEFERRED or INITIALLY IMMEDIATE

```
ALTER TABLE dept2  
ADD CONSTRAINT dept2_id_pk  
PRIMARY KEY (department_id)  
DEFERRABLE INITIALLY DEFERRED
```

Deferring constraint on creation

```
SET CONSTRAINTS dept2_id_pk IMMEDIATE
```

Changing a specific constraint attribute

```
ALTER SESSION  
SET CONSTRAINTS= IMMEDIATE
```

Changing all constraints for a session

ORACLE

2-13

Copyright © 2004, Oracle. All rights reserved.

## Deferring Constraints

You can defer checking constraints for validity until the end of the transaction. A constraint is deferred if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then commit causes the transaction to roll back. If a constraint is immediate (not deferred), then it is checked at the end of each statement. If it is violated, the statement is rolled back immediately. If a constraint causes an action (for example, DELETE CASCADE), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate. Use the SET CONSTRAINTS statement to specify, for a particular transaction, whether a deferrable constraint is checked following each DML statement or when the transaction is committed. In order to create deferrable constraints, you must create a nonunique index for that constraint.

You can define constraints as either deferrable or not deferrable, and either initially deferred or initially immediate. These attributes can be different for each constraint.

**Usage scenario:** Company policy dictates that department number 40 should be changed to 45. Changing the DEPARTMENT\_ID column affects employees assigned to this department. Therefore, you make the primary key and foreign keys deferrable and initially deferred. You update both department and employee information and at the time of commit all rows are validated.

## Dropping a Constraint

- Remove the manager constraint from the EMP2 table.

```
ALTER TABLE emp2
DROP CONSTRAINT emp_mgr_fk;
Table altered.
```

- Remove the PRIMARY KEY constraint on the DEPT2 table and drop the associated FOREIGN KEY constraint on the EMP2.DEPARTMENT\_ID column.

```
ALTER TABLE dept2
DROP PRIMARY KEY CASCADE;
Table altered.
```

ORACLE

2-14

Copyright © 2004, Oracle. All rights reserved.

### Dropping a Constraint

To drop a constraint, you can identify the constraint name from the USER\_CONSTRAINTS and USER\_CONS\_COLUMNS data dictionary views. Then use the ALTER TABLE statement with the DROP clause. The CASCADE option of the DROP clause causes any dependent constraints also to be dropped.

#### Syntax

```
ALTER TABLE table
DROP PRIMARY KEY | UNIQUE (column) |
CONSTRAINT constraint [CASCADE];
```

In the syntax:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column affected by the constraint
<i>constraint</i>	is the name of the constraint

When you drop an integrity constraint, that constraint is no longer enforced by the Oracle server and is no longer available in the data dictionary.

# Disabling Constraints

- Execute the **DISABLE** clause of the **ALTER TABLE** statement to deactivate an integrity constraint.
- Apply the **CASCADE** option to disable dependent integrity constraints.

```
ALTER TABLE emp2
DISABLE CONSTRAINT emp_dt_fk;
Table altered.
```

ORACLE

2-15

Copyright © 2004, Oracle. All rights reserved.

## Disabling a Constraint

You can disable a constraint without dropping it or re-creating it by using the **ALTER TABLE** statement with the **DISABLE** clause.

### Syntax

```
ALTER TABLE table
DISABLE CONSTRAINT constraint [CASCADE];
```

In the syntax:

*table* is the name of the table  
*constraint* is the name of the constraint

### Guidelines

- You can use the **DISABLE** clause in both the **CREATE TABLE** statement and the **ALTER TABLE** statement.
- The **CASCADE** clause disables dependent integrity constraints.
- Disabling a unique or primary key constraint removes the unique index.

# Enabling Constraints

- **Activate an integrity constraint currently disabled in the table definition by using the `ENABLE` clause.**

```
ALTER TABLE      emp2
ENABLE CONSTRAINT emp_dt_fk;
Table altered.
```

- **A `UNIQUE` index is automatically created if you enable a `UNIQUE` key or `PRIMARY KEY` constraint.**

ORACLE®

2-16

Copyright © 2004, Oracle. All rights reserved.

## Enabling a Constraint

You can enable a constraint without dropping it or re-creating it by using the `ALTER TABLE` statement with the `ENABLE` clause.

### Syntax

```
ALTER TABLE      table
ENABLE CONSTRAINT constraint;
```

In the syntax:

*table* is the name of the table  
*constraint* is the name of the constraint

### Guidelines

- If you enable a constraint, that constraint applies to all the data in the table. All the data in the table must comply with the constraint.
- If you enable a `UNIQUE` key or `PRIMARY KEY` constraint, a `UNIQUE` or `PRIMARY KEY` index is created automatically. If an index already exists, then it can be used by these keys.
- You can use the `ENABLE` clause in both the `CREATE TABLE` statement and the `ALTER TABLE` statement.

## Enabling a Constraint (continued)

### Guidelines (continued)

- Enabling a primary key constraint that was disabled with the `CASCADE` option does not enable any foreign keys that are dependent on the primary key.
- To enable a `UNIQUE` or `PRIMARY KEY` constraint, you must have the privileges necessary to create an index on the table.

# Cascading Constraints

- The **CASCADE CONSTRAINTS** clause is used along with the **DROP COLUMN** clause.
- The **CASCADE CONSTRAINTS** clause drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped columns.
- The **CASCADE CONSTRAINTS** clause also drops all multicolumn constraints defined on the dropped columns.

ORACLE

2-18

Copyright © 2004, Oracle. All rights reserved.

## Cascading Constraints

This statement illustrates the usage of the **CASCADE CONSTRAINTS** clause. Assume that table **TEST1** is created as follows:

```
CREATE TABLE test1 (  
  pk NUMBER PRIMARY KEY,  
  fk NUMBER,  
  col1 NUMBER,  
  col2 NUMBER,  
  CONSTRAINT fk_constraint FOREIGN KEY (fk) REFERENCES test1,  
  CONSTRAINT ck1 CHECK (pk > 0 and col1 > 0),  
  CONSTRAINT ck2 CHECK (col2 > 0));
```

An error is returned for the following statements:

```
ALTER TABLE test1 DROP (pk); —pk is a parent key.  
ALTER TABLE test1 DROP (col1); —col1 is referenced by multicolumn  
constraint ck1.
```

# Cascading Constraints

## Example:

```
ALTER TABLE emp2
DROP COLUMN employee_id CASCADE CONSTRAINTS;
Table altered.
```

```
ALTER TABLE test1
DROP (pk, fk, coll) CASCADE CONSTRAINTS;
Table altered.
```

ORACLE

2-19

Copyright © 2004, Oracle. All rights reserved.

## Cascading Constraints (continued)

Submitting the following statement drops column EMPLOYEE\_ID, the primary key constraint, and any foreign key constraints referencing the primary key constraint for the EMP2 table:

```
ALTER TABLE emp2 DROP COLUMN employee_id CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, then CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to column PK, it is valid to submit the following statement without the CASCADE CONSTRAINTS clause for the TEST1 table created in the previous page:

```
ALTER TABLE test1 DROP (pk, fk, coll);
```

# Overview of Indexes

## Indexes are created:

- **Automatically**
  - PRIMARY KEY creation
  - UNIQUE KEY creation
- **Manually**
  - CREATE INDEX statement
  - CREATE TABLE statement

ORACLE

2-20

Copyright © 2004, Oracle. All rights reserved.

## Overview of Indexes

Two types of indexes can be created. One type is a unique index. The Oracle server automatically creates a unique index when you define a column or group of columns in a table to have a PRIMARY KEY or a UNIQUE key constraint. The name of the index is the name given to the constraint.

The other type of index is a nonunique index, which a user can create. For example, you can create an index for a FOREIGN KEY column to be used in joins to improve retrieval speed.

You can create an index on one or more columns by issuing the CREATE INDEX statement.

For more information, see *Oracle Database 10g SQL Reference*.

**Note:** You can manually create a unique index, but it is recommended that you create a unique constraint, which implicitly creates a unique index.



## CREATE INDEX with CREATE TABLE Statement

```
CREATE TABLE NEW_EMP
(employee_id NUMBER(6)
PRIMARY KEY USING INDEX
(CREATE INDEX emp_id_idx ON
NEW_EMP(employee_id)),
first_name VARCHAR2(20),
last_name VARCHAR2(25));
Table created.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM USER_INDEXES
WHERE TABLE_NAME = 'NEW_EMP';
```

INDEX_NAME	TABLE_NAME
EMP_ID_IDX	NEW_EMP

ORACLE

2-21

Copyright © 2004, Oracle. All rights reserved.

### CREATE INDEX with CREATE TABLE Statement

In the example on the slide, the CREATE INDEX clause is used with the CREATE TABLE statement to create a primary key index explicitly. You can name your indexes at the time of primary key creation to be different from the name of the PRIMARY KEY constrain. The following example illustrates the database behavior if the index is not explicitly named:

```
CREATE TABLE EMP_UNNAMED_INDEX
(employee_id NUMBER(6) PRIMARY KEY ,
first_name VARCHAR2(20),
last_name VARCHAR2(25));
```

Table created.

```
SELECT INDEX_NAME, TABLE_NAME
FROM USER_INDEXES
WHERE TABLE_NAME = 'EMP_UNNAMED_INDEX';
```

INDEX_NAME	TABLE_NAME
SYS_C002835	EMP_UNNAMED_INDEX

## **CREATE INDEX with CREATE TABLE Statement (continued)**

Observe that the Oracle server gives a generic name to the index that is created for the PRIMARY KEY column.

You can also use an existing index for your PRIMARY KEY column, for example when you are expecting a large data load and want to speed the operation. You may want to disable the constraints while performing the load and then enable them, in which case having a unique index on the primary key will still cause the data to be verified during the load. So you can first create a nonunique index on the column designated as PRIMARY KEY, and then create the PRIMARY KEY column and specify that it should use the existing index. The following examples illustrate this process:

### **Step 1: Create the table**

```
CREATE TABLE NEW_EMP2
(employee_id NUMBER(6)
first_name  VARCHAR2(20),
last_name   VARCHAR2(25)
);
```

### **Step 2: Create the index**

```
CREATE INDEX emp_id_idx2 ON
new_emp2(employee_id);
```

### **Step 3: Create the Primary Key**

```
ALTER TABLE new_emp2 ADD PRIMARY KEY (employee_id) USING INDEX
emp_id_idx2;
```

# Function-Based Indexes

- A function-based index is based on expressions.
- The index expression is built from table columns, constants, SQL functions, and user-defined functions.

```
CREATE INDEX upper_dept_name_idx  
ON dept2(UPPER(department_name));
```

Index created.

```
SELECT *  
FROM   dept2  
WHERE  UPPER(department_name) = 'SALES';
```

ORACLE

2-23

Copyright © 2004, Oracle. All rights reserved.

## Function-Based Indexes

Function-based indexes defined with the `UPPER(column_name)` or `LOWER(column_name)` keywords allow case-insensitive searches. For example, the following index:

```
CREATE INDEX upper_last_name_idx ON emp2 (UPPER(last_name));
```

facilitates processing queries such as:

```
SELECT * FROM emp2 WHERE UPPER(last_name) = 'KING';
```

The Oracle server uses the index only when that particular function is used in a query. For example, the following statement may use the index, but without the `WHERE` clause the Oracle server may perform a full table scan:

```
SELECT *  
FROM   employees  
WHERE  UPPER (last_name) IS NOT NULL  
ORDER BY UPPER (last_name);
```

**Note:** The `QUERY_REWRITE_ENABLED` initialization parameter must be set to `TRUE` for a function-based index to be used.

## **Function-Based Indexes (continued)**

The Oracle server treats indexes with columns marked `DESC` as function-based indexes. The columns marked `DESC` are sorted in descending order.

## Removing an Index

- Remove an index from the data dictionary by using the `DROP INDEX` command.

```
DROP INDEX index;
```

- Remove the `UPPER_DEPT_NAME_IDX` index from the data dictionary.

```
DROP INDEX upper_dept_name_idx;  
Index dropped.
```

- To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

ORACLE

2-25

Copyright © 2004, Oracle. All rights reserved.

### Removing an Index

You cannot modify indexes. To change an index, you must drop it and then re-create it. Remove an index definition from the data dictionary by issuing the `DROP INDEX` statement. To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

In the syntax:

*index* is the name of the index

**Note:** If you drop a table, indexes and constraints are automatically dropped, but views and sequences remain.

## DROP TABLE ... PURGE

```
DROP TABLE dept80 PURGE;
```

ORACLE

2-26

Copyright © 2004, Oracle. All rights reserved.

### DROP TABLE ...PURGE

Oracle Database 10g introduces a new feature for dropping tables. When you drop a table, the database does not immediately release the space associated with the table. Rather, the database renames the table and places it in a recycle bin, where it can later be recovered with the `FLASHBACK TABLE` statement if you find that you dropped the table in error. If you want to immediately release the space associated with the table at the time you issue the `DROP TABLE` statement, then include the `PURGE` clause as shown in the statement on the slide.

Specify `PURGE` only if you want to drop the table and release the space associated with it in a single step. If you specify `PURGE`, then the database does not place the table and its dependent objects into the recycle bin.

Using this clause is equivalent to first dropping the table and then purging it from the recycle bin. This clause saves you one step in the process. It also provides enhanced security if you want to prevent sensitive material from appearing in the recycle bin.

**Note:** You cannot roll back a `DROP TABLE` statement with the `PURGE` clause, nor can you recover the table if you drop it with the `PURGE` clause. This feature was not available in earlier releases.

# The FLASHBACK TABLE Statement

- **Repair tool for accidental table modifications**
  - Restores a table to an earlier point in time
  - Benefits: Ease of use, availability, fast execution
  - Performed in place
- **Syntax:**

```
FLASHBACK TABLE[schema.]table[,  
[ schema.]table ]...  
TO { TIMESTAMP | SCN } expr  
[ { ENABLE | DISABLE } TRIGGERS ];
```

ORACLE

2-27

Copyright © 2004, Oracle. All rights reserved.

## The FLASHBACK TABLE Statement

### Self-Service Repair Facility

Oracle Database 10g provides a new SQL DDL command, `FLASHBACK TABLE`, to restore the state of a table to an earlier point in time in case it is inadvertently deleted or modified. The `FLASHBACK TABLE` command is a self-service repair tool to restore data in a table along with associated attributes such as indexes or views. This is done while the database is online by rolling back only the subsequent changes to the given table. Compared to traditional recovery mechanisms, this feature offers significant benefits such as ease of use, availability, and faster restoration. It also takes the burden off the DBA to find and restore application-specific properties. The flashback table feature does not address physical corruption caused because of a bad disk.

### Syntax

You can invoke a flashback table operation on one or more tables, even on tables in different schemas. You specify the point in time to which you want to revert by providing a valid timestamp. By default, database triggers are disabled for all tables involved. You can override this default behavior by specifying the `ENABLE TRIGGERS` clause.

**Note:** For more information about recycle bin and flashback semantics, refer to *Oracle Database Administrator's Reference 10g Release 1 (10.1)*.

# The FLASHBACK TABLE Statement

```
DROP TABLE emp2;  
Table dropped
```

```
SELECT original_name, operation, droptime,  
FROM recyclebin;
```

ORIGINAL_NAME	OPERATION	DROPTIME
EMP2	DROP	2004-03-03:07:57:11

...

```
FLASHBACK TABLE emp2 TO BEFORE DROP;  
Flashback complete
```

ORACLE

2-28

Copyright © 2004, Oracle. All rights reserved.

## The FLASHBACK TABLE Statement (continued)

### Syntax and Examples

The example restores the EMP2 table to a state prior to a DROP statement.

The recycle bin is actually a data dictionary table containing information about dropped objects. Dropped tables and any associated objects, such as indexes, constraints, nested tables, and so on, are not removed and still occupy space. They continue to count against user space quotas, until specifically purged from the recycle bin or the unlikely situation where they must be purged by the database because of tablespace space constraints.

Each user can be thought of as an owner of a recycle bin because, unless a user has the SYSDBA privilege, the only objects that the user has access to in the recycle bin are those that the user owns. A user can view his objects in the recycle bin using the following statement:

```
SELECT * FROM RECYCLEBIN;
```

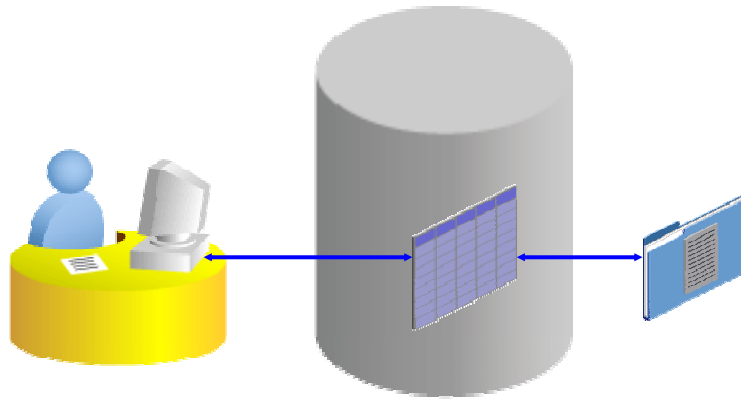
When you drop a user, any objects belonging to that user are not placed in the recycle bin and any objects in the recycle bin are purged.

You can purge the recycle bin with the following statement:

```
PURGE RECYCLEBIN;
```



# External Tables



ORACLE®

2-29

Copyright © 2004, Oracle. All rights reserved.

## External Tables

An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database. This external table definition can be thought of as a view that is used for running any SQL query against external data without requiring that the external data first be loaded into the database. The external table data can be queried and joined directly and in parallel without requiring that the external data first be loaded in the database. You can use SQL, PL/SQL, and Java to query the data in an external table.

The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations are possible, and no indexes can be created on them. However, you can create an external table, and thus unload data, by using the `CREATE TABLE AS SELECT` command.

The Oracle server provides two major access drivers for external tables. One, the loader access driver (or `ORACLE_LOADER`), is used for reading of data from external files whose format can be interpreted by the `SQL*Loader` utility. Note that not all `SQL*Loader` functionality is supported with external tables.

## External Tables (continued)

The ORACLE\_DATAPUMP access driver can be used to both import and export data using a platform-independent format. The ORACLE\_DATAPUMP access driver writes rows from a SELECT statement to be loaded into an external table as part of a CREATE TABLE ... ORGANIZATION EXTERNAL ... AS SELECT statement. You can then use SELECT to read data out of that data file. You can also create an external table definition on another system and use that data file. This allows data to be moved between Oracle databases.

## Creating a Directory for the External Table

**Create a DIRECTORY object that corresponds to the directory on the file system where the external data source resides.**

```
CREATE OR REPLACE DIRECTORY emp_dir
AS '/.../emp_dir';

GRANT READ ON DIRECTORY emp_dir TO hr;
```

ORACLE®

2-31

Copyright © 2004, Oracle. All rights reserved.

### Example of Creating an External Table

Use the `CREATE DIRECTORY` statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where an external data source resides. You can use directory names when referring to an external data source, rather than hard code the operating system path name, for greater file management flexibility.

You must have `CREATE ANY DIRECTORY` system privileges to create directories. When you create a directory, you are automatically granted the `READ` and `WRITE` object privileges and can grant `READ` and `WRITE` privileges to other users and roles. The DBA can also grant these privileges to other users and roles.

A user needs `READ` privileges for all directories used in external tables to be accessed and `WRITE` privileges for the log, bad, and discard file locations being used.

In addition, a `WRITE` privilege is necessary when the external table framework is being used to unload data.

Oracle also provides the `ORACLE_DATAPUMP` type, with which you can unload data (that is, read data from a table in the database and insert it into an external table) and then reload it into an Oracle database. This is a one-time operation that can be done when the table is created. After the creation and initial population is done, you cannot update, insert, or delete any rows.

## Example of Creating an External Table (continued)

### Syntax

```
CREATE [OR REPLACE] DIRECTORY AS 'path_name' ;
```

In the syntax:

OR REPLACE	Specify OR REPLACE to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory. Users who were previously granted privileges on a redefined directory can continue to access the directory without requiring that the privileges be regranted.
directory	Specify the name of the directory object to be created. The maximum length of the directory name is 30 bytes. You cannot qualify a directory object with a schema name.
'path_name'	Specify the full path name of the operating system directory on the result that the path name is case sensitive.

The syntax for using the ORACLE\_DATAPUMP access driver is as follows:

```
CREATE TABLE extract_emps
ORGANIZATION EXTERNAL (TYPE ORACLE_DATAPUMP
                        DEFAULT DIRECTORY ...
                        ACCESS PARAMETERS (... )
                        LOCATION (... )
                        PARALLEL 4
                        REJECT LIMIT UNLIMITED
AS
SELECT * FROM ...;
```

## Creating an External Table

```
CREATE TABLE <table_name>
  ( <col_name> <datatype>, ... )
  ORGANIZATION EXTERNAL
    (TYPE <access_driver_type>
     DEFAULT DIRECTORY <directory_name>
     ACCESS PARAMETERS
       (... ) )
     LOCATION ('<location_specifier>') )
  REJECT LIMIT [0 | <number> | UNLIMITED];
```

ORACLE®

2-33

Copyright © 2004, Oracle. All rights reserved.

### Creating an External Table

You create external tables using the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement. You are not, in fact, creating a table. Rather, you are creating metadata in the data dictionary that you can use to access external data. You use the `ORGANIZATION` clause to specify the order in which the data rows of the table are stored. By specifying `EXTERNAL` in the `ORGANIZATION` clause, you indicate that the table is a read-only table located outside the database. Note that the external files must already exist outside the database.

`TYPE <access_driver_type>` indicates the access driver of the external table. The access driver is the application programming interface (API) that interprets the external data for the database. If you do not specify `TYPE`, Oracle uses the default access driver, `ORACLE_LOADER`. The other option is the `ORACLE_DATAPUMP`.

You use the `DEFAULT DIRECTORY` clause to specify one or more Oracle database directory objects that correspond to directories on the file system where the external data sources may reside.

The optional `ACCESS PARAMETERS` clause enables you to assign values to the parameters of the specific access driver for this external table.

### **Creating an External Table (continued)**

Use the `LOCATION` clause to specify one external locator for each external data source. Usually, the `<location_specifier>` is a file, but it need not be.

The `REJECT LIMIT` clause enables you to specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted. The default value is 0.

## Creating an External Table Using ORACLE\_LOADER

```
CREATE TABLE oldemp (  
  fname char(25), lname CHAR(25))  
  ORGANIZATION EXTERNAL  
  (TYPE ORACLE_LOADER  
  DEFAULT DIRECTORY emp_dir  
  ACCESS PARAMETERS  
  (RECORDS DELIMITED BY NEWLINE  
  NOBADFILE  
  NOLOGFILE  
  FIELDS TERMINATED BY ', '  
  (fname POSITION ( 1:20) CHAR,  
  lname POSITION (22:41) CHAR))  
  LOCATION ('emp.dat'))  
  PARALLEL 5  
  REJECT LIMIT 200;  
Table created.
```

ORACLE

2-35

Copyright © 2004, Oracle. All rights reserved.

### Example of Creating an External Table Using the ORACLE\_LOADER Access Driver

Assume that there is a flat file that has records in the following format:

```
10,jones,11-Dec-1934  
20,smith,12-Jun-1972
```

Records are delimited by new lines, and the fields are all terminated by a comma ( , ). The name of the file is: /emp\_dir/emp.dat

To convert this file as the data source for an external table, whose metadata will reside in the database, you must perform the following steps:

1. Create a directory object emp\_dir as follows:  
CREATE DIRECTORY emp\_dir AS '/emp\_dir' ;
2. Run the CREATE TABLE command shown on the slide.

The example on the slide illustrates the table specification to create an external table for the file:  
/emp\_dir/emp.dat

## **Example of Creating an External Table Using the ORACLE\_LOADER Access Driver (continued)**

In the example, the TYPE specification is given only to illustrate its use. ORACLE\_LOADER is the default access driver if not specified. The ACCESS PARAMETERS option provides values to parameters of the specific access driver, which are interpreted by the access driver, not by the Oracle server.

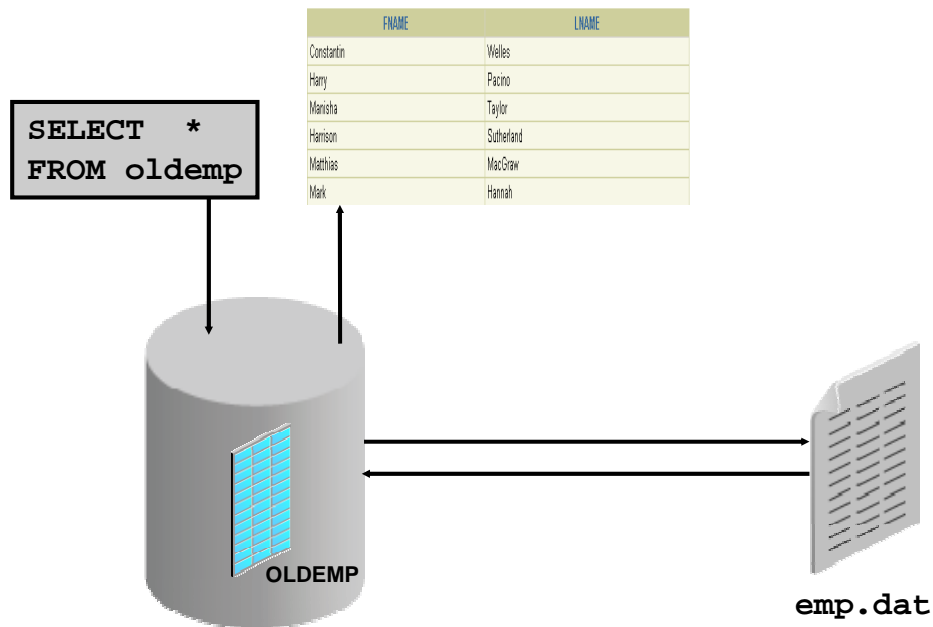
The PARALLEL clause enables five parallel execution servers to simultaneously scan the external data sources (files) when executing the INSERT INTO TABLE statement. For example, if PARALLEL=5 were specified, then more than one parallel execution server can be working on a data source. Because external tables can be very large, for performance reasons it is advisable to specify the PARALLEL clause, or a parallel hint for the query.

The REJECT LIMIT clause specifies that if more than 200 conversion errors occur during a query of the external data, the query is aborted and an error returned. These conversion errors can arise when the access driver tries to transform the data in the data file to match the external table definition.

After the CREATE TABLE command executes successfully, the external table OLDEMP can be described and queried like a relational table.



## Querying External Tables



ORACLE

2-37

Copyright © 2004, Oracle. All rights reserved.

### Querying External Tables

An external table does not describe any data that is stored in the database. Nor does it describe how data is stored in the external source. Instead, it describes how the external table layer must present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

When the database server accesses data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects.

It is important to remember that the description of the data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than there are columns in the table. Also, the data types for fields in the data source can be different from the columns in the table. The access driver takes care of ensuring that the data from the data source is processed so that it matches the definition of the external table.

# Summary

**In this lesson, you should have learned how to:**

- **Add constraints**
- **Create indexes**
- **Create a primary key constraint using an index**
- **Create indexes using the `CREATE TABLE` statement**
- **Creating function-based indexes**
- **Drop columns and set column `UNUSED`**
- **Perform `FLASHBACK` operations**
- **Create and use external tables**

**ORACLE**

2-38

Copyright © 2004, Oracle. All rights reserved.

## Summary

Alter tables to add or modify columns or constraints. Create indexes and function-based indexes using the `CREATE INDEX` statement. Drop unused columns. Use `FLASHBACK` mechanics to restore tables. Use the `external_table` clause to create an external table, which is a read-only table whose metadata is stored in the database but whose data is stored outside the database. Use external tables to query data without first loading it into the database. Name your `PRIMARY KEY` column indexes as you create the table with the `CREATE TABLE` statement.

## Practice 2: Overview

**This practice covers the following topics:**

- **Altering tables**
- **Adding columns**
- **Dropping columns**
- **Creating indexes**
- **Creating external tables**

ORACLE®

2-39

Copyright © 2004, Oracle. All rights reserved.

### **Practice 2: Overview**

In this practice, you use the `ALTER TABLE` command to modify columns and add constraints. You use the `CREATE INDEX` command to create indexes when creating a table, along with the `CREATE TABLE` command. You create external tables. You drop columns and use the `FLASHBACK` operation.

## Practice 2

1. Create the DEPT2 table based on the following table instance chart. Place the syntax in a script called lab\_02\_01.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

<b>Column Name</b>	ID	NAME
<b>Key Type</b>		
<b>Nulls/Unique</b>		
<b>FK Table</b>		
<b>FK Column</b>		
<b>Data type</b>	NUMBER	VARCHAR2
<b>Length</b>	7	25

Name	Null?	Type
ID		NUMBER(7)
NAME		VARCHAR2(25)

2. Populate the DEPT2 table with data from the DEPARTMENTS table. Include only the columns that you need.
3. Create the EMP2 table based on the following table instance chart. Place the syntax in a script called lab\_02\_03.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

<b>Column Name</b>	ID	LAST_NAME	FIRST_NAME	DEPT_ID
<b>Key Type</b>				
<b>Nulls/Unique</b>				
<b>FK Table</b>				
<b>FK Column</b>				
<b>Data type</b>	NUMBER	VARCHAR2	VARCHAR2	NUMBER
<b>Length</b>	7	25	25	7

Name	Null?	Type
ID		NUMBER(7)
LAST_NAME		VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
DEPT_ID		NUMBER(7)

## Practice 2 (continued)

4. Modify the EMP2 table to allow for longer employee last names. Confirm your modification.

Name	Null?	Type
ID		NUMBER(7)
LAST_NAME		VARCHAR2(50)
FIRST_NAME		VARCHAR2(25)
DEPT_ID		NUMBER(7)

5. Confirm that both the DEPT2 and EMP2 tables are stored in the data dictionary. (Hint: USER\_TABLES)

TABLE_NAME
DEPT2
EMP2

6. Create the EMPLOYEES2 table based on the structure of the EMPLOYEES table. Include only the EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, SALARY, and DEPARTMENT\_ID columns. Name the columns in your new table ID, FIRST\_NAME, LAST\_NAME, SALARY, and DEPT\_ID, respectively.
7. Drop the EMP2 table.
8. Query the recycle bin to see whether the table is present.

ORIGINAL_NAME	OPERATION	DROPTIME
EMP2	DROP	2004-02-13:10:40:22

9. Undrop the EMP2 table.

Name	Null?	Type
ID		NUMBER(7)
LAST_NAME		VARCHAR2(50)
FIRST_NAME		VARCHAR2(25)
DEPT_ID		NUMBER(7)

10. Drop the FIRST\_NAME column from the EMPLOYEES2 table. Confirm your modification by checking the description of the table.
11. In the EMPLOYEES2 table, mark the DEPT\_ID column as UNUSED. Confirm your modification by checking the description of the table.
12. Drop all the UNUSED columns from the EMPLOYEES2 table. Confirm your modification by checking the description of the table.
13. Add a table-level PRIMARY KEY constraint to the EMP2 table on the ID column. The constraint should be named at creation. Name the constraint my\_emp\_id\_pk.

## Practice 2 (continued)

14. Create a PRIMARY KEY constraint to the DEPT2 table using the ID column. The constraint should be named at creation. Name the constraint my\_dept\_id\_pk.
15. Add a foreign key reference on the EMP2 table that ensures that the employee is not assigned to a nonexistent department. Name the constraint my\_emp\_dept\_id\_fk.
16. Confirm that the constraints were added by querying the USER\_CONSTRAINTS view. Note the types and names of the constraints.

CONSTRAINT_NAME	CON
MY_DEPT_ID_PK	P
MY_EMP_ID_PK	P
MY_EMP_DEPT_ID_FK	R

17. Display the object names and types from the USER\_OBJECTS data dictionary view for the EMP2 and DEPT2 tables. Notice that the new tables and a new index were created.

If you have time, complete the following exercise:

18. Modify the EMP2 table. Add a COMMISSION column of NUMBER data type, precision 2, scale 2. Add a constraint to the COMMISSION column that ensures that a commission value is greater than zero.
19. Drop the EMP2 and DEPT2 tables so that they cannot be restored. Verify the recycle bin.
20. Create the DEPT\_NAMED\_INDEX table based on the following table instance chart. Name the index for the PRIMARY KEY column as DEPT\_PK\_IDX.

Column Name	Deptno	Dname
Primary Key	Yes	
Data Type	Number	VARCHAR2
Length	4	30

# 3

## Manipulating Large Data Sets

ORACLE®

Copyright © 2004, Oracle. All rights reserved.

# Objectives

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

- **Manipulate data using subqueries**
- **Describe the features of multitable inserts**
- **Use the following types of multitable inserts**
  - **Unconditional INSERT**
  - **Pivoting INSERT**
  - **Conditional ALL INSERT**
  - **Conditional FIRST INSERT**
- **Merge rows in a table**
- **Track the changes to data over a period of time**

**ORACLE**

3-2

Copyright © 2004, Oracle. All rights reserved.

## Objectives

In this lesson, you learn how to manipulate data in the Oracle database by using subqueries. You also learn about multitable insert statements, the MERGE statement, and tracking changes in the database.



# Using Subqueries to Manipulate Data

**You can use subqueries in DML statements to:**

- **Copy data from one table to another**
- **Retrieve data from an inline view**
- **Update data in one table based on the values of another table**
- **Delete rows from one table based on rows in a another table**

**ORACLE**

3-3

Copyright © 2004, Oracle. All rights reserved.

## Using Subqueries to Manipulate Data

Subqueries can be used to retrieve data from a table that you can use as input to an `INSERT` into a different table. In this way you can easily copy large volumes of data from one table to another with one single `SELECT` statement. Similarly, you can use subqueries to do mass updates and deletes by using them in the `WHERE` clause of the `UPDATE` and `DELETE` statements. You can also use subqueries in the `FROM` clause of a `SELECT` statement. This is called an inline view.

# Copying Rows from Another Table

- Write your INSERT statement with a subquery.

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
SELECT employee_id, last_name, salary, commission_pct
FROM employees
WHERE job_id LIKE '%REP%';
```

33 rows created.

- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause with that in the subquery.

ORACLE

3-4

Copyright © 2004, Oracle. All rights reserved.

## Copying Rows from Another Table

You can use the INSERT statement to add rows to a table where the values are derived from existing tables. In place of the VALUES clause, you use a subquery.

### Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

*table* is the table name  
*column* is the name of the column in the table to populate  
*subquery* is the subquery that returns rows into the table

The number of columns and their data types in the column list of the INSERT clause must match the number of values and their data types in the subquery. To create a copy of the rows of a table, use SELECT \* in the subquery.

```
INSERT INTO EMPL3
SELECT *
FROM employees;
```

For more information, see *Oracle Database 10g SQL Reference*.

## Inserting Using a Subquery as a Target

```
INSERT INTO
    (SELECT employee_id, last_name,
            email, hire_date, job_id, salary,
            department_id
     FROM   empl3
     WHERE  department_id = 50)
VALUES (99999, 'Taylor', 'DTAYLOR',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000, 50);

1 row created.
```

ORACLE

3-5

Copyright © 2004, Oracle. All rights reserved.

### Inserting Using a Subquery as a Target

You can use a subquery in place of the table name in the INTO clause of the INSERT statement.

The select list of this subquery must have the same number of columns as the column list of the VALUES clause. Any rules on the columns of the base table must be followed in order for the INSERT statement to work successfully. For example, you cannot put in a duplicate employee ID or leave out a value for a mandatory NOT NULL column.

This application of subqueries helps avoid having to create a view just for performing an INSERT.

## Inserting Using a Subquery as a Target

Verify the results.

```
SELECT employee_id, last_name, email, hire_date,  
       job_id, salary, department_id  
FROM   employees  
WHERE  department_id = 50;
```

EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
120	Weiss	MWEISS	18-JUL-96	ST_MAN	8000	50
121	Fripp	AFRIPP	10-APR-97	ST_MAN	8200	50
122	Kaufling	PKAUFLIN	01-MAY-95	ST_MAN	7900	50
...						
193	Everett	BEVERETT	03-MAR-97	SH_CLERK	3900	50
194	McCain	SMCCAIN	01-JUL-98	SH_CLERK	3200	50
195	Jones	VJONES	17-MAR-99	SH_CLERK	2800	50
196	Walsh	AWALSH	24-APR-98	SH_CLERK	3100	50
197	Feeney	KFEENEY	23-MAY-98	SH_CLERK	3000	50
198	OConnell	DOCONNEL	21-JUN-99	SH_CLERK	2600	50
199	Grant	DGRANT	13-JAN-00	SH_CLERK	2600	50
99999	Taylor	DTAYLOR	07-JUN-99	ST_CLERK	5000	50

46 rows selected.

ORACLE

### Inserting Using a Subquery as a Target (continued)

The example shows the results of the subquery that was used to identify the table for the INSERT statement.

## Retrieving Data with a Subquery as Source

```
SELECT  a.last_name, a.salary,  
        a.department_id, b.salavg  
FROM    employees a, (SELECT  department_id,  
                        AVG(salary) salavg  
                     FROM    employees  
                     GROUP BY department_id) b  
WHERE   a.department_id = b.department_id  
AND     a.salary > b.salavg;
```

LAST_NAME	SALARY	DEPARTMENT_ID	SALAVG
King	24000	90	19333.3333
Hunold	9000	60	5760
Ernst	6000	60	5760
Greenberg	12000	100	8600
Faviet	9000	100	8600
Raphaely	11000	30	4150
Weiss	8000	50	3475.55556
Fripp	8200	50	3475.55556

ORACLE

### Retrieving Data Using a Subquery as Source

You can use a subquery in the FROM clause of a SELECT statement, which is very similar to how views are used. A subquery in the FROM clause of a SELECT statement is also called an *inline* view. A subquery in the FROM clause of a SELECT statement defines a data source for that particular SELECT statement, and only that SELECT statement. The example on the slide displays employee last names, salaries, department numbers, and average salaries for all the employees who earn more than the average salary in their department. The subquery in the FROM clause is named b, and the outer query references the SALAVG column using this alias.

## Updating Two Columns with a Subquery

Update the job and salary of employee 114 to match the job of employee 205 and the salary of employee 168.

```
UPDATE emp13
SET   job_id = (SELECT job_id
                FROM   employees
                WHERE  employee_id = 205),
      salary = (SELECT salary
                FROM   employees
                WHERE  employee_id = 168)
WHERE employee_id = 114;
1 row updated.
```

ORACLE

3-8

Copyright © 2004, Oracle. All rights reserved.

### Updating Two Columns with a Subquery

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries.

#### Syntax

```
UPDATE table
SET   column =
      (SELECT column
       FROM table
       WHERE condition)
[ ,
  column =
      (SELECT column
       FROM table
       WHERE condition) ]
[WHERE condition] ;
```

**Note:** If no rows are updated, a message “0 rows updated.” is returned.

## Updating Rows Based on Another Table

Use subqueries in UPDATE statements to update rows in a table based on values from another table.

```
UPDATE empl3
SET    department_id = (SELECT department_id
                        FROM employees
                        WHERE employee_id = 100)
WHERE  job_id        = (SELECT job_id
                        FROM employees
                        WHERE employee_id = 200);

1 row updated.
```

ORACLE

3-9

Copyright © 2004, Oracle. All rights reserved.

### Updating Rows Based on Another Table

You can use subqueries in UPDATE statements to update rows in a table. The example on the slide updates the EMPL3 table based on the values from the EMPLOYEES table. It changes the department number of all employees with employee 200's job ID to employee 100's current department number.

## Deleting Rows Based on Another Table

Use subqueries in DELETE statements to remove rows from a table based on values from another table.

```
DELETE FROM empl3
WHERE department_id =
    (SELECT department_id
     FROM departments
     WHERE department_name
       LIKE '%Public%');

1 row deleted.
```

ORACLE

3-10

Copyright © 2004, Oracle. All rights reserved.

### Deleting Rows Based on Another Table

You can use subqueries to delete rows from a table based on values from another table. The example on the slide deletes all the employees who are in a department where the department name contains the string “Public.” The subquery searches the DEPARTMENTS table to find the department number based on the department name containing the string “Public.” The subquery then feeds the department number to the main query, which deletes rows of data from the EMPLOYEES table based on this department number.



## Using the WITH CHECK OPTION Keyword on DML Statements

- A subquery is used to identify the table and columns of the DML statement.
- The WITH CHECK OPTION keyword prohibits you from changing rows that are not in the subquery.

```
INSERT INTO (SELECT employee_id, last_name, email,
                hire_date, job_id, salary
            FROM   empl3
            WHERE  department_id = 50
            WITH CHECK OPTION)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000);
INSERT INTO
*
```

**ERROR at line 1:**  
**ORA-01402: view WITH CHECK OPTION where-clause violation**

ORACLE

### The WITH CHECK OPTION Keyword

Specify WITH CHECK OPTION to indicate that, if the subquery is used in place of a table in an INSERT, UPDATE, or DELETE statement, no changes that produce rows that are not included in the subquery are permitted to that table.

In the example shown, the WITH CHECK OPTION keyword is used. The subquery identifies rows that are in department 50, but the department ID is not in the SELECT list, and a value is not provided for it in the VALUES list. Inserting this row results in a department ID of null, which is not in the subquery.

## Overview of the Explicit Default Feature

- **With the explicit default feature, you can use the `DEFAULT` keyword as a column value where the column default is desired.**
- **The addition of this feature is for compliance with the `SQL:1999` standard.**
- **This allows the user to control where and when the default value should be applied to data.**
- **Explicit defaults can be used in `INSERT` and `UPDATE` statements.**

ORACLE

3-12

Copyright © 2004, Oracle. All rights reserved.

### Explicit Defaults

The `DEFAULT` keyword can be used in `INSERT` and `UPDATE` statements to identify a default column value. If no default value exists, a null value is used.

The `DEFAULT` option saves you from hard coding the default value in your programs or querying the dictionary to find it, as was done before this feature was introduced. Hard coding the default is a problem if the default changes because the code consequently needs changing. Accessing the dictionary is not usually done in an application program, so this is a very important feature.

## Using Explicit Default Values

- **DEFAULT with INSERT:**

```
INSERT INTO deptm3  
  (department_id, department_name, manager_id)  
VALUES (300, 'Engineering', DEFAULT);
```

- **DEFAULT with UPDATE:**

```
UPDATE deptm3  
SET manager_id = DEFAULT  
WHERE department_id = 10;
```

ORACLE®

3-13

Copyright © 2004, Oracle. All rights reserved.

### Using Explicit Default Values

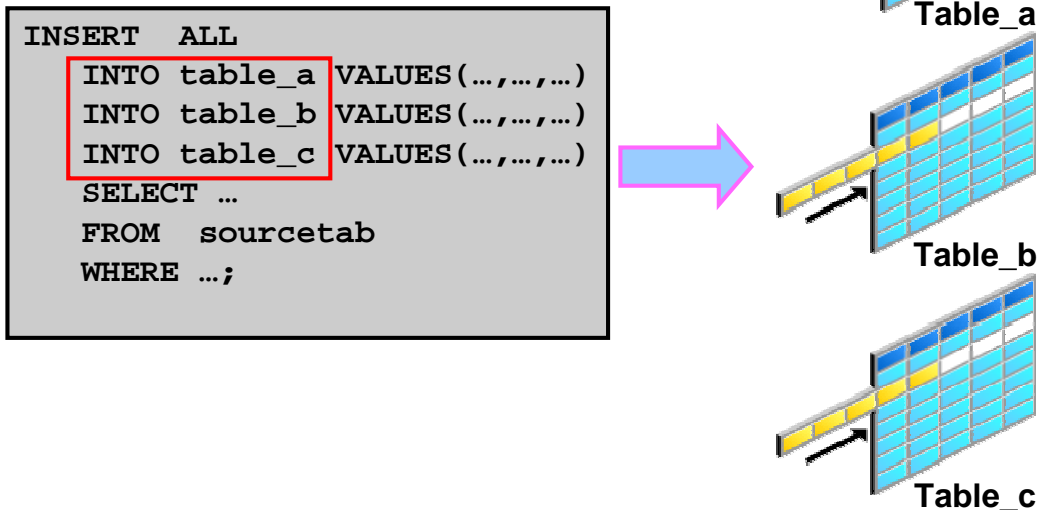
Specify **DEFAULT** to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, the Oracle server sets the column to null.

In the first example on the slide, the **INSERT** statement uses a default value for the **MANAGER\_ID** column. If there is no default value defined for the column, a null value is inserted instead.

The second example uses the **UPDATE** statement to set the **MANAGER\_ID** column to a default value for department 10. If no default value is defined for the column, it changes the value to null.

**Note:** When creating a table, you can specify a default value for a column. This is discussed in the lesson titled “Creating and Managing Tables.”

## Overview of Multitable INSERT Statements



ORACLE®

3-14

Copyright © 2004, Oracle. All rights reserved.

### Overview of Multitable INSERT Statements

In a multitable INSERT statement, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

Multitable INSERT statements can play a very useful role in a data warehouse scenario. You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called ETL, which stands for extraction, transformation, and loading.

During extraction, the desired data must be identified and extracted from many different sources, such as database systems and applications. After extraction, the data must be physically transported to the target system or an intermediate system for further processing. Depending on the chosen means of transportation, some transformations can be done during this process. For example, a SQL statement that directly accesses a remote target through a gateway can concatenate two columns as part of the SELECT statement.

After data is loaded into the Oracle database, data transformations can be executed using SQL operations. A multitable INSERT statement is one of the techniques for implementing SQL data transformations.

## Overview of Multitable INSERT Statements

- The **INSERT...SELECT** statement can be used to insert rows into multiple tables as part of a single DML statement.
- Multitable **INSERT** statements can be used in data warehousing systems to transfer data from one or more operational sources to a set of target tables.
- They provide significant performance improvement over:
  - Single DML versus multiple **INSERT...SELECT** statements
  - Single DML versus a procedure to do multiple inserts using **IF . . . THEN** syntax

ORACLE

3-15

Copyright © 2004, Oracle. All rights reserved.

### Overview of Multitable INSERT Statements (continued)

Multitable **INSERT** statements offer the benefits of the **INSERT . . . SELECT** statement when multiple tables are involved as targets. Using functionality prior to Oracle9i Database, you had to deal with  $n$  independent **INSERT . . . SELECT** statements, thus processing the same source data  $n$  times and increasing the transformation workload  $n$  times.

As with the existing **INSERT . . . SELECT** statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

Each record from any input stream, such as a nonrelational database table, can now be converted into multiple records for a more relational database table environment. To alternatively implement this functionality, you were required to write multiple **INSERT** statements.

# Types of Multitable INSERT Statements

The different types of multitable INSERT statements are:

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

ORACLE

3-16

Copyright © 2004, Oracle. All rights reserved.

## Types of Multitable INSERT Statements

The types of multitable INSERT statements are:

- Unconditional INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
- Pivoting INSERT

You use different clauses to indicate the type of INSERT to be executed.

# Multitable INSERT Statements

- **Syntax**

```
INSERT [ALL] [conditional_insert_clause]  
[insert_into_clause values_clause] (subquery)
```

- **conditional\_insert\_clause**

```
[ALL] [FIRST]  
[WHEN condition THEN] [insert_into_clause values_clause]  
[ELSE] [insert_into_clause values_clause]
```

ORACLE

3-17

Copyright © 2004, Oracle. All rights reserved.

## Multitable INSERT Statements

The slide displays the generic format for multitable INSERT statements.

### **Unconditional INSERT: ALL into\_clause**

Specify ALL followed by multiple insert\_into\_clauses to perform an unconditional multitable insert. The Oracle server executes each insert\_into\_clause once for each row returned by the subquery.

### **Conditional INSERT: conditional\_insert\_clause**

Specify the conditional\_insert\_clause to perform a conditional multitable INSERT. The Oracle server filters each insert\_into\_clause through the corresponding WHEN condition, which determines whether that insert\_into\_clause is executed. A single multitable INSERT statement can contain up to 127 WHEN clauses.

### **Conditional INSERT: ALL**

If you specify ALL, the Oracle server evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, the Oracle server executes the corresponding INTO clause list.

## **Multitable INSERT Statements (continued)**

### **Conditional INSERT: FIRST**

If you specify `FIRST`, the Oracle server evaluates each `WHEN` clause in the order in which it appears in the statement. If the first `WHEN` clause evaluates to true, the Oracle server executes the corresponding `INTO` clause and skips subsequent `WHEN` clauses for the given row.

### **Conditional INSERT: ELSE Clause**

For a given row, if no `WHEN` clause evaluates to true:

- If you have specified an `ELSE` clause, the Oracle server executes the `INTO` clause list associated with the `ELSE` clause.
- If you did not specify an `ELSE` clause, the Oracle server takes no action for that row.

### **Restrictions on Multitable INSERT Statements**

- You can perform multitable `INSERT` statements only on tables, not on views or materialized views.
- You cannot perform a multitable `INSERT` into a remote table.
- You cannot specify a table collection expression when performing a multitable `INSERT`.
- In a multitable `INSERT`, all of the `insert_into_clauses` cannot combine to specify more than 999 target columns.



## Unconditional INSERT ALL

- **Select the EMPLOYEE\_ID, HIRE\_DATE, SALARY, and MANAGER\_ID values from the EMPLOYEES table for those employees whose EMPLOYEE\_ID is greater than 200.**
- **Insert these values into the SAL\_HISTORY and MGR\_HISTORY tables using a multitable INSERT.**

```
INSERT ALL
  INTO sal_history VALUES(EMPID,HIREDATE,SAL)
  INTO mgr_history VALUES(EMPID,MGR,SAL)
  SELECT employee_id EMPID, hire_date HIREDATE,
         salary SAL, manager_id MGR
  FROM employees
  WHERE employee_id > 200;
12 rows created.
```

ORACLE

3-19

Copyright © 2004, Oracle. All rights reserved.

### Unconditional INSERT ALL

The example in the slide inserts rows into both the SAL\_HISTORY and the MGR\_HISTORY tables.

The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of the employee ID, hire date, and salary are inserted into the SAL\_HISTORY table. The details of employee ID, manager ID, and salary are inserted into the MGR\_HISTORY table.

This INSERT statement is referred to as an unconditional INSERT, because no further restriction is applied to the rows that are retrieved by the SELECT statement. All the rows retrieved by the SELECT statement are inserted into the two tables, SAL\_HISTORY and MGR\_HISTORY. The VALUES clause in the INSERT statements specifies the columns from the SELECT statement that must be inserted into each of the tables. Each row returned by the SELECT statement results in two insertions, one for the SAL\_HISTORY table and one for the MGR\_HISTORY table.

The feedback 12 rows created can be interpreted to mean that a total of eight insertions were performed on the base tables, SAL\_HISTORY and MGR\_HISTORY.

## Conditional INSERT ALL

- **Select the EMPLOYEE\_ID, HIRE\_DATE, SALARY, and MANAGER\_ID values from the EMPLOYEES table for those employees whose EMPLOYEE\_ID is greater than 200.**
- **If the SALARY is greater than \$10,000, insert these values into the SAL\_HISTORY table using a conditional multitable INSERT statement.**
- **If the MANAGER\_ID is greater than 200, insert these values into the MGR\_HISTORY table using a conditional multitable INSERT statement.**

ORACLE

3-20

Copyright © 2004, Oracle. All rights reserved.

### Conditional INSERT ALL

The problem statement for a conditional INSERT ALL statement is specified on the slide. The solution to this problem is shown on the next page.

## Conditional INSERT ALL

```
INSERT ALL
  WHEN SAL > 10000 THEN
    INTO sal_history VALUES(EMPID,HIREDATE,SAL)
  WHEN MGR > 200 THEN
    INTO mgr_history VALUES(EMPID,MGR,SAL)
  SELECT employee_id EMPID,hire_date HIREDATE,
         salary SAL, manager_id MGR
  FROM   employees
  WHERE  employee_id > 200;
4 rows created.
```

ORACLE

3-21

Copyright © 2004, Oracle. All rights reserved.

### Conditional INSERT ALL (continued)

The example on the slide is similar to the example on the previous slide because it inserts rows into both the SAL\_HISTORY and the MGR\_HISTORY tables. The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of employee ID, hire date, and salary are inserted into the SAL\_HISTORY table. The details of employee ID, manager ID, and salary are inserted into the MGR\_HISTORY table.

This INSERT statement is referred to as a conditional ALL INSERT, because a further restriction is applied to the rows that are retrieved by the SELECT statement. From the rows that are retrieved by the SELECT statement, only those rows in which the value of the SAL column is more than 10000 are inserted in the SAL\_HISTORY table, and similarly only those rows where the value of the MGR column is more than 200 are inserted in the MGR\_HISTORY table.

Observe that unlike the previous example, where eight rows were inserted into the tables, in this example only four rows are inserted.

The feedback 4 rows created can be interpreted to mean that a total of four inserts were performed on the base tables, SAL\_HISTORY and MGR\_HISTORY.

## Conditional INSERT FIRST

- **Select the `DEPARTMENT_ID`, `SUM(SALARY)`, and `MAX(HIRE_DATE)` from the `EMPLOYEES` table.**
- **If the `SUM(SALARY)` is greater than \$25,000, then insert these values into the `SPECIAL_SAL`, using a conditional `FIRST` multitable `INSERT`.**
- **If the first `WHEN` clause evaluates to true, then the subsequent `WHEN` clauses for this row should be skipped.**
- **For the rows that do not satisfy the first `WHEN` condition, insert into the `HIREDATE_HISTORY_00`, `HIREDATE_HISTORY_99`, or `HIREDATE_HISTORY` tables, based on the value in the `HIRE_DATE` column using a conditional multitable `INSERT`.**

ORACLE

3-22

Copyright © 2004, Oracle. All rights reserved.

### Conditional INSERT FIRST

The problem statement for a conditional `FIRST INSERT` statement is specified on the slide. The solution to this problem is shown on the next page.

## Conditional INSERT FIRST

```
INSERT FIRST
  WHEN SAL > 25000 THEN
    INTO special_sal VALUES(DEPTID, SAL)
  WHEN HIREDATE like ('%00%') THEN
    INTO hiredate_history_00 VALUES(DEPTID, HIREDATE)
  WHEN HIREDATE like ('%99%') THEN
    INTO hiredate_history_99 VALUES(DEPTID, HIREDATE)
  ELSE
    INTO hiredate_history VALUES(DEPTID, HIREDATE)
  SELECT department_id DEPTID, SUM(salary) SAL,
         MAX(hire_date) HIREDATE
  FROM   employees
  GROUP BY department_id;
12 rows created.
```

ORACLE

3-23

Copyright © 2004, Oracle. All rights reserved.

### Conditional INSERT FIRST (continued)

The example on the slide inserts rows into more than one table using a single INSERT statement. The SELECT statement retrieves the details of department ID, total salary, and maximum hire date for every department in the EMPLOYEES table.

This INSERT statement is referred to as a conditional FIRST INSERT, because an exception is made for the departments whose total salary is more than \$25,000. The condition WHEN ALL > 25000 is evaluated first. If the total salary for a department is more than \$25,000, then the record is inserted into the SPECIAL\_SAL table irrespective of the hire date. If this first WHEN clause evaluates to true, the Oracle server executes the corresponding INTO clause and skips subsequent WHEN clauses for this row.

For the rows that do not satisfy the first WHEN condition (WHEN SAL > 25000), the rest of the conditions are evaluated in the same way as a conditional INSERT statement, and the records retrieved by the SELECT statement are inserted into the HIREDATE\_HISTORY\_00, or HIREDATE\_HISTORY\_99, or HIREDATE\_HISTORY tables, based on the value in the HIREDATE column.

The feedback 12 rows created can be interpreted to mean that a total of eight INSERT statements were performed on the base tables, SPECIAL\_SAL, HIREDATE\_HISTORY\_00, HIREDATE\_HISTORY\_99, and HIREDATE\_HISTORY.

## Pivoting INSERT

- **Suppose you receive a set of sales records from a nonrelational database table, `SALES_SOURCE_DATA`, in the following format:**  
`EMPLOYEE_ID, WEEK_ID, SALES_MON, SALES_TUE, SALES_WED, SALES_THUR, SALES_FRI`
- **You want to store these records in the `SALES_INFO` table in a more typical relational format:**  
`EMPLOYEE_ID, WEEK, SALES`
- **Using a pivoting INSERT, convert the set of sales records from the nonrelational database table to relational format.**

ORACLE

3-24

Copyright © 2004, Oracle. All rights reserved.

### Pivoting INSERT

Pivoting is an operation in which you must build a transformation such that each record from any input stream, such as a nonrelational database table, must be converted into multiple records for a more relational database table environment.

To solve the problem mentioned on the slide, you must build a transformation such that each record from the original nonrelational database table, `SALES_SOURCE_DATA`, is converted into five records for the data warehouse's `SALES_INFO` table. This operation is commonly referred to as *pivoting*.

The problem statement for a pivoting INSERT statement is specified on the slide. The solution to this problem is shown on the next page.

# Pivoting INSERT

```
INSERT ALL
  INTO sales_info VALUES (employee_id,week_id,sales_MON)
  INTO sales_info VALUES (employee_id,week_id,sales_TUE)
  INTO sales_info VALUES (employee_id,week_id,sales_WED)
  INTO sales_info VALUES (employee_id,week_id,sales_THUR)
  INTO sales_info VALUES (employee_id,week_id, sales_FRI)
SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
       sales_WED, sales_THUR,sales_FRI
FROM sales_source_data;
5 rows created.
```

ORACLE

3-25

Copyright © 2004, Oracle. All rights reserved.

## Pivoting INSERT (continued)

In the example on the slide, the sales data is received from the nonrelational database table SALES\_SOURCE\_DATA, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID.

DESC SALES\_SOURCE\_DATA

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK_ID		NUMBER(2)
SALES_MON		NUMBER(8,2)
SALES_TUE		NUMBER(8,2)
SALES_WED		NUMBER(8,2)
SALES_THUR		NUMBER(8,2)
SALES_FRI		NUMBER(8,2)

## Pivoting INSERT (continued)

```
SELECT * FROM SALES_SOURCE_DATA;
```

EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
176	6	2000	3000	4000	5000	6000

```
DESC SALES_INFO
```

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK		NUMBER(2)
SALES		NUMBER(8,2)

```
SELECT * FROM sales_info;
```

EMPLOYEE_ID	WEEK	SALES
176	6	2000
176	6	3000
176	6	4000
176	6	5000
176	6	6000

Observe in the preceding example that by using a pivoting INSERT, one row from the SALES\_SOURCE\_DATA table is converted into five records for the relational table, SALES\_INFO.



# The MERGE Statement

- **Provides the ability to conditionally update or insert data into a database table**
- **Performs an UPDATE if the row exists, and an INSERT if it is a new row:**
  - **Avoids separate updates**
  - **Increases performance and ease of use**
  - **Is useful in data warehousing applications**

ORACLE

3-27

Copyright © 2004, Oracle. All rights reserved.

## MERGE Statements

The Oracle server supports the MERGE statement for INSERT, UPDATE, and DELETE operations. Using this statement, you can update, insert, or delete a row conditionally into a table, thus avoiding multiple DML statements. The decision whether to update, insert, or delete into the target table is based on a condition in the ON clause.

You must have the INSERT and UPDATE object privileges on the target table and the SELECT object privilege on the source table. To specify the DELETE clause of the merge\_update\_clause, you must also have the DELETE object privilege on the target table.

The MERGE statement is deterministic. You cannot update the same row of the target table multiple times in the same MERGE statement.

An alternative approach is to use PL/SQL loops and multiple DML statements. The MERGE statement, however, is easy to use and more simply expressed as a single SQL statement.

The MERGE statement is suitable in a number of data warehousing applications. For example, in a data warehousing application you may need to work with data coming from multiple sources, some of which may be duplicates. With the MERGE statement, you can conditionally add or modify rows.

# The MERGE Statement Syntax

You can conditionally insert or update rows in a table by using the MERGE statement.

```
MERGE INTO table_name table_alias
  USING (table/view/sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
      col1 = col_val1,
      col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

ORACLE®

3-28

Copyright © 2004, Oracle. All rights reserved.

## Merging Rows

You can update existing rows and insert new rows conditionally by using the MERGE statement.

In the syntax:

INTO clause	specifies the target table you are updating or inserting into
USING clause	identifies the source of the data to be updated or inserted; can be a table, view, or subquery
ON clause	the condition upon which the MERGE operation either updates or inserts
WHEN MATCHED	instructs the server how to respond to the results of the join condition
WHEN NOT MATCHED	

For more information, see *Oracle Database 10g SQL Reference*, “MERGE.”

# Merging Rows

**Insert or update rows in the EMPL3 table to match the EMPLOYEES table.**

```
MERGE INTO empl3 c
  USING employees e
  ON (c.employee_id = e.employee_id)
  WHEN MATCHED THEN
    UPDATE SET
      c.first_name      = e.first_name,
      c.last_name       = e.last_name,
      ...
      c.department_id  = e.department_id
  WHEN NOT MATCHED THEN
    INSERT VALUES(e.employee_id, e.first_name, e.last_name,
                  e.email, e.phone_number, e.hire_date, e.job_id,
                  e.salary, e.commission_pct, e.manager_id,
                  e.department_id);
```

ORACLE

3-29

Copyright © 2004, Oracle. All rights reserved.

## Example of Merging Rows

```
MERGE INTO empl3 c
  USING employees e
  ON (c.employee_id = e.employee_id)
  WHEN MATCHED THEN
    UPDATE SET
      c.first_name      = e.first_name,
      c.last_name       = e.last_name,
      c.email           = e.email,
      c.phone_number    = e.phone_number,
      c.hire_date       = e.hire_date,
      c.job_id          = e.job_id,
      c.salary          = e.salary,
      c.commission_pct  = e.commission_pct,
      c.manager_id      = e.manager_id,
      c.department_id   = e.department_id
  WHEN NOT MATCHED THEN
    INSERT VALUES(e.employee_id, e.first_name, e.last_name,
                  e.email, e.phone_number, e.hire_date, e.job_id,
                  e.salary, e.commission_pct, e.manager_id,
                  e.department_id);
```

# Merging Rows

```
TRUNCATE TABLE empl3;
```

```
SELECT *  
FROM empl3;  
no rows selected
```

```
MERGE INTO empl3 c  
  USING employees e  
  ON (c.employee_id = e.employee_id)  
WHEN MATCHED THEN  
  UPDATE SET  
    ...  
WHEN NOT MATCHED THEN  
  INSERT VALUES...;
```

```
SELECT *  
FROM empl3;  
107 rows selected.
```

ORACLE

3-30

Copyright © 2004, Oracle. All rights reserved.

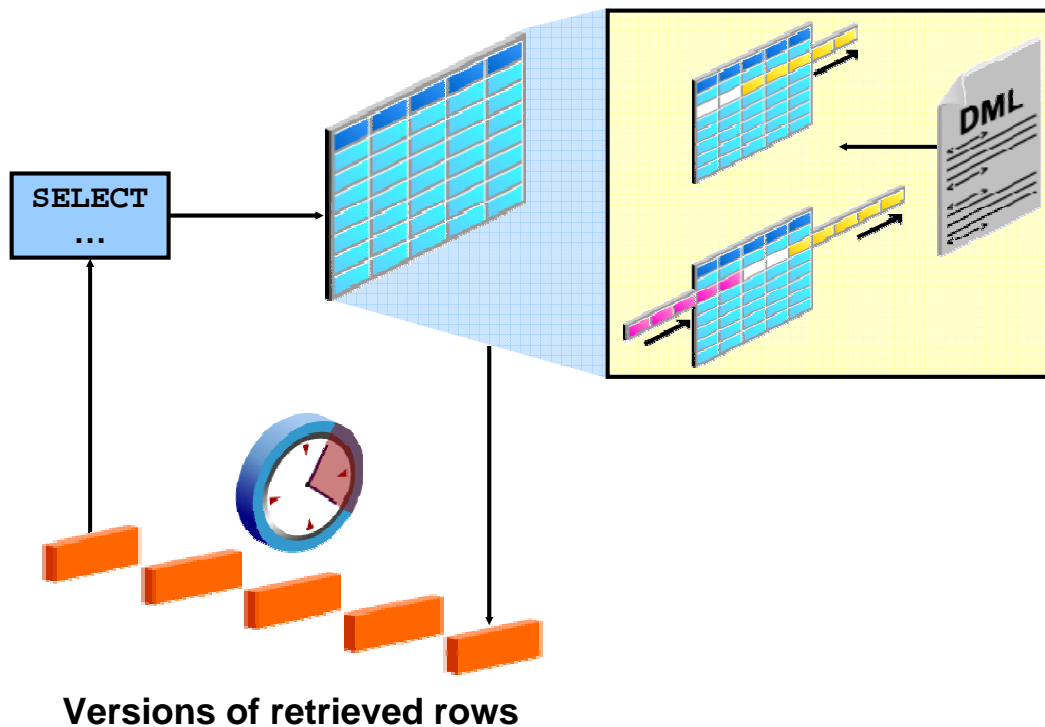
## Example of Merging Rows (continued)

The example on the slide matches the `EMPLOYEE_ID` in the `EMPL3` table to the `EMPLOYEE_ID` in the `EMPLOYEES` table. If a match is found, the row in the `EMPL3` table is updated to match the row in the `EMPLOYEES` table. If the row is not found, it is inserted into the `EMPL3` table.

The condition `c.employee_id = e.employee_id` is evaluated. Because the `EMPL3` table is empty, the condition returns false—there are no matches. The logic falls into the `WHEN NOT MATCHED` clause, and the `MERGE` command inserts the rows of the `EMPLOYEES` table into the `EMPL3` table.

If rows existed in the `EMPL3` table and employee IDs matched in both tables (the `EMPL3` and `EMPLOYEES` tables), then the existing rows in the `EMPL3` table would be updated to match the `EMPLOYEES` table.

# Tracking Changes in Data



ORACLE

3-31

Copyright © 2004, Oracle. All rights reserved.

## Tracking Changes in Data

You may discover that somehow data in a table has been inappropriately changed. To research this, you can use multiple flashback queries to view row data at specific points in time. More efficiently, you can use the Flashback Version Query feature to view all changes to a row over a period of time. This feature enables you to append a `VERSIONS` clause to a `SELECT` statement that specifies an SCN or timestamp range between which you want to view changes to row values. The query also can return associated metadata, such as the transaction responsible for the change.

Further, after you identify an erroneous transaction, you can then use the Flashback Transaction Query feature to identify other changes that were done by the transaction. You then have the option of using the Flashback Table feature to restore the table to a state before the changes were made.

You can use a query on a table with a `VERSIONS` clause to produce all the versions of all the rows that exist or ever existed between the time the query was issued and the `undo_retention` seconds before the current time. `undo_retention` is an initialization parameter which is an auto-tuned parameter. A query that includes a `VERSIONS` clause is referred to as a version query. The results of a version query behaves as if the `WHERE` clause were applied to the versions of the rows. The version query returns versions of the rows only across transactions.

**System change number (SCN):** The Oracle server assigns a system change number (SCN) to identify the redo records for each committed transaction.

## Example of the Flashback Version Query

```
SELECT salary FROM employees3  
WHERE employee_id = 107;
```

1

SALARY
4200

```
UPDATE employees3 SET salary = salary * 1.30  
WHERE employee_id = 107;
```

```
COMMIT;
```

2

```
SELECT salary FROM employees3  
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE  
WHERE employee_id = 107;
```

3

SALARY
5460
4200

ORACLE

3-32

Copyright © 2004, Oracle. All rights reserved.

### Example of the Flashback Version Query

In the example on the slide, the salary for employee 107 is retrieved (1). The salary for employee 107 is increased by 30 percent and this change is committed (2). The different versions of salary are displayed (3).

The `VERSIONS` clause does not change the plan of the query. For example, if you run a query on a table that uses the index access method, then the same query on the same table with a `VERSIONS` clause continues to use the index access method. The versions of the rows returned by the version query are versions of the rows across transactions. The `VERSIONS` clause has no effect on the transactional behavior of a query. This means that a query on a table with a `VERSIONS` clause still inherits the query environment of the ongoing transaction.

The default `VERSIONS` clause can be specified as `VERSIONS BETWEEN {SCN|TIMESTAMP} MINVALUE AND MAXVALUE`.

The `VERSIONS` clause is a SQL extension only for queries. You can have DML and DDL operations that use a `VERSIONS` clause within subqueries. The row version query retrieves all the committed versions of the selected rows. Changes made by the current active transaction are not returned. The version query retrieves all incarnations of the rows. This essentially means that versions returned include deleted and subsequent reinserted versions of the rows.

## Example of Obtaining Row Versions

The row access for a version query can be defined in one of the following two categories:

- ROWID-based row access: In case of ROWID-based access, all versions of the specified ROWID are returned irrespective of the row content. This essentially means that all versions of the slot in the block indicated by the ROWID are returned.
- All other row access: For all other row access, all versions of the rows are returned.

## The VERSIONS BETWEEN Clause

```
SELECT versions_starttime "START_DATE",
       versions_endtime   "END_DATE",
       salary
FROM   employees
       VERSIONS BETWEEN SCN MINVALUE
       AND MAXVALUE
WHERE  last_name = 'Lorentz';
```

START_DATE	END_DATE	SALARY
13-FEB-04 11.16.41 AM		5460
	13-FEB-04 11.16.41 AM	4200

ORACLE

3-34

Copyright © 2004, Oracle. All rights reserved.

### The VERSIONS BETWEEN Clause

You can use the VERSIONS BETWEEN clause to retrieve all of the versions of the rows that exist or have ever existed between the time the query was issued and a point back in time.

If the undo retention time is smaller than the lower bound time/SCN of the BETWEEN clause, then the query retrieves versions up to the undo retention time only. The time interval of the BETWEEN clause can be specified as an SCN interval, or a wall clock interval. This time interval is closed at both the lower and the upper bound.

In the example, Lorentz's salary changes are retrieved. The NULL value for the END\_DATE for the first version indicates that this was the existing version at the time of the query. The NULL for the START\_DATE for the last version indicates that this version was created at a time before the undo retention time.



# Summary

**In this lesson, you should have learned how to:**

- **Use DML statements and control transactions**
- **Describe the features of multitable inserts**
- **Use the following types of multitable inserts**
  - **Unconditional INSERT**
  - **Pivoting INSERT**
  - **Conditional ALL INSERT**
  - **Conditional FIRST INSERT**
- **Merge rows in a table**
- **Manipulate data using subqueries**
- **Track the changes to data over a period of time**

**ORACLE**

3-35

Copyright © 2004, Oracle. All rights reserved.

## Summary

In this lesson, you should have learned how to manipulate data in the Oracle database by using subqueries. You also should have learned about multitable INSERT statements, the MERGE statement, and tracking changes in the database.

## Practice 3: Overview

**This practice covers the following topics:**

- **Performing multitable INSERTs**
- **Performing MERGE operations**
- **Tracking row versions**

ORACLE®

3-36

Copyright © 2004, Oracle. All rights reserved.

### **Practice 3: Overview**

In this practice, you add rows to the emp\_data table, update and delete data from the table, and track your transactions.

### Practice 3

1. Run the lab\_03\_01.sql script in the lab folder to create the SAL\_HISTORY table.
2. Display the structure of the SAL\_HISTORY table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
HIRE_DATE		DATE
SALARY		NUMBER(8,2)

3. Run the lab\_03\_03.sql script in the lab folder to create the MGR\_HISTORY table.
4. Display the structure of the MGR\_HISTORY table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
MANAGER_ID		NUMBER(6)
SALARY		NUMBER(8,2)

5. Run the lab\_03\_05.sql script in the lab folder to create the SPECIAL\_SAL table.
6. Display the structure of the SPECIAL\_SAL table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
SALARY		NUMBER(8,2)

7. a. Write a query to do the following:
  - Retrieve the details of the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the EMPLOYEES table.
  - If the salary is more than \$20,000, insert the details of employee ID and salary into the SPECIAL\_SAL table.
  - Insert the details of employee ID, hire date, and salary into the SAL\_HISTORY table.
  - Insert the details of the employee ID, manager ID, and salary into the MGR\_HISTORY table.

### Practice 3 (continued)

- b. Display the records from the SPECIAL\_SAL table.

EMPLOYEE_ID	SALARY
100	24000

- c. Display the records from the SAL\_HISTORY table.

EMPLOYEE_ID	HIRE_DATE	SALARY
101	21-SEP-89	17000
102	13-JAN-93	17000
103	03-JAN-90	9000
104	21-MAY-91	6000
105	25-JUN-97	4800
106	05-FEB-98	4800
107	07-FEB-99	4200
108	17-AUG-94	12000
109	16-AUG-94	9000
110	28-SEP-97	8200
111	30-SEP-97	7700
112	07-MAR-98	7800
113	07-DEC-99	6900

114	07-DEC-94	11000
115	18-MAY-95	3100
116	24-DEC-97	2900
117	24-JUL-97	2800
118	15-NOV-98	2600
119	10-AUG-99	2500
120	18-JUL-96	8000
121	10-APR-97	8200
122	01-MAY-95	7900
123	10-OCT-97	6500
124	16-NOV-99	5800

24 rows selected.

### Practice 3 (continued)

d. Display the records from the MGR\_HISTORY table.

EMPLOYEE_ID	MANAGER_ID	SALARY
101	100	17000
102	100	17000
103	102	9000
104	103	6000
105	103	4800
106	103	4800
107	103	4200
108	101	12000
109	108	9000
110	108	8200
111	108	7700
112	108	7800
113	108	6900
114	100	11000
115	114	3100
116	114	2900
117	114	2800
118	114	2600
119	114	2500
120	100	8000
121	100	8200
122	100	7900
123	100	6500
124	100	5800

24 rows selected.

### Practice 3 (continued)

8. a. Run the lab\_03\_08a.sql script in the lab folder to create the SALES\_SOURCE\_DATA table.
- b. Run the lab\_03\_08b.sql script in the lab folder to insert records into the SALES\_SOURCE\_DATA table.
- c. Display the structure of the SALES\_SOURCE\_DATA table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK_ID		NUMBER(2)
SALES_MON		NUMBER(8,2)
SALES_TUE		NUMBER(8,2)
SALES_WED		NUMBER(8,2)
SALES_THUR		NUMBER(8,2)
SALES_FRI		NUMBER(8,2)

- d. Display the records from the SALES\_SOURCE\_DATA table.

EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
178	6	1750	2200	1500	1500	3000

- e. Run the lab\_03\_08c.sql script in the lab folder to create the SALES\_INFO table.
- f. Display the structure of the SALES\_INFO table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK		NUMBER(2)
SALES		NUMBER(8,2)

### Practice 3 (continued)

- g. Write a query to do the following:  
Retrieve the details of employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the SALES\_SOURCE\_DATA table.  
Build a transformation such that each record retrieved from the SALES\_SOURCE\_DATA table is converted into multiple records for the SALES\_INFO table.  
**Hint:** Use a pivoting INSERT statement.
- h. Display the records from the SALES\_INFO table.

EMPLOYEE_ID	WEEK	SALES
178	6	1750
178	6	2200
178	6	1500
178	6	1500
178	6	3000

9. You have the data of past employees stored in a flat file called `emp.data`. You want to store the names and e-mail IDs of all employees past and present in a table. To do this, first create an external table called `EMP_DATA` using the `emp.dat` source file in the `emp_dir` directory. You can use the script in `lab_03_09.sql` to do this.
10. Next, run the `lab_03_10.sql` script to create the `EMP_HIST` table.
- Increase the size of the e-mail column to 45.
  - Merge the data in the `EMP_DATA` table created in the last lab into the data in the `EMP_HIST` table. Assume that the data in the external `EMP_DATA` table is the most up-to-date. If a row in the `EMP_DATA` table matches the `EMP_HIST` table, update the e-mail column of the `EMP_HIST` table to match the `EMP_DATA` table row. If a row in the `EMP_DATA` table does not match, insert it into the `EMP_HIST` table. Rows are considered matching when the employee's first and last name are identical.
  - Retrieve the rows from `EMP_HIST` after the merge.

### Practice 3 (continued)

FIRST_NAME	LAST_NAME	EMAIL
Steven	King	SKING
Neena	Kochhar	nkochh@pipit.com
Lex	De Haan	LDEHAAN
Alexander	Hunold	AHun@MOORHEN.COM
Bruce	Ernst	BERNST
David	Austin	DAUSTIN
Valli	Pataballa	VPATABAL
Diana	Lorentz	DLORENTZ
Nancy	Greenberg	NGREENBE
Daniel	Faviet	DFAVIET
John	Chen	JCHEN
Ismael	Sciarra	ISCIARRA

...

FIRST_NAME	LAST_NAME	EMAIL
Diana	lorentz	dlor@limpkin.com
Stephen	King	sking@merganser.com
Hema	Voight	Hema.Voight@PHALAROPE.COM
Nancy	greenberg	ngreenb@plover.com

148 rows selected.

11. Create table EMP3 using the lab\_03\_11.sql script. In the EMP3 table change the department for Kochhar to 60 and commit your change. Next, change the department for Kochhar to 50 and commit your change. Track the changes to Kochhar using the Row Versions feature.

START_DATE	END_DATE	DEPARTMENT_ID
13-FEB-04 12.33.56 PM		50
13-FEB-04 12.33.53 PM	13-FEB-04 12.33.56 PM	60
	13-FEB-04 12.33.53 PM	90



# 4

## **Generating Reports by Grouping Related Data**

**ORACLE®**

Copyright © 2004, Oracle. All rights reserved.

# Objectives

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

- **Use the ROLLUP operation to produce subtotal values**
- **Use the CUBE operation to produce cross-tabulation values**
- **Use the GROUPING function to identify the row values created by ROLLUP or CUBE**
- **Use GROUPING SETS to produce a single result set**

ORACLE®

4-2

Copyright © 2004, Oracle. All rights reserved.

## Objectives

In this lesson you learn how to:

- Group data to obtain the following:
  - Subtotal values by using the ROLLUP operator
  - Cross-tabulation values by using the CUBE operator
- Use the GROUPING function to identify the level of aggregation in the result set produced by a ROLLUP or CUBE operator
- Use GROUPING SETS to produce a single result set that is equivalent to a UNION ALL approach

# Review of Group Functions

- **Group functions operate on sets of rows to give one result per group.**

```
SELECT      [column,] group_function(column). . .
FROM        table
[WHERE      condition]
[GROUP BY  group_by_expression]
[ORDER BY   column];
```

- **Example:**

```
SELECT AVG(salary), STDDEV(salary),
       COUNT(commission_pct), MAX(hire_date)
FROM   employees
WHERE  job_id LIKE 'SA%';
```

ORACLE

## Group Functions

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use group functions to return summary information for each group. Group functions can appear in select lists and in ORDER BY and HAVING clauses. The Oracle server applies the group functions to each group of rows and returns a single result row for each group.

**Types of group functions:** Each of the group functions AVG, SUM, MAX, MIN, COUNT, STDDEV, and VARIANCE accept one argument. The functions AVG, SUM, STDDEV, and VARIANCE operate only on numeric values. MAX and MIN can operate on numeric, character, or date data values. COUNT returns the number of non-null rows for the given expression. The example on the slide calculates the average salary, standard deviation on the salary, number of employees earning a commission, and the maximum hire date for those employees whose JOB\_ID begins with SA.

### Guidelines for Using Group Functions

- The data types for the arguments can be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions except COUNT ( \* ) ignore null values. To substitute a value for null values, use the NVL function. COUNT returns either a number or zero.
- The Oracle server implicitly sorts the result set in ascending order of the grouping columns specified, when you use a GROUP BY clause. To override this default ordering, you can use DESC in an ORDER BY clause.

# Review of the GROUP BY Clause

- **Syntax:**

```
SELECT      [column,] group_function(column). . .
FROM        table
[WHERE      condition]
[GROUP BY   group_by_expression]
[ORDER BY   column];
```

- **Example:**

```
SELECT  department_id, job_id, SUM(salary),
        COUNT(employee_id)
FROM    employees
GROUP BY department_id, job_id ;
```

ORACLE

4-4

Copyright © 2004, Oracle. All rights reserved.

## Review of GROUP BY Clause

The example illustrated on the slide is evaluated by the Oracle server as follows:

- The **SELECT** clause specifies that the following columns are to be retrieved:
  - Department ID and job ID columns from the **EMPLOYEES** table
  - The sum of all the salaries and the number of employees in each group that you have specified in the **GROUP BY** clause
- The **GROUP BY** clause specifies how the rows should be grouped in the table. The total salary and the number of employees are calculated for each job ID within each department. The rows are grouped by department ID and then grouped by job within each department.

## Review of the HAVING Clause

- Use the HAVING clause to specify which groups are to be displayed.
- You further restrict the groups on the basis of a limiting condition.

```
SELECT      [column,] group_function(column)...  
FROM        table  
[WHERE      condition]  
[GROUP BY   group_by_expression]  
[HAVING     having_expression]  
[ORDER BY   column];
```

ORACLE®

4-5

Copyright © 2004, Oracle. All rights reserved.

### The HAVING Clause

Groups are formed and group functions are calculated before the HAVING clause is applied to the groups. The HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because it is more logical.

The Oracle server performs the following steps when you use the HAVING clause:

1. Groups rows
2. Applies the group functions to the groups and displays the groups that match the criteria in the HAVING clause

## GROUP BY with ROLLUP and CUBE Operators

- Use ROLLUP or CUBE with GROUP BY to produce superaggregate rows by cross-referencing columns.
- ROLLUP grouping produces a result set containing the regular grouped rows and the subtotal values.
- CUBE grouping produces a result set containing the rows from ROLLUP and cross-tabulation rows.

ORACLE

4-6

Copyright © 2004, Oracle. All rights reserved.

### GROUP BY with the ROLLUP and CUBE Operators

You specify ROLLUP and CUBE operators in the GROUP BY clause of a query. ROLLUP grouping produces a result set containing the regular grouped rows and subtotal rows. The CUBE operation in the GROUP BY clause groups the selected rows based on the values of all possible combinations of expressions in the specification and returns a single row of summary information for each group. You can use the CUBE operator to produce cross-tabulation rows.

**Note:** When working with ROLLUP and CUBE, make sure that the columns following the GROUP BY clause have meaningful, real-life relationships with each other; otherwise the operators return irrelevant information.

## ROLLUP Operator

- **ROLLUP is an extension to the GROUP BY clause.**
- **Use the ROLLUP operation to produce cumulative aggregates, such as subtotals.**

```
SELECT      [column,] group_function(column). . .
FROM        table
[WHERE      condition]
[GROUP BY   [ROLLUP] group_by_expression]
[HAVING     having_expression];
[ORDER BY   column];
```

ORACLE®

4-7

Copyright © 2004, Oracle. All rights reserved.

### The ROLLUP Operator

The ROLLUP operator delivers aggregates and superaggregates for expressions within a GROUP BY statement. The ROLLUP operator can be used by report writers to extract statistics and summary information from result sets. The cumulative aggregates can be used in reports, charts, and graphs.

The ROLLUP operator creates groupings by moving in one direction, from right to left, along the list of columns specified in the GROUP BY clause. It then applies the aggregate function to these groupings.

#### Note

- To produce subtotals in  $n$  dimensions (that is,  $n$  columns in the GROUP BY clause) without a ROLLUP operator,  $n+1$  SELECT statements must be linked with UNION ALL. This makes the query execution inefficient, because each of the SELECT statements causes table access. The ROLLUP operator gathers its results with just one table access. The ROLLUP operator is useful when there are many columns involved in producing the subtotals.
- Subtotals and totals are produced with ROLLUP. CUBE produces totals as well but effectively rolls up in each possible direction, producing cross-tabular data.

## ROLLUP Operator: Example

```
SELECT  department_id, job_id, SUM(salary)
FROM    employees
WHERE   department_id < 60
GROUP BY ROLLUP(department_id, job_id);
```

DEPARTMENT ID	JOB ID	SUM(SALARY)	
10	AD_ASST	4400	1
10		4400	
20	MK_MAN	13000	2
20	MK_REP	6000	
20		19000	
30	PU_MAN	11000	
30	PU_CLERK	13900	
30		24900	3
40	HR_REP	6500	
40		6500	2
50	ST_MAN	36400	
50	SH_CLERK	64300	
50	ST_CLERK	55700	
50		156400	3
		211200	

15 rows selected.

ORACLE

4-8

Copyright © 2004, Oracle. All rights reserved.

### Example of a ROLLUP Operator

In the example on the slide:

- Total salaries for every job ID within a department for those departments whose department ID is less than 60 are displayed by the GROUP BY clause.
- The ROLLUP operator displays:
  - Total salary for each department whose department ID is less than 60
  - Total salary for all departments whose department ID is less than 60, irrespective of the job IDs

In this example, 1 indicates a group totaled by both DEPARTMENT\_ID and JOB\_ID, 2 indicates a group totaled only by DEPARTMENT\_ID, and 3 indicates the grand total.

The ROLLUP operator creates subtotals that roll up from the most detailed level to a grand total, following the grouping list specified in the GROUP BY clause. First, it calculates the standard aggregate values for the groups specified in the GROUP BY clause (in the example, the sum of salaries grouped on each job within a department). Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. (In the example, the sum of salaries for each department is calculated, followed by the sum of salaries for all departments.)

- Given  $n$  expressions in the ROLLUP operator of the GROUP BY clause, the operation results in  $n + 1$  (in this case  $2 + 1 = 3$ ) groupings.
- Rows based on the values of the first  $n$  expressions are called rows or regular rows and the others are called superaggregate rows.



# CUBE Operator

- CUBE is an extension to the GROUP BY clause.
- You can use the CUBE operator to produce cross-tabulation values with a single SELECT statement.

```
SELECT      [column,] group_function(column)...  
FROM        table  
[WHERE      condition]  
[GROUP BY   [CUBE] group_by_expression]  
[HAVING     having_expression]  
[ORDER BY   column];
```

ORACLE®

4-9

Copyright © 2004, Oracle. All rights reserved.

## The CUBE Operator

The CUBE operator is an additional switch in the GROUP BY clause in a SELECT statement. The CUBE operator can be applied to all aggregate functions, including AVG, SUM, MAX, MIN, and COUNT. It is used to produce result sets that are typically used for cross-tabular reports. Whereas ROLLUP produces only a fraction of possible subtotal combinations, CUBE produces subtotals for all possible combinations of groupings specified in the GROUP BY clause, and a grand total.

The CUBE operator is used with an aggregate function to generate additional rows in a result set. Columns included in the GROUP BY clause are cross-referenced to produce a superset of groups. The aggregate function specified in the select list is applied to these groups to produce summary values for the additional superaggregate rows. The number of extra groups in the result set is determined by the number of columns included in the GROUP BY clause.

In fact, every possible combination of the columns or expressions in the GROUP BY clause is used to produce superaggregates. If you have  $n$  columns or expressions in the GROUP BY clause, there will be  $2^n$  possible superaggregate combinations. Mathematically, these combinations form an  $n$ -dimensional cube, which is how the operator got its name.

By using application or programming tools, these superaggregate values can then be fed into charts and graphs that convey results and relationships visually and effectively.

## CUBE Operator: Example

```
SELECT  department_id, job_id, SUM(salary)
FROM    employees
WHERE   department_id < 60
GROUP BY CUBE (department_id, job_id) ;
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
		211200
	HR_REP	6500
	MK_MAN	13000
	MK_REP	6000
	PU_MAN	11000
	ST_MAN	36400
	AD_ASST	4400
	PU_CLERK	13900
	SH_CLERK	64300
	ST_CLERK	55700
10		4400
10	AD_ASST	4400
20		19000
20	MK_MAN	13000
20	MK_REP	6000
30		24900
30	PU_MAN	11000

1

2

3

4

ORACLE

4-10

Copyright © 2004, Oracle. All rights reserved.

### Example of a CUBE Operator

The output of the SELECT statement in the example can be interpreted as follows:

- The total salary for every job within a department (for those departments whose department ID is less than 60) is displayed by the GROUP BY clause.
- The total salary for those departments whose department ID is less than 60.
- The total salary for every job irrespective of the department.
- Total salary for those departments whose department ID is less than 60, irrespective of the job titles.

In this example, 1 indicates the grand total. 2 indicates the rows totaled by JOB\_ID alone. 3 indicates some of the rows totaled by DEPARTMENT\_ID and JOB\_ID. 4 indicates some of the rows totaled by DEPARTMENT\_ID alone.

The CUBE operator has also performed the ROLLUP operation to display the subtotals for those departments whose department ID is less than 60 and the total salary for those departments whose department ID is less than 60, irrespective of the job titles. Additionally, the CUBE operator displays the total salary for every job irrespective of the department.

**Note:** Similar to the ROLLUP operator, producing subtotals in  $n$  dimensions (that is,  $n$  columns in the GROUP BY clause) without a CUBE operator requires that  $2^n$  SELECT statements be linked with UNION ALL. Thus, a report with three dimensions requires  $2^3 = 8$  SELECT statements to be linked with UNION ALL.

# GROUPING Function

## The GROUPING function:

- Is used with either the CUBE or ROLLUP operator
- Is used to find the groups forming the subtotal in a row
- Is used to differentiate stored NULL values from NULL values created by ROLLUP or CUBE
- Returns 0 or 1

```
SELECT      [column,] group_function(column) .. ,  
            GROUPING(expr)  
FROM        table  
[WHERE      condition]  
[GROUP BY  [ROLLUP][CUBE] group_by_expression]  
[HAVING    having_expression]  
[ORDER BY  column];
```

ORACLE

4-11

Copyright © 2004, Oracle. All rights reserved.

## The GROUPING Function

The GROUPING function can be used with either the CUBE or ROLLUP operator to help you understand how a summary value has been obtained.

The GROUPING function uses a single column as its argument. The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 0 or 1.

The values returned by the GROUPING function are useful to:

- Determine the level of aggregation of a given subtotal; that is, the group or groups on which the subtotal is based
- Identify whether a NULL value in the expression column of a row of the result set indicates:
  - A NULL value from the base table (stored NULL value)
  - A NULL value created by ROLLUP or CUBE (as a result of a group function on that expression)

A value of 0 returned by the GROUPING function based on an expression indicates one of the following:

- The expression has been used to calculate the aggregate value.
- The NULL value in the expression column is a stored NULL value.

A value of 1 returned by the GROUPING function based on an expression indicates one of the following:

- The expression has not been used to calculate the aggregate value.
- The NULL value in the expression column is created by ROLLUP or CUBE as a result of grouping.

## GROUPING Function: Example

```
SELECT  department_id DEPTID, job_id JOB,
        SUM(salary),
        GROUPING(department_id) GRP_DEPT,
        GROUPING(job_id) GRP_JOB
FROM    employees
WHERE   department_id < 50
GROUP BY ROLLUP(department_id, job_id);
```

DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
10	AD_ASST	4400	0	0
10		4400	0	1
20	MK_MAN	13000	0	0
20	MK_REP	6000	0	0
20		19000	0	1
30	PU_MAN	11000	0	0
30	PU_CLERK	13900	0	0
30		24900	0	1
40	HR_REP	6500	0	0
40		6500	0	1
		54800	1	1

11 rows selected.

ORACLE

### Example of a GROUPING Function

In the example on the slide, consider the summary value 4400 in the first row (labeled 1). This summary value is the total salary for the job ID of AD\_ASST within department 10. To calculate this summary value, both the DEPARTMENT\_ID and JOB\_ID columns have been taken into account. Thus, a value of 0 is returned for both the GROUPING(department\_id) and GROUPING(job\_id) expressions.

Consider the summary value 4400 in the second row (labeled 2). This value is the total salary for department 10 and has been calculated by taking into account the DEPARTMENT\_ID column; thus, a value of 0 has been returned by GROUPING(department\_id). Because the JOB\_ID column has not been taken into account to calculate this value, a value of 1 has been returned for GROUPING(job\_id). You can observe similar output in the fifth row.

In the last row, consider the summary value 54800 (labeled 3). This is the total salary for those departments whose department ID is less than 50 and all job titles. To calculate this summary value, neither of the DEPARTMENT\_ID and JOB\_ID columns have been taken into account. Thus a value of 1 is returned for both the GROUPING(department\_id) and GROUPING(job\_id) expressions.

## GROUPING SETS

- **GROUPING SETS syntax is used to define multiple groupings in the same query.**
- **All groupings specified in the GROUPING SETS clause are computed and the results of individual groupings are combined with a UNION ALL operation.**
- **Grouping set efficiency:**
  - Only one pass over the base table is required.
  - There is no need to write complex UNION statements.
  - The more elements GROUPING SETS has, the greater the performance benefit.

ORACLE

4-13

Copyright © 2004, Oracle. All rights reserved.

### GROUPING SETS

GROUPING SETS is a further extension of the GROUP BY clause that you can use to specify multiple groupings of data. Doing so facilitates efficient aggregation and, therefore, facilitates analysis of data across multiple dimensions.

A single SELECT statement can now be written using GROUPING SETS to specify various groupings (which can also include ROLLUP or CUBE operators), rather than multiple SELECT statements combined by UNION ALL operators. For example:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY
GROUPING SETS
((department_id, job_id, manager_id),
 (department_id, manager_id), (job_id, manager_id));
```

This statement calculates aggregates over three groupings:

```
(department_id, job_id, manager_id), (department_id,
manager_id) and (job_id, manager_id)
```

Without this feature, multiple queries combined together with UNION ALL are required to obtain the output of the preceding SELECT statement. A multiquery approach is inefficient, because it requires multiple scans of the same data.

## GROUPING SETS (continued)

Compare the previous example with the following alternative:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY CUBE(department_id, job_id, manager_id);
```

This statement computes all the 8 (2 \*2 \*2) groupings, though only the groups (department\_id, job\_id, manager\_id), (department\_id, manager\_id), and (job\_id, manager\_id) are of interest to you.

Another alternative is the following statement:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, NULL, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, manager_id
UNION ALL
SELECT NULL, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY job_id, manager_id;
```

This statement requires three scans of the base table, which makes it inefficient.

CUBE and ROLLUP can be thought of as grouping sets with very specific semantics. The following equivalencies show this fact:

CUBE(a, b, c) is equivalent to	GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ( ))
ROLLUP(a, b, c) is equivalent to	GROUPING SETS ((a, b, c), (a, b), (a), ( ))

## GROUPING SETS: Example

```
SELECT  department_id, job_id,
        manager_id, avg(salary)
FROM    employees
GROUP BY GROUPING SETS
        ((department_id, job_id), (job_id, manager_id));
```

DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)	
	AD_VP	100	17000	1
	AC_MGR	101	12000	
	FI_MGR	101	12000	
	HR_REP	101	6500	
	MK_MAN	100	13000	
	MK_REP	201	6000	
...	PR_REP	101	10000	
DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)	
100	FI_MGR		12000	2
100	FI_ACCOUNT		7920	
110	AC_MGR		12000	
110	AC_ACCOUNT		8300	

ORACLE

4-15

Copyright © 2004, Oracle. All rights reserved.

### GROUPING SETS: Example

The query on the slide calculates aggregates over two groupings. The table is divided into the following groups:

- Job ID, Manager ID
- Department ID, Job ID

The average salaries for each of these groups are calculated. The result set displays the average salary for each of the two groups.

In the output, the group marked as 1 can be interpreted as:

- The average salary of all employees with the job ID AD\_VP under manager 100 is 17000.
- The average salary of all employees with the job ID AC\_MGR under manager 101 is 12000, and so on.

The group marked as 2 in the output is interpreted as:

- The average salary of all employees with the job ID FI\_MGR in department 100 is 12000.
- The average salary of all employees with the job ID FI\_ACCOUNT in department 100 is 7920, and so on.

### **GROUPING SETS: Example (continued)**

The example on the slide can also be written as:

```
SELECT department_id, job_id, NULL as manager_id,  
       AVG(salary) as AVGSAL  
FROM employees  
GROUP BY department_id, job_id  
UNION ALL  
SELECT NULL, job_id, manager_id, avg(salary) as AVGSAL  
FROM employees  
GROUP BY job_id, manager_id;
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need two scans of the base table, EMPLOYEES. This could be very inefficient. Therefore, the usage of the GROUPING SETS statement is recommended.



# Composite Columns

- **A composite column is a collection of columns that are treated as a unit.**  
`ROLLUP (a, (b, c), d)`
- **Use parentheses within the GROUP BY clause to group columns, so that they are treated as a unit while computing ROLLUP or CUBE operations.**
- **When used with ROLLUP or CUBE, composite columns would require skipping aggregation across certain levels.**

ORACLE

4-17

Copyright © 2004, Oracle. All rights reserved.

## Composite Columns

A composite column is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement:

```
ROLLUP (a, (b, c), d)
```

Here, (b, c) forms a composite column and is treated as a unit. In general, composite columns are useful in ROLLUP, CUBE, and GROUPING SETS. For example, in CUBE or ROLLUP, composite columns would require skipping aggregation across certain levels.

That is, GROUP BY ROLLUP(a, (b, c)) is equivalent to

```
GROUP BY a, b, c UNION ALL  
GROUP BY a UNION ALL  
GROUP BY ( )
```

Here, (b, c) is treated as a unit and ROLLUP is not applied across (b, c). It is as if you have an alias, for example z, for (b, c), and the GROUP BY expression reduces to GROUP BY ROLLUP(a, z).

**Note:** GROUP BY ( ) is typically a SELECT statement with NULL values for the columns a and b and only the aggregate function. This is generally used for generating grand totals.

```
SELECT NULL, NULL, aggregate_col  
FROM <table_name>  
GROUP BY ( );
```

## Composite Columns (continued)

Compare this with the normal ROLLUP as in:

```
GROUP BY ROLLUP(a, b, c)
```

which would be

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Similarly,

```
GROUP BY CUBE((a, b), c)
```

would be equivalent to

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP BY ()
```

The following table shows grouping sets specification and the equivalent GROUP BY specification.

GROUPING SETS Statements	Equivalent GROUP BY Statements
GROUP BY GROUPING SETS(a, b, c)	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c
GROUP BY GROUPING SETS(a, b, (b, c)) (The GROUPING SETS expression has a composite column.)	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c
GROUP BY GROUPING SETS((a, b, c))	GROUP BY a, b, c
GROUP BY GROUPING SETS(a, (b), ())	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()
GROUP BY GROUPING SETS (a, ROLLUP(b, c)) (The GROUPING SETS expression has a composite column.)	GROUP BY a UNION ALL GROUP BY ROLLUP(b, c)

## Composite Columns: Example

```
SELECT    department_id, job_id, manager_id,
          SUM(salary)
FROM      employees
GROUP BY ROLLUP( department_id,(job_id, manager_id));
```

	DEPARTMENT ID	JOB ID	MANAGER ID	SUM(SALARY)	
1		SA_REP	149	7000	
				7000	
	10	AD_ASST	101	4400	
	10			4400	
	20	MK_MAN	100	13000	2
	20	MK_REP	201	6000	
	20			19000	
	...				
	100	FI_MGR	101	12000	
	100	FI_ACCOUNT	108	39600	
	100			51600	3
	110	AC_MGR	101	12000	
	110	AC_ACCOUNT	205	8300	
	110			20300	
				691400	4

45 rows selected.

ORACLE

4-19

Copyright © 2004, Oracle. All rights reserved.

## Composite Columns: Example

Consider the example:

```
SELECT department_id, job_id, manager_id, SUM(salary)
FROM   employees
GROUP BY ROLLUP( department_id, job_id, manager_id);
```

This query results in the Oracle server computing the following groupings:

1. (job\_id, manager\_id)
2. (department\_id, job\_id, manager\_id)
3. (department\_id)
4. Grand total

If you are only interested in specific groups, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating JOB\_ID and MANAGER\_ID columns as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing ROLLUP and CUBE. This is illustrated in the example on the slide. By enclosing the JOB\_ID and MANAGER\_ID columns in parentheses, you indicate to the Oracle server to treat JOB\_ID and MANAGER\_ID as a single unit, that is a composite column.

## Composite Columns: Example (continued)

The example on the slide computes the following groupings:

- (department\_id, job\_id, manager\_id)
- (department\_id)
- ( )

The example on the slide displays the following:

- Total salary for every job , and manager (labeled 1)
- Total salary for every department, job , and manager (labeled 2)
- Total salary for every department (labeled 3)
- Grand total (labeled 4)

The example on the slide can also be written as:

```
SELECT  department_id, job_id, manager_id, SUM(salary)
FROM    employees
GROUP   BY department_id, job_id, manager_id
UNION   ALL
SELECT  department_id, TO_CHAR(NULL), TO_NUMBER(NULL), SUM(salary)
FROM    employees
GROUP   BY department_id
UNION   ALL
SELECT  TO_NUMBER(NULL), TO_CHAR(NULL), TO_NUMBER(NULL), SUM(salary)
FROM    employees
GROUP   BY ( );
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need three scans of the base table, EMPLOYEES. This could be very inefficient. Therefore, the use of composite columns is recommended.

## Concatenated Groupings

- **Concatenated groupings offer a concise way to generate useful combinations of groupings.**
- **To specify concatenated grouping sets, you separate multiple grouping sets, ROLLUP, and CUBE operations with commas so that the Oracle server combines them into a single GROUP BY clause.**
- **The result is a cross-product of groupings from each grouping set.**

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

ORACLE

4-21

Copyright © 2004, Oracle. All rights reserved.

### Concatenated Columns

Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified by listing multiple grouping sets, cubes, and rollups, and separating them with commas. The following is an example of concatenated grouping sets:

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

This SQL example defines the following groupings:

```
(a, c), (a, d), (b, c), (b, d)
```

Concatenation of grouping sets is very helpful for these reasons:

- **Ease of query development:** You need not manually enumerate all groupings.
- **Use by applications:** SQL generated by OLAP applications often involves concatenation of grouping sets, with each grouping set defining groupings needed for a dimension.

## Concatenated Groupings: Example

```
SELECT  department_id, job_id, manager_id,
        SUM(salary)
FROM    employees
GROUP BY department_id,
        ROLLUP(job_id),
        CUBE(manager_id);
```

1		DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
			SA_REP	149	7000
		10	AD_ASST	101	4400
		20	MK_MAN	100	13000
		20	MK_REP	201	6000
2		...			
		90	AD_VP	100	34000
		90	AD PRES		24000
				149	7000
		...			7000
3			SA_REP		7000
		10	AD_ASST		4400
		...			
4		110		101	12000
		110		205	8300
		110			20300

93 rows selected.

ORACLE

4-22

Copyright © 2004, Oracle. All rights reserved.

### Concatenated Groupings: Example

The example on the slide results in the following groupings:

- (job\_id, manager\_id) (1)
- (department\_id, job\_id, manager\_id) (2)
- (job\_id)(3)
- (department\_id, manager\_id)(4)
- (department\_id) (5)

The total salary for each of these groups is calculated.

# Summary

**In this lesson, you should have learned how to use the:**

- **ROLLUP operation to produce subtotal values**
- **CUBE operation to produce cross-tabulation values**
- **GROUPING function to identify the row values created by ROLLUP or CUBE**
- **GROUPING SETS syntax to define multiple groupings in the same query**
- **GROUP BY clause to combine expressions in various ways:**
  - **Composite columns**
  - **Concatenated grouping sets**

**ORACLE**

4-23

Copyright © 2004, Oracle. All rights reserved.

## Summary

- ROLLUP and CUBE are extensions of the GROUP BY clause.
- ROLLUP is used to display subtotal and grand total values.
- CUBE is used to display cross-tabulation values.
- The GROUPING function enables you to determine whether a row is an aggregate produced by a CUBE or ROLLUP operator.
- With the GROUPING SETS syntax, you can define multiple groupings in the same query. GROUP BY computes all the groupings specified and combines them with UNION ALL.
- Within the GROUP BY clause, you can combine expressions in various ways:
  - To specify composite columns, you group columns within parentheses so that the Oracle server treats them as a unit while computing ROLLUP or CUBE operations.
  - To specify concatenated grouping sets, you separate multiple grouping sets, ROLLUP, and CUBE operations with commas so that the Oracle server combines them into a single GROUP BY clause. The result is a cross-product of groupings from each grouping set.

## Practice 4: Overview

**This practice covers using:**

- **ROLLUP operators**
- **CUBE operators**
- **GROUPING functions**
- **GROUPING SETS**

**ORACLE**

4-24

Copyright © 2004, Oracle. All rights reserved.

### **Practice 4: Overview**

In this practice, you use the ROLLUP and CUBE operators as extensions of the GROUP BY clause. You will also use GROUPING SETS.



## Practice 4

1. Write a query to display the following for those employees whose manager ID is less than 120:
  - Manager ID
  - Job ID and total salary for every job ID for employees who report to the same manager
  - Total salary of those managers
  - Total salary of those managers, irrespective of the job IDs

MANAGER_ID	JOB_ID	SUM(SALARY)
100	AD_VP	34000
100	MK_MAN	13000
100	PU_MAN	11000
100	SA_MAN	61000
100	ST_MAN	36400
100		155400
101	AC_MGR	12000
101	FI_MGR	12000
101	HR_REP	6500
101	PR_REP	10000
...	AD_ASST	4400
...	AD_ASST	4400
101		44900
102	IT_PROG	9000
102		9000
103	IT_PROG	19800
103		19800
108	FI_ACCOUNT	39600
108		39600
114	PU_CLERK	13900
114		13900
		282600

21 rows selected.

### Practice 4 (continued)

2. Observe the output from question 1. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the ROLLUP operation.

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
100	AD_VP	34000	0	0
100	MK_MAN	13000	0	0
100	PU_MAN	11000	0	0
100	SA_MAN	61000	0	0
100	ST_MAN	36400	0	0
100		155400	0	1
...				
102	IT_PROG	9000	0	0
102		9000	0	1
103	IT_PROG	19800	0	0
103		19800	0	1
108	FI_ACCOUNT	39600	0	0
108		39600	0	1
114	PU_CLERK	13900	0	0
114		13900	0	1
		282600	1	1

21 rows selected.

### Practice 4 (continued)

3. Write a query to display the following for those employees whose manager ID is less than 120:
- Manager ID
  - Job and total salaries for every job for employees who report to the same manager
  - Total salary of those managers
  - Cross-tabulation values to display the total salary for every job, irrespective of the manager
  - Total salary irrespective of all job titles

MANAGER_ID	JOB_ID	SUM(SALARY)
		282600
	AD_VP	34000
	AC_MGR	12000
	FI_MGR	12000
	HR_REP	6500
...	MK_MAN	13000

MANAGER_ID	JOB_ID	SUM(SALARY)
101	PR_REP	10000
101	AD_ASST	4400
102		9000
102	IT_PROG	9000
103		19800
103	IT_PROG	19800
108		39600
108	FI_ACCOUNT	39600
114		13900
114	PU_CLERK	13900

34 rows selected.

## Practice 4 (continued)

- Observe the output from question 3. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the CUBE operation.

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
		282600	1	1
	AD_VP	34000	1	0
	AC_MGR	12000	1	0
	FI_MGR	12000	1	0
	HR_REP	6500	1	0
	MK_MAN	13000	1	0
	PR_REP	10000	1	0
	PU_MAN	11000	1	0
	SA_MAN	61000	1	0
	ST_MAN	36400	1	0
	AD_ASST	4400	1	0
	IT_PROG	28800	1	0
	PU_CLERK	13900	1	0
	FI_ACCOUNT	39600	1	0
100		155400	0	1

...

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
101	PR_REP	10000	0	0
101	AD_ASST	4400	0	0
102		9000	0	1
102	IT_PROG	9000	0	0
103		19800	0	1
103	IT_PROG	19800	0	0
108		39600	0	1
108	FI_ACCOUNT	39600	0	0
114		13900	0	1
114	PU_CLERK	13900	0	0

34 rows selected.

## Practice 4 (continued)

5. Using GROUPING SETS, write a query to display the following groupings:
- department\_id, manager\_id, job\_id
  - department\_id, job\_id
  - manager\_id, job\_id

The query should calculate the sum of the salaries for each of these groups.

DEPARTMENT_ID	MANAGER_ID	JOB_ID	SUM(SALARY)
90		AD_PRES	24000
90	100	AD_VP	34000
20	100	MK_MAN	13000
30	100	PU_MAN	11000
80	100	SA_MAN	61000
50	100	ST_MAN	36400
110	101	AC_MGR	12000
100	101	FI_MGR	12000
...		AD_PRES	24000
	100	AD_VP	34000
	100	MK_MAN	13000
	100	PU_MAN	11000
...			
		SA_REP	7000
10		AD_ASST	4400
20		MK_MAN	13000
20		MK_REP	6000
...			
50		ST_MAN	36400
50		SH_CLERK	64300
50		ST_CLERK	55700
60		IT_PROG	28800
70		PR_REP	10000
80		SA_MAN	61000
80		SA_REP	243500
90		AD_VP	34000
90		AD_PRES	24000
100		FI_MGR	12000
100		FI_ACCOUNT	39600
110		AC_MGR	12000
110		AC_ACCOUNT	8300

85 rows selected.



# 5

## Managing Data in Different Time Zones

ORACLE®

Copyright © 2004, Oracle. All rights reserved.

# Objectives

**After completing this lesson, you should be able to use the following datetime functions:**

- `TZ_OFFSET`
- `FROM_TZ`
- `TO_TIMESTAMP`
- `TO_TIMESTAMP_TZ`
- `TO_YMINTERVAL`
- `TO_DSINTERVAL`
- `CURRENT_DATE`
- `CURRENT_TIMESTAMP`
- `LOCALTIMESTAMP`
- `DBTIMEZONE`
- `SESSIONTIMEZONE`
- `EXTRACT`

**ORACLE**

5-2

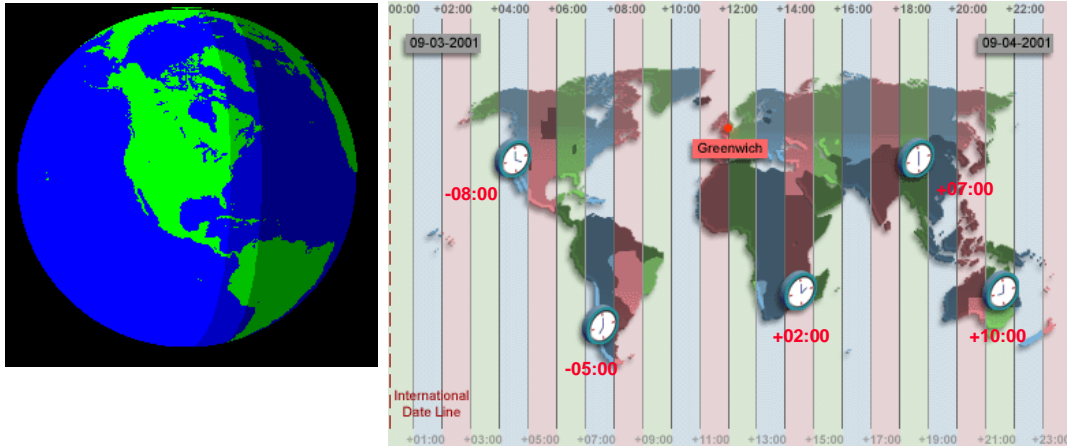
Copyright © 2004, Oracle. All rights reserved.

## Objectives

This lesson addresses some of the datetime functions available in the Oracle database.



# Time Zones



**The image represents the time for each time zone when Greenwich time is 12:00.**

ORACLE®

5-3

Copyright © 2004, Oracle. All rights reserved.

## Time Zones

The hours of the day are measured by the turning of the earth. The time of day at any particular moment depends on where you are. When it is noon in Greenwich, England, it is midnight along the international date line. The earth is divided into 24 time zones, one for each hour of the day. The time along the prime meridian in Greenwich, England, is known as Greenwich Mean Time, or GMT. GMT is the time standard against which all other time zones in the world are referenced. It is the same all year round and is not affected by summer time or daylight saving time. The meridian line is an imaginary line that runs from the North Pole to the South Pole. It is known as zero longitude and it is the line from which all other lines of longitude are measured. All time is measured relative to GMT and all places have a latitude (their distance north or south of the equator) and a longitude (their distance east or west of the Greenwich meridian).

## TIME\_ZONE Session Parameter

**TIME\_ZONE** may be set to:

- **An absolute offset**
- **Database time zone**
- **OS local 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

5-4

Copyright © 2004, Oracle. All rights reserved.

### TIME\_ZONE Session Parameter

The Oracle database supports storing the time zone in your date and time data, as well as fractional seconds. 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, a database time zone, or the local time zone.

## **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**

5-5

Copyright © 2004, Oracle. All rights reserved.

### **CURRENT\_DATE, CURRENT\_TIMESTAMP, and LOCALTIMESTAMP**

The **CURRENT\_DATE** and **CURRENT\_TIMESTAMP** functions 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 values returned display the time zone displacement of the SQL session executing the functions. The time zone displacement is the difference (in hours and minutes) between local time and UTC. The **TIMESTAMP WITH TIME ZONE** data type has the format:

**TIMESTAMP [ (fractional\_seconds\_precision) ] WITH TIME ZONE**

where **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.

The **LOCALTIMESTAMP** function returns the current date and time in the session time zone. The difference between **LOCALTIMESTAMP** and **CURRENT\_TIMESTAMP** is that **LOCALTIMESTAMP** returns a **TIMESTAMP** value, whereas **CURRENT\_TIMESTAMP** returns a **TIMESTAMP WITH TIME ZONE** value.

These functions are NLS sensitive, that is, the results will be in the current NLS calendar and datetime formats.

## CURRENT\_DATE

**Display the current date and time in the session's time zone.**

```
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;
```

SESSIONTIMEZONE	CURRENT_DATE
-05:00	03-OCT-2001 09:37:06

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_DATE
-08:00	03-OCT-2001 06:38:07

ORACLE

5-6

Copyright © 2004, Oracle. All rights reserved.

### CURRENT\_DATE

The `CURRENT_DATE` function returns the current date in the session's time zone. The return value is a date in the Gregorian calendar.

The examples on the slide illustrate that `CURRENT_DATE` is sensitive to the session time zone. In the first example, the session is altered to set the `TIME_ZONE` parameter to `-5:0`. The `TIME_ZONE` parameter specifies the default local time zone displacement for the current SQL session. `TIME_ZONE` is a session parameter only, not an initialization parameter. The `TIME_ZONE` parameter is set as follows:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The format mask (`[+ | -] hh:mm`) indicates the hours and minutes before or after UTC (Coordinated Universal Time, formerly known as Greenwich Mean Time).

Observe in the output that the value of `CURRENT_DATE` changes when the `TIME_ZONE` parameter value is changed to `-8:0` in the second example.

**Note:** The `ALTER SESSION` command sets the date format of the session to `'DD-MON-YYYY HH24:MI:SS'` that is day of month (1-31)-abbreviated name of month-4-digit year hour of day (0-23):minute (0-59):second (0-59).

## CURRENT\_TIMESTAMP

**Display the current date and fractional time in the session's time zone.**

```
ALTER SESSION SET TIME_ZONE = '-5:0';  
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP  
FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_TIMESTAMP
-05:00	03-OCT-01 09.40.59.000000 AM -05:00

```
ALTER SESSION SET TIME_ZONE = '-8:0';  
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP  
FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_TIMESTAMP
-08:00	03-OCT-01 06.41.38.000000 AM -08:00

ORACLE

5-7

Copyright © 2004, Oracle. All rights reserved.

### CURRENT\_TIMESTAMP

The `CURRENT_TIMESTAMP` function returns the current date and time in the session time zone, as a value of the data type `TIMESTAMP WITH TIME ZONE`. The time zone displacement reflects the current local time of the SQL session. The syntax of the `CURRENT_TIMESTAMP` function is:

`CURRENT_TIMESTAMP` (*precision*)

where *precision* is an optional argument that specifies the fractional second precision of the time value returned. If you omit precision, the default is 6.

The examples on the slide illustrate that `CURRENT_TIMESTAMP` is sensitive to the session time zone. In the first example, the session is altered to set the `TIME_ZONE` parameter to `-5:0`. Observe in the output that the value of `CURRENT_TIMESTAMP` changes when the `TIME_ZONE` parameter value is changed to `-8:0` in the second example.

## LOCALTIMESTAMP

- Display the current date and time in the session's time zone in a value of **TIMESTAMP** data type.

```
ALTER SESSION SET TIME_ZONE = '-5:0';  
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP  
FROM DUAL;
```

CURRENT_TIMESTAMP	LOCALTIMESTAMP
03-OCT-01 09.44.21.000000 AM -05:00	03-OCT-01 09.44.21.000000 AM

```
ALTER SESSION SET TIME_ZONE = '-8:0';  
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP  
FROM DUAL;
```

CURRENT_TIMESTAMP	LOCALTIMESTAMP
03-OCT-01 06.45.21.000001 AM -08:00	03-OCT-01 06.45.21.000001 AM

- **LOCALTIMESTAMP returns a TIMESTAMP value, whereas CURRENT\_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value.**

ORACLE

5-8

Copyright © 2004, Oracle. All rights reserved.

### LOCALTIMESTAMP

The **LOCALTIMESTAMP** function returns the current date and time in the session time zone. **LOCALTIMESTAMP** returns a **TIMESTAMP** value. The syntax of the **LOCAL\_TIMESTAMP** function is:

**LOCAL\_TIMESTAMP** (**TIMESTAMP\_precision**)

Where, **TIMESTAMP\_precision** is an optional argument that specifies the fractional second precision of the **TIMESTAMP** value returned.

The examples on the slide illustrate the difference between **LOCALTIMESTAMP** and **CURRENT\_TIMESTAMP**. Observe that the **LOCALTIMESTAMP** does not display the time zone value, whereas the **CURRENT\_TIMESTAMP** does.

## DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone.

```
SELECT DBTIMEZONE FROM DUAL;
```

DBTIME
-05:00

- Display the value of the session's time zone.

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

SESSIONTIMEZONE
-08:00

ORACLE

5-9

Copyright © 2004, Oracle. All rights reserved.

### DBTIMEZONE and SESSIONTIMEZONE

The DBA sets the database's default time zone by specifying the `SET TIME_ZONE` clause of the `CREATE DATABASE` statement. If omitted, the default database time zone is the operating system time zone. The database time zone cannot be changed for a session with an `ALTER SESSION` statement.

The `DBTIMEZONE` function returns the value of the database time zone. The return type is a time zone offset (a character type in the format '`[+ | -] TZh:TzM`') 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. The example on the slide shows that the database time zone is set to `-05:00`, as the `TIME_ZONE` parameter is in the format:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The `SESSIONTIMEZONE` function returns the value of the current session's time zone. The return type is a time zone offset (a character type in the format '`[+ | -] TZh:TzM`') or a time zone region name, depending on how the user specified the session time zone value in the most recent `ALTER SESSION` statement. The example on the slide shows that the session time zone is offset to UTC by `-8` hours. Observe that the database time zone is different from the current session's time zone.

## 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

5-10

Copyright © 2004, Oracle. All rights reserved.

### Datetime Data Types

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

The **TIMESTAMP WITH TIME ZONE** data type contains the datetime fields **YEAR**, **MONTH**, **DAY**, **HOURL**, **MINUTE**, **SECOND**, **TIMEZONE\_HOUR**, and **TIMEZONE\_MINUTE** and fractional seconds.

The **TIMESTAMP WITH LOCAL TIME ZONE** data type contains same information as the **TIMESTAMP** data type, except that the data is normalized to the database time zone when stored, and adjusted to match the client's time zone when retrieved.

**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.



## TIMESTAMP Data Types

Data Type	Fields
<b>TIMESTAMP</b>	Year, Month, Day, Hour, Minute, Second with fractional seconds
<b>TIMESTAMP WITH TIME ZONE</b>	Same as the <b>TIMESTAMP</b> data type; also includes: TimeZone_Hour, and TimeZone_Minute or TimeZone_Region
<b>TIMESTAMP WITH LOCAL TIME ZONE</b>	Same as the <b>TIMESTAMP</b> data type; also includes a a time zone offset in its value

ORACLE

5-11

Copyright © 2004, Oracle. All rights reserved.

### TIMESTAMP 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. The 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.

#### **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.

## **TIMESTAMP Fields**

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

**ORACLE**

5-12

Copyright © 2004, Oracle. All rights reserved.

### **TIMESTAMP 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.

## Difference between DATE and TIMESTAMP

**A**

```
-- when hire_date is  
of type DATE  
  
SELECT hire_date  
FROM emp5;
```

HIRE_DATE
17-JUN-87
21-SEP-89
13-JAN-93
03-JAN-90
21-MAY-91
25-JUN-97
05-FEB-98
07-FEB-99
17-AUG-94
16-AUG-94
28-SEP-97

...

**B**

```
ALTER TABLE emp5  
MODIFY hire_date TIMESTAMP;  
  
SELECT hire_date  
FROM emp5;
```

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
25-JUN-97 12.00.00.000000 AM
05-FEB-98 12.00.00.000000 AM
07-FEB-99 12.00.00.000000 AM
17-AUG-94 12.00.00.000000 AM
16-AUG-94 12.00.00.000000 AM
28-SEP-97 12.00.00.000000 AM
30-SEP-97 12.00.00.000000 AM

...

ORACLE

5-13

Copyright © 2004, Oracle. All rights reserved.

### TIMESTAMP Data Type: Example

On the slide, example A shows the data from the `hire_date` column of the `EMP5` 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. You can convert from `DATE` to `TIMESTAMP` when the column has data, but you cannot convert from `DATE` or `TIMESTAMP` to `TIMESTAMP WITH TIME ZONE` unless the column is empty. You can 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 emp5  
MODIFY hire_date TIMESTAMP(7);
```

**Note:** The Oracle date data type by default appears as shown in this example. However, the date data type also contains additional information such as hours, minutes, seconds, a.m., and p.m. To obtain the date in this format, you can apply a format mask or a function to the date value.

## 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

5-14

Copyright © 2004, Oracle. 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'
```

## TIMESTAMP WITH TIMEZONE: Example

```
CREATE TABLE web_orders  
(ord_id number primary key,  
 order_date TIMESTAMP WITH TIME ZONE);
```

```
INSERT INTO web_orders values  
(ord_seq.nextval, current_date);
```

```
SELECT * FROM web_orders;
```

ORD_ID	ORDER_DATE
100	09-FEB-04 07.04.44.000000 AM -07:00

ORACLE

5-15

Copyright © 2004, Oracle. All rights reserved.

### TIMESTAMP WITH TIME ZONE: Example

In the example on the slide, a new table `web_orders` is created with a column of data type `TIMESTAMP WITH TIME ZONE`. This table is populated whenever a `web_order` is placed. The timestamp and time zone for the user placing the order is inserted based on the `CURRENT_DATE` value. That way when a Web-based company guarantees shipping, they can estimate their delivery time based on the time zone of the person placing the order.

## 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.**
- **The Oracle database returns the data in the user's 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 TIMEZONE: Example

```
CREATE TABLE shipping (delivery_time TIMESTAMP WITH  
LOCAL TIME ZONE);  
INSERT INTO shipping VALUES(current_timestamp + 2);
```

```
SELECT * FROM shipping;
```

DELIVERY\_TIME

11-FEB-04 07.09.02.000000 AM

```
ALTER SESSION SET TIME_ZONE = 'EUROPE/LONDON';  
SELECT * FROM shipping;
```

DELIVERY\_TIME

11-FEB-04 02.09.02.000000 PM

ORACLE

5-17

Copyright © 2004, Oracle. All rights reserved.

### TIMESTAMP WITH LOCAL TIME ZONE: Example

In the example on the slide, a new table SHIPPING is created with a column of the data type TIMESTAMP WITH LOCAL TIME ZONE. This table is populated by inserting two days from the CURRENT\_TIMESTAMP value into it every time an order is placed. The output from the DATE\_TAB table shows that the data is stored without the time zone offset. Then the ALTER SESSION command is issued to change the time zone to the local time zone at the place of delivery. A second query on the same table now reflects the data with the local time zone reflected in the time value, so that the customer can be notified about the expected delivery time.

## 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 constitutes an interval
  - Specified in the interval qualifier

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

ORACLE

5-18

Copyright © 2004, Oracle. 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 of YEAR and MONTH, whereas a day-time interval is made up of a contiguous subset of fields consisting of DAY, HOUR, MINUTE, and SECOND. The actual subset of fields that constitute 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 Data Types (continued)**

### **INTERVAL YEAR (year\_precision) TO MONTH**

This data type stores a period of time in years and months, where `year_precision` is the number of digits in the YEAR datetime field. The accepted values are 0 to 9. The default is 6.

### **INTERVAL DAY (day\_precision) TO SECOND (fractional\_seconds\_precision)**

This data type 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. The 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

5-20

Copyright © 2004, Oracle. 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 constitute 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®

5-21

Copyright © 2004, Oracle. 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.

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
(prod_id number, warranty_time INTERVAL YEAR(3)
TO MONTH);

INSERT INTO warranty VALUES (123, INTERVAL '8'
MONTH);

INSERT INTO warranty VALUES (155, INTERVAL '200'
YEAR(3));

INSERT INTO warranty VALUES (678, '200-11');

SELECT * FROM warranty;
```

PROD_ID	WARRANTY_TIME
123	+000-08
155	+200-00
678	+200-11

ORACLE

5-22

Copyright © 2004, Oracle. All rights reserved.

### INTERVAL YEAR TO MONTH Data Type (continued)

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 Oracle database supports two interval data types: Interval Year to Month and Interval Day to Second; the 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 such as 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 INTERVAL YEAR(3) TO MONTH data type. Different values are inserted into it to indicate years and months for various products. When these rows are retrieved from the table, you see a year value displaced by the month value by a (-).

## 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
( exp_id number, test_time INTERVAL DAY(2) TO
SECOND);

INSERT INTO lab VALUES (100012, '90 00:00:00');
INSERT INTO lab VALUES (56098,
INTERVAL '6 03:30:16' DAY TO SECOND);
```

```
SELECT * FROM lab;
```

EXP_ID	TEST_TIME
100012	+90 00:00:00.000000
56098	+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 statement shows how this data is displayed in the database.

## EXTRACT

- Display the YEAR component from the SYSDATE.

```
SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;
```

EXTRACT(YEARFROMSYSDATE)

2001

- Display the MONTH component from the HIRE\_DATE for those employees whose MANAGER\_ID is 100.

```
SELECT last_name, hire_date,  
       EXTRACT (MONTH FROM HIRE_DATE)  
FROM employees  
WHERE manager_id = 100;
```

LAST_NAME	HIRE_DATE	EXTRACT(MONTHFROMHIRE_DATE)
Kochhar	21-SEP-89	9
De Haan	13-JAN-93	1
Mourgos	16-NOV-99	11
Zlotkey	29-JAN-00	1
Hartstein	17-FEB-96	2

ORACLE

5-25

Copyright © 2004, Oracle. All rights reserved.

### EXTRACT

The EXTRACT expression extracts and returns the value of a specified datetime field from a datetime or interval value expression. You can extract any of the components mentioned in the following syntax using the EXTRACT function. The syntax of the EXTRACT function is:

```
SELECT EXTRACT ([YEAR] [MONTH][DAY] [HOUR] [MINUTE][SECOND]  
               [TIMEZONE_HOUR] [TIMEZONE_MINUTE]  
               [TIMEZONE_REGION] [TIMEZONE_ABBR]  
FROM [datetime_value_expression] [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 a date in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC.

In the first example on the slide, the EXTRACT function is used to extract the YEAR from SYSDATE. In the second example on the slide, the EXTRACT function is used to extract the MONTH from HIRE\_DATE column of the EMPLOYEES table, for those employees who report to the manager whose EMPLOYEE\_ID is 100.

## TZ\_OFFSET

- **Display the time zone offset for the time zone 'US/Eastern'.**

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;
```

TZ_OFFSET
-04:00

- **Display the time zone offset for the time zone 'Canada/Yukon'.**

```
SELECT TZ_OFFSET('Canada/Yukon') FROM DUAL;
```

TZ_OFFSET
-07:00

- **Display the time zone offset for the time zone 'Europe/London'.**

```
SELECT TZ_OFFSET('Europe/London') FROM DUAL;
```

TZ_OFFSET
+01:00

ORACLE

5-26

Copyright © 2004, Oracle. All rights reserved.

### TZ\_OFFSET

The TZ\_OFFSET function returns the time zone offset corresponding to the value entered. The return value is dependent on the date when the statement is executed. For example, if the TZ\_OFFSET function returns a value -08:00, this value indicates that the time zone where the command was executed is eight hours behind UTC. You can enter a valid time zone name, a time zone offset from UTC (which simply returns itself), or the keyword SESSIONTIMEZONE or DBTIMEZONE. The syntax of the TZ\_OFFSET function is:

```
TZ_OFFSET ( [ 'time_zone_name' ] '[+ | -] hh:mm' |  
           [ SESSIONTIMEZONE ] [ DBTIMEZONE ] )
```

The Fold Motor Company has a headquarters in Michigan, USA, which is in US/Eastern time zone. The company president, Mr. Fold, wants to conduct a conference call with the vice president of the Canadian operations and the vice president of European operations, who are in the Canada/Yukon and Europe/London time zones, respectively. Mr. Fold wants to find out the time in each of these places to make sure that his senior management will be available to attend the meeting. His secretary, Mr. Scott, helps by issuing the queries shown in the example and gets the following results:

- The time zone 'US/Eastern' is four hours behind UTC.
- The time zone 'Canada/Yukon' is seven hours behind UTC.
- The time zone 'Europe/London' is one hour ahead of UTC.



## TZ\_OFFSET (continued)

For a listing of valid time zone name values, you can query the V\$TIMEZONE\_NAMES dynamic performance view.

```
SELECT * FROM V$TIMEZONE_NAMES;
```

TZNAME	TZABBREV
Africa/Algiers	LMT
Africa/Algiers	PMT
Africa/Algiers	WET
Africa/Algiers	WEST
Africa/Algiers	CET
Africa/Algiers	CEST
Africa/Cairo	LMT
Africa/Cairo	EET
Africa/Cairo	EEST
Africa/Casablanca	LMT
Africa/Casablanca	WET
Africa/Casablanca	WEST
Africa/Casablanca	CET
Africa/Ceuta	LMT
Africa/Ceuta	WET
Africa/Ceuta	WEST

■ ■ ■

## TIMESTAMP Conversion Using FROM\_TZ

- **Display the TIMESTAMP value '2000-03-28 08:00:00' as a TIMESTAMP WITH TIME ZONE value.**

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

```
FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
28-MAR-00 08.00.00.000000000 AM +03:00
```

- **Display the TIMESTAMP value '2000-03-28 08:00:00' as a TIMESTAMP WITH TIME ZONE value for the time zone region 'Australia/North'.**

```
SELECT FROM_TZ(TIMESTAMP
                '2000-03-28 08:00:00', 'Australia/North')
FROM DUAL;
```

```
FROM_TZ(TIMESTAMP'2000-03-2808:00:00','AUSTRALIA/NORTH')
28-MAR-00 08.00.00.000000000 AM AUSTRALIA/NORTH
```

ORACLE

5-28

Copyright © 2004, Oracle. All rights reserved.

### TIMESTAMP Conversion Using FROM\_TZ

The FROM\_TZ function converts a TIMESTAMP value to a TIMESTAMP WITH TIME ZONE value.

The syntax of the FROM\_TZ function is as follows:

```
FROM_TZ(TIMESTAMP timestamp_value, time_zone_value)
```

where time\_zone\_value is a character string in the format 'TZH:TZM' or a character expression that returns a string in TZR (time zone region) with optional TZD format. TZD is an abbreviated time zone string with daylight saving information. TZR represents the time zone region in datetime input strings. Examples are 'Australia/North', 'PST' for US/Pacific standard time and 'PDT' for US/Pacific daylight time and so on. To see a listing of valid values for the TZR and TZD format elements, query the V\$TIMEZONE\_NAMES dynamic performance view.

The example on the slide converts a TIMESTAMP value to TIMESTAMP WITH TIME ZONE.

## Converting to TIMESTAMP Using TO\_TIMESTAMP and TO\_TIMESTAMP\_TZ

- Display the character string '2000-12-01 11:00:00' as a TIMESTAMP value.

```
SELECT TO_TIMESTAMP ('2000-12-01 11:00:00',  
                    'YYYY-MM-DD HH:MI:SS')  
FROM DUAL;
```

```
TO_TIMESTAMP('2000-12-0111:00:00','YYYY-MM-DDHH:MI:SS')  
01-DEC-00 11.00.00.000000000 AM
```

- Display the character string '1999-12-01 11:00:00 -8:00' as a TIMESTAMP WITH TIME ZONE value.

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

```
TO_TIMESTAMP_TZ('1999-12-0111:00:00-8:00','YYYY-MM-DDHH:MI:SSTZH:TZM')  
01-DEC-99 11.00.00.000000000 AM -08:00
```

ORACLE

5-29

Copyright © 2004, Oracle. All rights reserved.

### Converting to TIMESTAMP Using TO\_TIMESTAMP and TO\_TIMESTAMP\_TZ

The TO\_TIMESTAMP function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of TIMESTAMP data type. The syntax of the TO\_TIMESTAMP function is:

```
TO_TIMESTAMP (char,[fmt],[ 'nlsparam' ])
```

The optional fmt specifies the format of char if omitted, the string must be in the default format of the TIMESTAMP data type. The optional nlsparam specifies the language in which month and day names, and abbreviations are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit nlsparams, this function uses the default date language for your session. The example on the slide converts a character string to a value of TIMESTAMP.

The TO\_TIMESTAMP\_TZ function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of TIMESTAMP WITH TIME ZONE data type. The syntax of the TO\_TIMESTAMP\_TZ function is:

```
TO_TIMESTAMP_TZ (char,[fmt],[ 'nlsparam' ])
```

The optional fmt specifies the format of char. If omitted, a string must be in the default format of the TIMESTAMP WITH TIME ZONE data type. The example on the slide converts a character string to a value of TIMESTAMP WITH TIME ZONE.

## Time Interval Conversion with TO\_YMINTERVAL

**Display a date that is one year, two months after the hire date for the employees working in the department with the DEPARTMENT\_ID 20.**

```
SELECT hire_date,  
       hire_date + TO_YMINTERVAL('01-02') AS  
       HIRE_DATE_YMININTERVAL  
FROM   employees  
WHERE  department_id = 20;
```

HIRE_DATE	HIRE_DATE_YMININTERV
17-FEB-1996 00:00:00	17-APR-1997 00:00:00
17-AUG-1997 00:00:00	17-OCT-1998 00:00:00

ORACLE®

5-30

Copyright © 2004, Oracle. All rights reserved.

### Time Interval Conversion with TO\_YMINTERVAL

The TO\_YMINTERVAL function converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL YEAR TO MONTH data type. The INTERVAL YEAR TO MONTH data type stores a period of time using the YEAR and MONTH datetime fields. The format of INTERVAL YEAR TO MONTH is 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.

The syntax of the TO\_YMINTERVAL function is:

TO\_YMINTERVAL (char)

where char is the character string to be converted.

The example on the slide calculates a date that is one year and two months after the hire date for the employees working in the department 20 of the EMPLOYEES table.

A reverse calculation can also be done using the TO\_YMINTERVAL function. For example:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('-02-04') AS  
       HIRE_DATE_YMININTERVAL  
FROM   EMPLOYEES WHERE department_id = 20;
```

Observe that the character string passed to the TO\_YMINTERVAL function has a negative value. The example returns a date that is two years and four months before the hire date for the employees working in the department 20 of the EMPLOYEES table.

# 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
Hunold	01-03-90:12:00:00	04-13-90:10:00:00
Ernst	05-21-91:12:00:00	08-29-91:10:00:00
Austin	06-25-97:12:00:00	10-03-97:10:00:00
Pataballa	02-05-98:12:00:00	05-16-98:10:00:00
Lorentz	02-07-99:12:00:00	05-18-99:10:00:00
Greenberg	08-17-94:12:00:00	11-25-94:10:00:00
Faviet	08-16-94:12:00:00	11-24-94:10:00:00

...

ORACLE

5-31

Copyright © 2004, Oracle. 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
```

...

# Daylight Saving Time

- **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®

5-32

Copyright © 2004, Oracle. All rights reserved.

## Daylight Saving Time (DST)

Most western nations advance the clock ahead one hour during the summer months. This period is called daylight saving time. Daylight saving time lasts from the first Sunday in April to the last Sunday in October in the most of the United States, Mexico, and Canada. The nations of the European Union observe daylight saving time, but they call it the summer time period. Europe's summer time period begins a week earlier than its North American counterpart, but ends at the same time.

The Oracle database automatically determines, for any given time zone region, whether daylight saving time is in effect and returns local time values accordingly. The datetime value is sufficient for the Oracle database to determine whether daylight saving time is in effect for a given region in all cases except boundary cases. A boundary case occurs during the period when daylight saving time goes into or out of effect. For example, in the US-Eastern region, when daylight saving time 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 saving time 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.

## Daylight Saving Time (DST) (continued)

### **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 occurs in the overlapped period and no time zone abbreviation was specified to distinguish the period.

For example, if daylight saving time ends on October 31, at 02:00:01 a.m. The overlapped periods were:

- 10/31/2004 01:00:01 a.m. to 10/31/2004 02:00:00 a.m. (EDT)
- 10/31/2004 01:00:01 a.m. to 10/31/2004 02:00:00 a.m. (EST)

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

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

# Summary

**In this lesson, you should have learned how to use the following functions:**

- `TZ_OFFSET`
- `FROM_TZ`
- `TO_TIMESTAMP`
- `TO_TIMESTAMP_TZ`
- `TO_YMINTERVAL`
- `CURRENT_DATE`
- `CURRENT_TIMESTAMP`
- `LOCALTIMESTAMP`
- `DBTIMEZONE`
- `SESSIONTIMEZONE`
- `EXTRACT`

**ORACLE**

5-34

Copyright © 2004, Oracle. All rights reserved.

## Summary

This lesson addressed some of the datetime functions available in the Oracle database.



## Practice 5: Overview

**This practice covers using the datetime functions.**

ORACLE

5-35

Copyright © 2004, Oracle. All rights reserved.

### Practice 5: Overview

In this practice, you display time zone offsets, `CURRENT_DATE`, `CURRENT_TIMESTAMP`, and the `LOCALTIMESTAMP`. You also set time zones and use the `EXTRACT` function.

## Practice 5

1. Alter the session to set the NLS\_DATE\_FORMAT to DD-MON-YYYY HH24:MI:SS.
2.
  - a. Write queries to display the time zone offsets (TZ\_OFFSET), for the following time zones.
    - *US/Pacific-New*

TZ_OFFSET('US/PACIFIC')
-08:00

- *Singapore*

TZ_OFFSET('SINGAPORE')
+08:00

- *Egypt*

TZ_OFFSET('EGYPT')
+02:00

- b. Alter the session to set the TIME\_ZONE parameter value to the time zone offset of US/Pacific-New.
- c. Display the CURRENT\_DATE, CURRENT\_TIMESTAMP, and LOCALTIMESTAMP for this session.

CURRENT_DATE	CURRENT_TIMESTAMP	LOCALTIMESTAMP
16-FEB-04	16-FEB-04 05.12.22.557032 PM -07:00	16-FEB-04 05.12.22.557032 PM

- d. Alter the session to set the TIME\_ZONE parameter value to the time zone offset of Singapore.
- e. Display the CURRENT\_DATE, CURRENT\_TIMESTAMP, and LOCALTIMESTAMP for this session.

**Note:** The output might be different, based on the date when the command is executed.

CURRENT_DATE	CURRENT_TIMESTAMP	LOCALTIMESTAMP
17-FEB-04	17-FEB-04 08.06.18.057870 AM +08:00	17-FEB-04 08.06.18.057870 AM

**Note:** Observe in the preceding practice that CURRENT\_DATE, CURRENT\_TIMESTAMP, and LOCALTIMESTAMP are all sensitive to the session time zone.

3. Write a query to display the DBTIMEZONE and SESSIONTIMEZONE.

DBTIMEZONE	SESSIONTIMEZONE
+00:00	-07:00

## Practice 5 (continued)

- Write a query to extract the YEAR from the HIRE\_DATE column of the EMPLOYEES table for those employees who work in department 80.

LAST_NAME	EXTRACT(YEARFROMHIRE_DATE)
Russell	1996
Partners	1997
Errazuriz	1997
Cambrault	1999
Zlotkey	2000

...

LAST_NAME	EXTRACT(YEARFROMHIRE_DATE)
Bloom	1998
Fox	1998
Smith	1999
Bates	1999
Kumar	2000
Abel	1996
Hutton	1997
Taylor	1998
Livingston	1998
Johnson	2000

34 rows selected.

- Alter the session to set NLS\_DATE\_FORMAT to DD-MON-YYYY.

## Practice 5 (continued)

6. Examine and run script lab\_05\_06.sql to create the SAMPLE\_DATES table and populate it.

- a. Select from the table and view the data.

DATE_COL
16-FEB-04

- b. Modify the data type of the DATE\_COL column and change it to TIMESTAMP. Select from the table to view the data.

DATE_COL
16-FEB-04 05.38.50.000000 PM

- c. Try to modify the data type of the DATE\_COL column and change it to TIMESTAMP WITH TIME ZONE. What happens?
7. Create a query to retrieve last names from the EMPLOYEES table and calculate review status. If the year hired was 2000, display Needs Review for the review status, otherwise display not this year! Name the review status column Review. Sort the results by the HIRE\_DATE column.

**Hint:** Use a CASE expression with EXTRACT function to calculate the review status.

LAST_NAME	Review
King	not this year!
Kochhar	not this year!
De Haan	not this year!
Hunold	not this year!
Ernst	not this year!
Austin	not this year!
Pataballa	Needs Review
Lorentz	not this year!

...

LAST_NAME	Review
Walsh	Needs Review
Feeney	Needs Review
OConnell	not this year!
Grant	not this year!
Whalen	not this year!
Hartstein	not this year!
Fay	not this year!
Mavris	not this year!
Baer	not this year!
Higgins	not this year!
Gietz	not this year!

107 rows selected.

## Practice 5 (continued)

8. Create a query to print the last names and the number of years of service for each employee. If the employee has been employed five or more years, then print 5 years of service. If the employee has been employed 10 or more years, then print 10 years of service. If the employee has been employed 15 or more years, then print 15 years of service. If none of these conditions match, then print maybe next year! Sort the results by the HIRE\_DATE column. Use EMPLOYEES table.

**Hint:** Use CASE expressions and TO\_YMINTERVAL.

LAST_NAME	HIRE_DATE	SYSDATE	Awards
King	17-JUN-87	16-FEB-04	15 years of service
Kochhar	21-SEP-89	16-FEB-04	10 years of service
De Haan	13-JAN-93	16-FEB-04	10 years of service
Hunold	03-JAN-90	16-FEB-04	10 years of service
Ernst	21-MAY-91	16-FEB-04	10 years of service
Austin	25-JUN-97	16-FEB-04	5 years of service
Pataballa	05-FEB-98	16-FEB-04	5 years of service
Lorentz	07-FEB-99	16-FEB-04	5 years of service

...

LAST_NAME	HIRE_DATE	SYSDATE	Awards
Walsh	24-APR-98	16-FEB-04	5 years of service
Feeney	23-MAY-98	16-FEB-04	5 years of service
OConnell	21-JUN-99	16-FEB-04	maybe next year!
Grant	13-JAN-00	16-FEB-04	maybe next year!
Whalen	17-SEP-87	16-FEB-04	15 years of service
Hartstein	17-FEB-96	16-FEB-04	5 years of service
Fay	17-AUG-97	16-FEB-04	5 years of service
Mavris	07-JUN-94	16-FEB-04	5 years of service
Baer	07-JUN-94	16-FEB-04	5 years of service
Higgins	07-JUN-94	16-FEB-04	5 years of service
Gietz	07-JUN-94	16-FEB-04	5 years of service

107 rows selected.



# 6

## Retrieving Data Using Subqueries

ORACLE®

Copyright © 2004, Oracle. All rights reserved.

# Objectives

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

- **Write a multiple-column subquery**
- **Use scalar subqueries in SQL**
- **Solve problems with correlated subqueries**
- **Update and delete rows using correlated subqueries**
- **Use the EXISTS and NOT EXISTS operators**
- **Use the WITH clause**

**ORACLE**

6-2

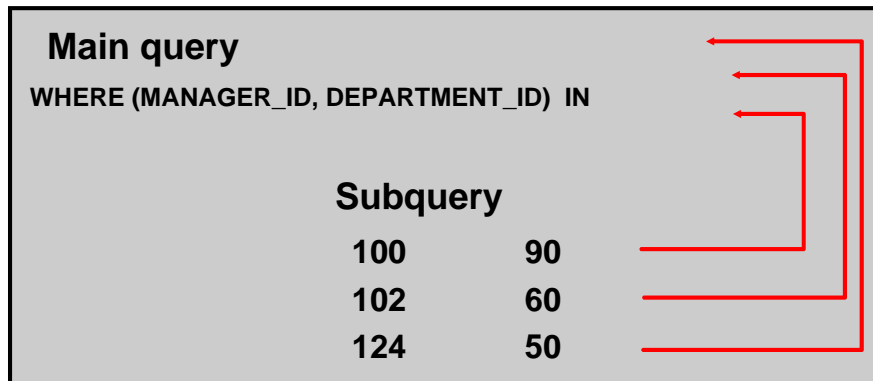
Copyright © 2004, Oracle. All rights reserved.

## Objectives

In this lesson, you learn how to write multiple-column subqueries and subqueries in the FROM clause of a SELECT statement. You also learn how to solve problems by using scalar, correlated subqueries and the WITH clause.



# Multiple-Column Subqueries



**Each row of the main query is compared to values from a multiple-row and multiple-column subquery.**

ORACLE

6-3

Copyright © 2004, Oracle. All rights reserved.

## Multiple-Column Subqueries

So far you have written single-row subqueries and multiple-row subqueries where only one column is returned by the inner `SELECT` statement and this is used to evaluate the expression in the parent select statement. If you want to compare two or more columns, you must write a compound `WHERE` clause using logical operators. Using multiple-column subqueries, you can combine duplicate `WHERE` conditions into a single `WHERE` clause.

### Syntax

```
SELECT column, column, ...
FROM table
WHERE (column, column, ...) IN
      (SELECT column, column, ...
       FROM table
       WHERE condition);
```

The graphic on the slide illustrates that the values of the `MANAGER_ID` and `DEPARTMENT_ID` from the main query are being compared with the `MANAGER_ID` and `DEPARTMENT_ID` values retrieved by the subquery. Because the number of columns that are being compared are more than one, the example qualifies as a multiple-column subquery.

# Column Comparisons

**Column comparisons in a multiple-column subquery can be:**

- **Pairwise comparisons**
- **Nonpairwise comparisons**

ORACLE

6-4

Copyright © 2004, Oracle. All rights reserved.

## Pairwise Versus Nonpairwise Comparisons

Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

In the example on the next slide, a pairwise comparison is executed in the WHERE clause. Each candidate row in the SELECT statement must have *both* the same MANAGER\_ID and the DEPARTMENT\_ID columns as the employees with the EMPLOYEE\_ID 199 or 174.

A multiple-column subquery can also be a nonpairwise comparison. In a nonpairwise comparison, each of the columns from the WHERE clause of the parent SELECT statement is individually compared to multiple values retrieved by the inner SELECT statement. The individual columns can match any of the values retrieved by the inner SELECT statement. But collectively, all the multiple conditions of the main SELECT statement must be satisfied for the row to be displayed. The example on the next page illustrates a pairwise comparison.

## Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager *and* work in the same department as the employees with EMPLOYEE\_ID 199 or 174.

```
SELECT employee_id, manager_id, department_id
FROM   employees
WHERE  (manager_id, department_id) IN
      (SELECT manager_id, department_id
       FROM   employees
       WHERE  employee_id IN (199,174))
AND    employee_id NOT IN (199,174);
```

ORACLE

6-5

Copyright © 2004, Oracle. All rights reserved.

### Pairwise Comparison Subquery

The example on the slide is that of a multiple-column subquery because the subquery returns more than one column. It compares the values in the MANAGER\_ID column and the DEPARTMENT\_ID column of each row in the EMPLOYEES table with the values in the MANAGER\_ID column and the DEPARTMENT\_ID column for the employees with the EMPLOYEE\_ID 199 or 174.

First, the subquery to retrieve the MANAGER\_ID and DEPARTMENT\_ID values for the employees with the EMPLOYEE\_ID 199 or 174 is executed. These values are compared with the MANAGER\_ID column and the DEPARTMENT\_ID column of each row in the EMPLOYEES table. If the values match, the row is displayed. In the output, the records of the employees with the EMPLOYEE\_ID 199 or 174 will not be displayed. The following is the output of the query on the slide:

EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
141	124	50
142	124	50

...

11 rows selected.

## Nonpairwise Comparison Subquery

**Display the details of the employees who are managed by the same manager as the employees with EMPLOYEE\_ID 174 or 199 *and* work in the same department as the employees with EMPLOYEE\_ID 174 or 199.**

```
SELECT employee_id, manager_id, department_id
FROM   employees
WHERE  manager_id IN
      (SELECT manager_id
       FROM   employees
       WHERE  employee_id IN (174,199))
AND    department_id IN
      (SELECT department_id
       FROM   employees
       WHERE  employee_id IN (174,199))
AND    employee_id NOT IN(174,199);
```

ORACLE

6-6

Copyright © 2004, Oracle. All rights reserved.

### Nonpairwise Comparison Subquery

The example shows a nonpairwise comparison of the columns. It displays the EMPLOYEE\_ID, MANAGER\_ID, and DEPARTMENT\_ID of any employee whose manager ID matches any of the manager IDs of employees whose employee IDs are either 174 or 199 and DEPARTMENT\_ID match any of the department IDs of employees whose employee IDs are either 174 or 199.

First, the subquery to retrieve the MANAGER\_ID values for the employees with the EMPLOYEE\_ID 174 or 199 is executed. Similarly, the second subquery to retrieve the DEPARTMENT\_ID values for the employees with the EMPLOYEE\_ID 174 or 199 is executed. The retrieved values of the MANAGER\_ID and DEPARTMENT\_ID columns are compared with the MANAGER\_ID and DEPARTMENT\_ID column for each row in the EMPLOYEES table. If the MANAGER\_ID column of the row in the EMPLOYEES table matches with any of the values of the MANAGER\_ID retrieved by the inner subquery and if the DEPARTMENT\_ID column of the row in the EMPLOYEES table matches with any of the values of the DEPARTMENT\_ID retrieved by the second subquery, the record is displayed. The following is the output of the query on the slide:

EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
198	124	50
...	124	50

11 rows selected.

# Scalar Subquery Expressions

- **A scalar subquery expression is a subquery that returns exactly one column value from one row.**
- **Scalar subqueries can be used in:**
  - **Condition and expression part of `DECODE` and `CASE`**
  - **All clauses of `SELECT` except `GROUP BY`**

ORACLE

6-7

Copyright © 2004, Oracle. All rights reserved.

## Scalar Subqueries in SQL

A subquery that returns exactly one column value from one row is also referred to as a scalar subquery. Multiple-column subqueries that are written to compare two or more columns, using a compound `WHERE` clause and logical operators, do not qualify as scalar subqueries.

The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is `NULL`. If the subquery returns more than one row, the Oracle server returns an error. The Oracle server has always supported the usage of a scalar subquery in a `SELECT` statement. You can use scalar subqueries in:

- The condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- The `SET` clause and `WHERE` clause of an `UPDATE` statement

However, scalar subqueries are not valid expressions in the following places:

- As default values for columns and hash expressions for clusters
- In the `RETURNING` clause of DML statements
- As the basis of a function-based index
- In `GROUP BY` clauses, `CHECK` constraints, `WHEN` conditions
- In `CONNECT BY` clauses
- In statements that are unrelated to queries, such as `CREATE PROFILE`

## Scalar Subqueries: Examples

- **Scalar subqueries in CASE expressions**

```
SELECT employee_id, last_name,  
       (CASE  
         WHEN department_id = 20  
           (SELECT department_id  
            FROM departments  
            WHERE location_id = 1800)  
         THEN 'Canada' ELSE 'USA' END) location  
FROM   employees;
```

- **Scalar subqueries in ORDER BY clause**

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

ORACLE

6-8

Copyright © 2004, Oracle. All rights reserved.

### Scalar Subqueries: Examples

The first example on the slide demonstrates that scalar subqueries can be used in CASE expressions. The inner query returns the value 20, which is the department ID of the department whose location ID is 1800. The CASE expression in the outer query uses the result of the inner query to display the employee ID, last names, and a value of Canada or USA, depending on whether the department ID of the record retrieved by the outer query is 20 or not.

The result of the first example on the slide follows:

...

EMPLOYEE_ID	LAST_NAME	LOCATION
196	Walsh	USA
197	Feeney	USA
198	OConnell	USA
199	Grant	USA
200	Whalen	USA
201	Hartstein	Canada
202	Fay	Canada
203	Mavris	USA
204	Baer	USA
205	Higgins	USA
206	Gietz	USA

107 rows selected.

## Scalar Subqueries: Examples (continued)

The second example on the slide demonstrates that scalar subqueries can be used in the ORDER BY clause. The example orders the output based on the DEPARTMENT\_NAME by matching the DEPARTMENT\_ID from the EMPLOYEES table with the DEPARTMENT\_ID from the DEPARTMENTS table. This comparison is done in a scalar subquery in the ORDER BY clause. The result of the second example follows:

EMPLOYEE_ID	LAST_NAME
205	Higgins
206	Gietz
200	Whalen
100	King
101	Kochhar
102	De Haan
108	Greenberg
109	Faviet

...

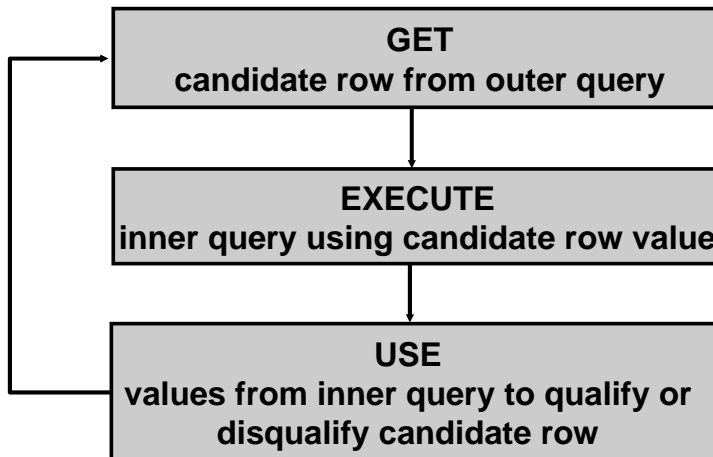
EMPLOYEE_ID	LAST_NAME
135	Gee
136	Philtanker
137	Ladwig
138	Stiles
139	Seo
140	Patel
141	Rajs
142	Davies
143	Matos
144	Vargas
178	Grant

107 rows selected.

The second example uses a correlated subquery. In a correlated subquery, the subquery references a column from a table referred to in the parent statement. Correlated subqueries are explained later in this lesson.

# Correlated Subqueries

**Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.**



ORACLE

6-10

Copyright © 2004, Oracle. All rights reserved.

## Correlated Subqueries

The Oracle server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement.

### Nested Subqueries Versus Correlated Subqueries

With a normal nested subquery, the inner `SELECT` query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

### Nested Subquery Execution

- The inner query executes first and finds a value.
- The outer query executes once, using the value from the inner query.

### Correlated Subquery Execution

- Get a candidate row (fetched by the outer query).
- Execute the inner query using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.



# Correlated Subqueries

The subquery references a column from a table in the parent query.

```
SELECT column1, column2, ...  
FROM   table1 outer  
WHERE  column1 operator  
        (SELECT column1, column2  
         FROM   table2  
         WHERE  expr1 =  
                outer.expr2);
```

ORACLE

6-11

Copyright © 2004, Oracle. All rights reserved.

## Correlated Subqueries (continued)

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.


The Oracle server performs a correlated subquery when the subquery references a column from a table in the parent query.

**Note:** You can use the ANY and ALL operators in a correlated subquery.

# Using Correlated Subqueries

**Find all employees who earn more than the average salary in their department.**

```
SELECT last_name, salary, department_id
FROM   employees outer
WHERE  salary >
      (SELECT AVG(salary)
       FROM   employees
       WHERE  department_id =
             outer.department_id);
```



Each time a row from the outer query is processed, the inner query is evaluated.

ORACLE

6-12

Copyright © 2004, Oracle. All rights reserved.

## Using Correlated Subqueries

The example on the slide determines which employees earn more than the average salary of their department. In this case, the correlated subquery specifically computes the average salary for each department.

Because both the outer query and inner query use the EMPLOYEES table in the FROM clause, an alias is given to EMPLOYEES in the outer SELECT statement, for clarity. Not only does the alias make the entire SELECT statement more readable, but without the alias the query would not work properly, because the inner statement would not be able to distinguish the inner table column from the outer table column.

# Using Correlated Subqueries

**Display details of those employees who have changed jobs at least twice.**

```
SELECT e.employee_id, last_name,e.job_id
FROM   employees e
WHERE  2 <= (SELECT COUNT(*)
              FROM   job_history
              WHERE  employee_id = e.employee_id);
```

EMPLOYEE_ID	LAST_NAME	JOB_ID
101	Kochhar	AD_VP
176	Taylor	SA_REP
200	Whalen	AD_ASST

ORACLE

6-13

Copyright © 2004, Oracle. All rights reserved.

## Using Correlated Subqueries (continued)

The example on the slide displays the details of those employees who have changed jobs at least twice. The Oracle server evaluates a correlated subquery as follows:

1. Select a row from the table specified in the outer query. This will be the current candidate row.
2. Store the value of the column referenced in the subquery from this candidate row. (In the example on the slide, the column referenced in the subquery is E.EMPLOYEE\_ID.)
3. Perform the subquery with its condition referencing the value from the outer query's candidate row. (In the example on the slide, group function COUNT(\*) is evaluated based on the value of the E.EMPLOYEE\_ID column obtained in step 2.)
4. Evaluate the WHERE clause of the outer query on the basis of results of the subquery performed in step 3. This determines whether the candidate row is selected for output. (In the example, the number of times an employee has changed jobs, evaluated by the subquery, is compared with 2 in the WHERE clause of the outer query. If the condition is satisfied, that employee record is displayed.)
5. Repeat the procedure for the next candidate row of the table, and so on until all the rows in the table have been processed.

The correlation is established by using an element from the outer query in the subquery. In this example you compare EMPLOYEE\_ID from the table in the subquery with the EMPLOYEE\_ID from the table in the outer query.

## Using the EXISTS Operator

- The **EXISTS** operator tests for existence of rows in the results set of the subquery.
- If a subquery row value is found:
  - The search does not continue in the inner query
  - The condition is flagged **TRUE**
- If a subquery row value is not found:
  - The condition is flagged **FALSE**
  - The search continues in the inner query

ORACLE

6-14

Copyright © 2004, Oracle. All rights reserved.

### The EXISTS Operator

With nesting **SELECT** statements, all logical operators are valid. In addition, you can use the **EXISTS** operator. This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query. If the subquery returns at least one row, the operator returns **TRUE**. If the value does not exist, it returns **FALSE**. Accordingly, **NOT EXISTS** tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.

## Find Employees Who Have at Least One Person Reporting to Them

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT 'X'
                FROM   employees
                WHERE  manager_id =
                      outer.employee_id);
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
108	Greenberg	FL_MGR	100
114	Raphaely	PU_MAN	30
120	Weiss	ST_MAN	50
121	Fripp	ST_MAN	50
122	Kaufling	ST_MAN	50
123	Vollman	ST_MAN	50
124	Mourgos	ST_MAN	50
145	Russell	SA_MAN	80
146	Partners	SA_MAN	80
147	Errazuriz	SA_MAN	80
148	Cambrault	SA_MAN	80
149	Zlotkey	SA_MAN	80
201	Hartstein	MK_MAN	20
205	Higgins	AC_MGR	110

18 rows selected.

ORACLE®

### Using the EXISTS Operator

The EXISTS operator ensures that the search in the inner query does not continue when at least one match is found for the manager and employee number by the condition:

```
WHERE manager_id = outer.employee_id.
```

Note that the inner SELECT query does not need to return a specific value, so a constant can be selected.

# Find All Departments That Do Not Have Any Employees

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
                  FROM employees
                  WHERE department_id = d.department_id);
```

DEPARTMENT_ID	DEPARTMENT_NAME
120	Treasury
130	Corporate Tax
140	Control And Credit
150	Shareholder Services
160	Benefits
170	Manufacturing
...	
260	Recruiting
270	Payroll

16 rows selected.

ORACLE

6-16

Copyright © 2004, Oracle. All rights reserved.

## Using the NOT EXISTS Operator

### Alternative Solution

A NOT IN construct can be used as an alternative for a NOT EXISTS operator, as shown in the following example:

```
SELECT department_id, department_name
FROM departments
WHERE department_id NOT IN (SELECT department_id
                           FROM employees);
```

No rows selected.

However, NOT IN evaluates to FALSE if any member of the set is a NULL value. Therefore, your query will not return any rows even if there are rows in the departments table that satisfy the WHERE condition.

# Correlated UPDATE

Use a correlated subquery to update rows in one table based on rows from another table.

```
UPDATE table1 alias1
SET    column = (SELECT expression
                     FROM   table2 alias2
                     WHERE  alias1.column =
                           alias2.column);
```

ORACLE®

6-17

Copyright © 2004, Oracle. All rights reserved.

## Correlated UPDATE

In the case of the UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

## Using Correlated UPDATE

- Denormalize the EMPL6 table by adding a column to store the department name.
- Populate the table by using a correlated update.

```
ALTER TABLE empl6  
ADD(department_name VARCHAR2(25));
```

```
UPDATE empl6 e  
SET    department_name =  
        (SELECT department_name  
         FROM   departments d  
         WHERE  e.department_id = d.department_id);
```

ORACLE

6-18

Copyright © 2004, Oracle. All rights reserved.

### Correlated UPDATE (continued)

The example on the slide denormalizes the EMPL6 table by adding a column to store the department name and then populates the table by using a correlated update.

Following is another example for a correlated update.

#### Problem Statement

The REWARDS table has a list of employees who have exceeded expectations in their performance. Use a correlated subquery to update rows in the EMPL6 table based on rows from the REWARDS table:

```
UPDATE empl6  
SET    salary = (SELECT employees.salary + rewards.pay_raise  
                 FROM   rewards  
                 WHERE  employee_id =  
                        employees.employee_id  
                 AND    payraise_date =  
                        (SELECT MAX(payraise_date)  
                         FROM   rewards  
                         WHERE  employee_id = employees.employee_id))  
WHERE  employees.employee_id  
IN      (SELECT employee_id FROM rewards);
```



### **Correlated UPDATE (continued)**

This example uses the REWARDS table. The REWARDS table has the columns EMPLOYEE\_ID, PAY\_RAISE, and PAYRAISE\_DATE. Every time an employee gets a pay raise, a record with the details of the employee ID, the amount of the pay raise, and the date of receipt of the pay raise is inserted into the REWARDS table. The REWARDS table can contain more than one record for an employee. The PAYRAISE \_DATE column is used to identify the most recent pay raise received by an employee.

In the example, the SALARY column in the EMPL6 table is updated to reflect the latest pay raise received by the employee. This is done by adding the current salary of the employee with the corresponding pay raise from the REWARDS table.

# Correlated DELETE

**Use a correlated subquery to delete rows in one table based on rows from another table.**

```
DELETE FROM table1 alias1
WHERE column operator
      (SELECT expression
       FROM table2 alias2
       WHERE alias1.column = alias2.column);
```

ORACLE

6-20

Copyright © 2004, Oracle. All rights reserved.

## Correlated DELETE

In the case of a DELETE statement, you can use a correlated subquery to delete only those rows that also exist in another table. If you decide that you will maintain only the last four job history records in the JOB\_HISTORY table, then when an employee transfers to a fifth job, you delete the oldest JOB\_HISTORY row by looking up the JOB\_HISTORY table for the MIN(START\_DATE) for the employee. The following code illustrates how the preceding operation can be performed using a correlated DELETE:

```
DELETE FROM emp_history JH
WHERE employee_id =
      (SELECT employee_id
       FROM employees E
       WHERE JH.employee_id = E.employee_id
       AND START_DATE =
           (SELECT MIN(start_date)
            FROM job_history JH
            WHERE JH.employee_id = E.employee_id)
       AND 5 > (SELECT COUNT(*)
                FROM job_history JH
                WHERE JH.employee_id = E.employee_id
                GROUP BY EMPLOYEE_ID
                HAVING COUNT(*) >= 4));
```

## Using Correlated DELETE

**Use a correlated subquery to delete only those rows from the EMPL6 table that also exist in the EMP\_HISTORY table.**

```
DELETE FROM empl6 E
WHERE employee_id =
      (SELECT employee_id
       FROM   emp_history
       WHERE  employee_id = E.employee_id);
```

ORACLE

6-21

Copyright © 2004, Oracle. All rights reserved.

### Correlated DELETE (continued)

#### Example

Two tables are used in this example. They are:

- The EMPL6 table, which provides details of all the current employees
- The EMP\_HISTORY table, which provides details of previous employees

EMP\_HISTORY contains data regarding previous employees, so it would be erroneous if the same employee's record existed in both the EMPL6 and EMP\_HISTORY tables. You can delete such erroneous records by using the correlated subquery shown on the slide.

# The WITH Clause

- Using the **WITH** clause, you can use the same query block in a **SELECT** statement when it occurs more than once within a complex query.
- The **WITH** clause retrieves the results of a query block and stores it in the user's temporary tablespace.
- The **WITH** clause improves performance.

ORACLE

6-22

Copyright © 2004, Oracle. All rights reserved.

## The WITH Clause

Using the **WITH** clause, you can define a query block before using it in a query. The **WITH** clause (formally known as `subquery_factoring_clause`) enables you to reuse the same query block in a **SELECT** statement when it occurs more than once within a complex query. This is particularly useful when a query has many references to the same query block and there are joins and aggregations.

Using the **WITH** clause, you can reuse the same query when it is costly to evaluate the query block and it occurs more than once within a complex query. Using the **WITH** clause, the Oracle server retrieves the results of a query block and stores it in the user's temporary tablespace. This can improve performance.

### **WITH** Clause Benefits

- Makes the query easy to read
- Evaluates a clause only once, even if it appears multiple times in the query
- In most cases may improve performance for large queries

## WITH Clause: Example

**Using the `WITH` clause, write a query to display the department name and total salaries for those departments whose total salary is greater than the average salary across departments.**

ORACLE

6-23

Copyright © 2004, Oracle. All rights reserved.

### WITH Clause: Example

The problem on the slide would require the following intermediate calculations:

1. Calculate the total salary for every department, and store the result using a `WITH` clause.
2. Calculate the average salary across departments, and store the result using a `WITH` clause.
3. Compare the total salary calculated in the first step with the average salary calculated in the second step. If the total salary for a particular department is greater than the average salary across departments, then display the department name and the total salary for that department.

The solution for this problem is provided on the next page.

## WITH Clause: Example

```
WITH
dept_costs AS (
    SELECT d.department_name, SUM(e.salary) AS dept_total
    FROM   employees e JOIN departments d
    ON     e.department_id = d.department_id
    GROUP BY d.department_name),
avg_cost AS (
    SELECT SUM(dept_total)/COUNT(*) AS dept_avg
    FROM   dept_costs)
SELECT *
FROM   dept_costs
WHERE  dept_total >
      (SELECT dept_avg
       FROM avg_cost)
ORDER BY department_name;
```

ORACLE

6-24

Copyright © 2004, Oracle. All rights reserved.

### WITH Clause: Example (continued)

The SQL code on the slide is an example of a situation in which you can improve performance and write SQL more simply by using the WITH clause. The query creates the query names DEPT\_COSTS and AVG\_COST and then uses them in the body of the main query. Internally, the WITH clause is resolved either as an in-line view or a temporary table. The optimizer chooses the appropriate resolution depending on the cost or benefit of temporarily storing the results of the WITH clause.

The output generated by the SQL code on the slide is as follows:

DEPARTMENT_NAME	DEPT_TOTAL
Sales	304500
Shipping	156400

### The WITH Clause Usage Notes

- It is used only with SELECT statements.
- 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).
- When the query name is the same as an existing table name, the parser searches from the inside out, and the query block name takes precedence over the table name.
- The WITH clause can hold more than one query. Each query is then separated by a comma.

# Summary

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

- **A multiple-column subquery returns more than one column.**
- **Multiple-column comparisons can be pairwise or nonpairwise.**
- **A multiple-column subquery can also be used in the FROM clause of a SELECT statement.**

ORACLE

6-25

Copyright © 2004, Oracle. All rights reserved.

## Summary

You can use multiple-column subqueries to combine multiple WHERE conditions in a single WHERE clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

You can use a subquery to define a table to be operated on by a containing query.

Scalar subqueries can be used in:

- Condition and expression part of DECODE and CASE
- All clauses of SELECT except GROUP BY
- A SET clause and WHERE clause of UPDATE statement

## Summary

- **Correlated subqueries are useful whenever a subquery must return a different result for each candidate row.**
- **The `EXISTS` operator is a Boolean operator that tests the presence of a value.**
- **Correlated subqueries can be used with `SELECT`, `UPDATE`, and `DELETE` statements.**
- **You can use the `WITH` clause to use the same query block in a `SELECT` statement when it occurs more than once.**

ORACLE

6-26

Copyright © 2004, Oracle. All rights reserved.

### Summary (continued)

The Oracle server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement. Using the `WITH` clause, you can reuse the same query when it is costly to reevaluate the query block and it occurs more than once within a complex query.



## Practice 6: Overview

**This practice covers the following topics:**

- **Creating multiple-column subqueries**
- **Writing correlated subqueries**
- **Using the `EXISTS` operator**
- **Using scalar subqueries**
- **Using the `WITH` clause**

**ORACLE**

6-27

Copyright © 2004, Oracle. All rights reserved.

### **Practice 6: Overview**

In this practice, you write multiple-column subqueries, and correlated and scalar subqueries. You also solve problems by writing the `WITH` clause.

## Practice 6

1. Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

LAST_NAME	DEPARTMENT_ID	SALARY
Russell	80	14000
Partners	80	13500
Errazuriz	80	12000
Abel	80	11000
Cambraut	80	11000

...

2. Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID 1700.

...

LAST_NAME	DEPARTMENT_NAME	SALARY
Matos	Shipping	2600
OConnell	Shipping	2600
Grant	Shipping	2600
Himuro	Purchasing	2600
Vargas	Shipping	2500
Sullivan	Shipping	2500
Perkins	Shipping	2500
Patel	Shipping	2500
Marlow	Shipping	2500
Colmenares	Purchasing	2500
Whalen	Administration	4400
Gietz	Accounting	8300

36 rows selected.

3. Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar.

**Note:** Do not display Kochhar in the result set.

LAST_NAME	HIRE_DATE	SALARY
De Haan	13-JAN-93	17000

4. Create a query to display the employees who earn a salary that is higher than the salary of all of the sales managers (JOB\_ID = 'SA\_MAN'). Sort the results on salary from highest to lowest.

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Kochhar	AD_VP	17000
De Haan	AD_VP	17000

## Practice 6 (continued)

5. Display the details of the employee ID, last name, and department ID of those employees who live in cities whose name begins with T.

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
202	Fay	20
201	Hartstein	20

6. Write a query to find all employees who earn more than the average salary in their departments.  
Display last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

ENAME	SALARY	DEPTNO	DEPT_AVG
Bell	4000	50	3475.55556
Bull	4100	50	3475.55556
Rajs	3500	50	3475.55556
Dilly	3600	50	3475.55556
Weiss	8000	50	3475.55556
Fripp	8200	50	3475.55556
Everett	3900	50	3475.55556
Vollman	6500	50	3475.55556
Kaufling	7900	50	3475.55556
Sarchand	4200	50	3475.55556
Mourgos	5800	50	3475.55556
Ladwig	3600	50	3475.55556
...			
Vishney	10500	80	8955.88235
Russell	14000	80	8955.88235
Tucker	10000	80	8955.88235
McEwen	9000	80	8955.88235
Greene	9500	80	8955.88235
Sully	9500	80	8955.88235
King	10000	80	8955.88235
Bloom	10000	80	8955.88235
Ozer	11500	80	8955.88235
Hall	9000	80	8955.88235
Abel	11000	80	8955.88235
Hartstein	13000	20	9500
Higgins	12000	110	10150
King	24000	90	19333.3333

38 rows selected.

## Practice 6 (continued)

7. Find all employees who are not supervisors.
  - a. First do this using the NOT EXISTS operator.

LAST_NAME
Ernst
Austin
Pataballa
Lorentz
Faviet
OConnell
Grant
Whalen
Fay
Mavris
Baer
Gietz

89 rows selected.

- b. Can this be done by using the NOT IN operator? How, or why not?
8. Write a query to display the last names of the employees who earn less than the average salary in their departments.

LAST_NAME
Fay
Khoo
Baida
Tobias
Himuro
Colmenares
Nayer
Kochhar
De Haan
Chen
Sciarra
Urman
Popp
Gietz

65 rows selected.

## Practice 6 (continued)

- Write a query to display the last names of the employees who have one or more coworkers in their departments with later hire dates but higher salaries.

LAST_NAME
Faviet
Sciarra
Tobias
Bell
Sarchand
Marvins
Tuvault
Grant
Perkins
Gee

66 rows selected.

- Write a query to display the employee ID, last names, and department names of all employees.

**Note:** Use a scalar subquery to retrieve the department name in the SELECT statement.

EMPLOYEE_ID	LAST_NAME	DEPARTMENT
205	Higgins	Accounting
206	Gietz	Accounting
200	Whalen	Administration
100	King	Executive
101	Kochhar	Executive
102	De Haan	Executive
108	Greenberg	Finance
109	Faviet	Finance
110	Chen	Finance
111	Sciarra	Finance
113	Popp	Finance
112	Urman	Finance
203	Mavris	Human Resources
140	Patel	Shipping
141	Rajs	Shipping
142	Davies	Shipping
143	Matos	Shipping
144	Vargas	Shipping
178	Grant	

107 rows selected.

**Practice 6 (continued)**

11. Write a query to display the department names of those departments whose total salary cost is above one-eighth ( $1/8$ ) of the total salary cost of the whole company. Use the `WITH` clause to write this query. Name the query `SUMMARY`.

DEPARTMENT_NAME	DEPT_TOTAL
Sales	304500
Shipping	156400



# Hierarchical Retrieval

ORACLE®

Copyright © 2004, Oracle. All rights reserved.

# Objectives

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

- **Interpret the concept of a hierarchical query**
- **Create a tree-structured report**
- **Format hierarchical data**
- **Exclude branches from the tree structure**

**ORACLE**

7-2

Copyright © 2004, Oracle. All rights reserved.

## Objectives

In this lesson, you learn how to use hierarchical queries to create tree-structured reports.



## Sample Data from the EMPLOYEES Table

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
100	King	AD_PRES	
101	Kochhar	AD_VP	100
102	De Haan	AD_VP	100
103	Hunold	IT_PROG	102
104	Ernst	IT_PROG	103
105	Austin	IT_PROG	103
106	Pataballa	IT_PROG	103
107	Lorentz	IT_PROG	103
108	Greenberg	FI_MGR	101

...

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
196	Walsh	SH_CLERK	124
197	Feeney	SH_CLERK	124
198	OConnell	SH_CLERK	124
199	Grant	SH_CLERK	124
200	Whalen	AD_ASST	101
201	Hartstein	MK_MAN	100
202	Fay	MK_REP	201
203	Mavris	HR_REP	101
204	Baer	PR_REP	101
205	Higgins	AC_MGR	101
206	Gietz	AC_ACCOUNT	205

107 rows selected.

ORACLE

## Sample Data from the EMPLOYEES Table

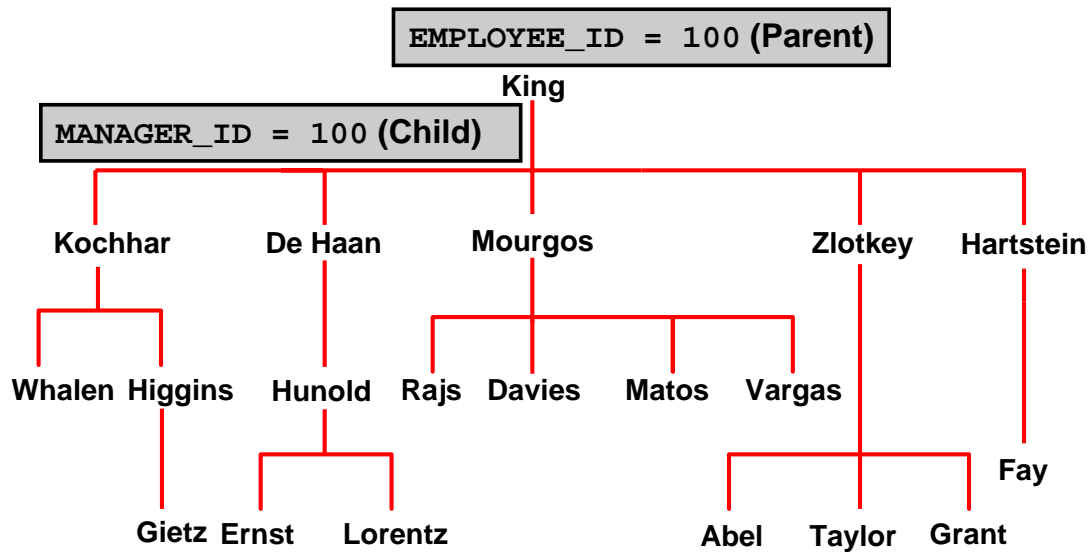
Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called *tree walking* enables the hierarchy to be constructed. A hierarchical query is a method of reporting, the branches of a tree in a specific order.

Imagine a family tree with the eldest members of the family found close to the base or trunk of the tree and the youngest members representing branches of the tree. Branches can have their own branches, and so on.

A hierarchical query is possible when a relationship exists between rows in a table. For example, on the slide, you see that employees with the job IDs of AD\_VP, ST\_MAN, SA\_MAN, and MK\_MAN report directly to the president of the company. We know this because the MANAGER\_ID column of these records contains the employee ID 100, which belongs to the president (AD\_PRES).

**Note:** Hierarchical trees are used in various fields such as human genealogy (family trees), livestock (breeding purposes), corporate management (management hierarchies), manufacturing (product assembly), evolutionary research (species development), and scientific research.

# Natural Tree Structure



ORACLE

7-4

Copyright © 2004, Oracle. All rights reserved.

## Natural Tree Structure

The EMPLOYEES table has a tree structure representing the management reporting line. The hierarchy can be created by looking at the relationship between equivalent values in the EMPLOYEE\_ID and MANAGER\_ID columns. This relationship can be exploited by joining the table to itself. The MANAGER\_ID column contains the employee number of the employee's manager.

The parent-child relationship of a tree structure enables you to control:

- The direction in which the hierarchy is walked
- The starting point inside the hierarchy

**Note:** The slide displays an inverted tree structure of the management hierarchy of the employees in the EMPLOYEES table.

# Hierarchical Queries

```
SELECT [LEVEL], column, expr...  
FROM table  
[WHERE condition(s)]  
[START WITH condition(s)]  
[CONNECT BY PRIOR condition(s)] ;
```

**WHERE *condition*:**

```
expr comparison_operator expr
```

ORACLE®

7-5

Copyright © 2004, Oracle. All rights reserved.

## Keywords and Clauses

Hierarchical queries can be identified by the presence of the CONNECT BY and START WITH clauses.

In the syntax:

SELECT	Is the standard SELECT clause
LEVEL	For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.
FROM <i>table</i>	Specifies the table, view, or snapshot containing the columns. You can select from only one table.
WHERE	Restricts the rows returned by the query without affecting other rows of the hierarchy
<i>condition</i>	Is a comparison with expressions
START WITH	Specifies the root rows of the hierarchy (where to start). This clause is required for a true hierarchical query.
CONNECT BY	Specifies the columns in which the relationship between parent and child PRIOR rows exist. This clause is required for a hierarchical query.

The SELECT statement cannot contain a join or query from a view that contains a join.

# Walking the Tree

## Starting Point

- Specifies the condition that must be met
- Accepts any valid condition

```
START WITH column1 = value
```

Using the **EMPLOYEES** table, start with the employee whose last name is Kochhar.

```
...START WITH last_name = 'Kochhar'
```

ORACLE

7-6

Copyright © 2004, Oracle. All rights reserved.

## Walking the Tree

The row or rows to be used as the root of the tree are determined by the **START WITH** clause. The **START WITH** clause can be used in conjunction with any valid condition.

### Examples

Using the **EMPLOYEES** table, start with King, the president of the company.

```
... START WITH manager_id IS NULL
```

Using the **EMPLOYEES** table, start with employee Kochhar. A **START WITH** condition can contain a subquery.

```
... START WITH employee_id = (SELECT employee_id
                               FROM   employees
                               WHERE  last_name = 'Kochhar')
```

If the **START WITH** clause is omitted, the tree walk is started with all of the rows in the table as root rows. If a **WHERE** clause is used, the walk is started with all the rows that satisfy the **WHERE** condition. This no longer reflects a true hierarchy.

**Note:** The clauses **CONNECT BY PRIOR** and **START WITH** are not ANSI SQL standard.

# Walking the Tree

```
CONNECT BY PRIOR column1 = column2
```

Walk from the top down, using the EMPLOYEES table.

```
... CONNECT BY PRIOR employee_id = manager_id
```

## Direction

Top down	————→	Column1 = Parent Key Column2 = Child Key
Bottom up	————→	Column1 = Child Key Column2 = Parent Key

ORACLE

7-7

Copyright © 2004, Oracle. All rights reserved.

## Walking the Tree (continued)

The direction of the query, whether it is from parent to child or from child to parent, is determined by the `CONNECT BY PRIOR` column placement. The `PRIOR` operator refers to the parent row. To find the child rows of a parent row, the Oracle server evaluates the `PRIOR` expression for the parent row and the other expressions for each row in the table. Rows for which the condition is true are the child rows of the parent. The Oracle server always selects child rows by evaluating the `CONNECT BY` condition with respect to a current parent row.

### Examples

Walk from the top down using the `EMPLOYEES` table. Define a hierarchical relationship in which the `EMPLOYEE_ID` value of the parent row is equal to the `MANAGER_ID` value of the child row.

```
... CONNECT BY PRIOR employee_id = manager_id
```

Walk from the bottom up using the `EMPLOYEES` table.

```
... CONNECT BY PRIOR manager_id = employee_id
```

The `PRIOR` operator does not necessarily need to be coded immediately following the `CONNECT BY`. Thus, the following `CONNECT BY PRIOR` clause gives the same result as the one in the preceding example.

```
... CONNECT BY employee_id = PRIOR manager_id
```

**Note:** The `CONNECT BY` clause cannot contain a subquery.

## Walking the Tree: From the Bottom Up

```
SELECT employee_id, last_name, job_id, manager_id
FROM   employees
START WITH employee_id = 101
CONNECT BY PRIOR manager_id = employee_id ;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
101	Kochhar	AD_VP	100
100	King	AD_PRES	

ORACLE®

7-8

Copyright © 2004, Oracle. All rights reserved.

### Walking the Tree: From the Bottom Up

The example on the slide displays a list of managers starting with the employee whose employee ID is 101.

#### Example

In the following example, EMPLOYEE\_ID values are evaluated for the parent row and MANAGER\_ID, and SALARY values are evaluated for the child rows. The PRIOR operator applies only to the EMPLOYEE\_ID value.

```
... CONNECT BY PRIOR employee_id = manager_id
                AND salary > 15000;
```

To qualify as a child row, a row must have a MANAGER\_ID value equal to the EMPLOYEE\_ID value of the parent row and must have a SALARY value greater than \$15,000.

## Walking the Tree: From the Top Down

```
SELECT last_name || ' reports to ' ||  
PRIOR last_name "Walk Top Down"  
FROM employees  
START WITH last_name = 'King'  
CONNECT BY PRIOR employee_id = manager_id ;
```

### Walk Top Down

King reports to

King reports to

Kochhar reports to King

Greenberg reports to Kochhar

Faviet reports to Greenberg

Chen reports to Greenberg

...

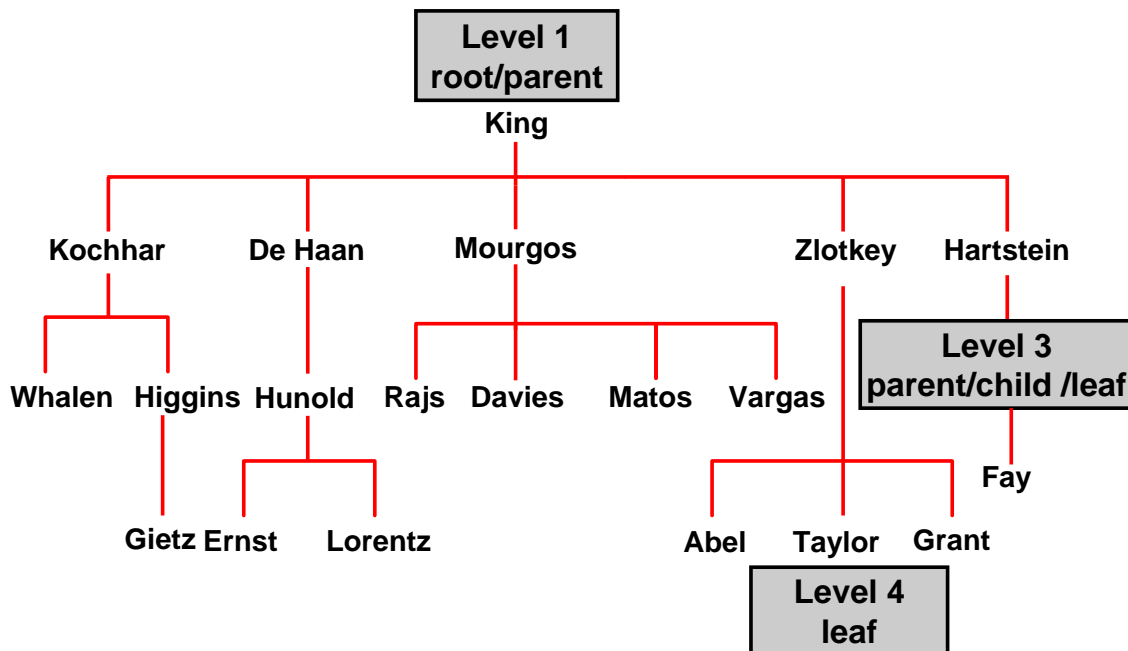
108 rows selected.

ORACLE

### Walking the Tree: From the Top Down

Walking from the top down, display the names of the employees and their manager. Use employee King as the starting point. Print only one column.

## Ranking Rows with the LEVEL Pseudocolumn



ORACLE

7-10

Copyright © 2004, Oracle. All rights reserved.

### Ranking Rows with the LEVEL Pseudocolumn

You can explicitly show the rank or level of a row in the hierarchy by using the LEVEL pseudocolumn. This will make your report more readable. The forks where one or more branches split away from a larger branch are called nodes, and the very end of a branch is called a leaf, or leaf node. The diagram on the slide shows the nodes of the inverted tree with their LEVEL values. For example, employee Higgins is a parent and a child, whereas employee Davies is a child and a leaf.

#### The LEVEL Pseudocolumn

Value	Level
1	A root node
2	A child of a root node
3	A child of a child, and so on

On the slide, King is the root or parent (LEVEL = 1). Kochhar, De Haan, Mourgos, Zlotkey, Hartstein, Higgins, and Hunold are children and also parents (LEVEL = 2). Whalen, Rajs, Davies, Matos, Vargas, Gietz, Ernst, Lorentz, Abel, Taylor, Grant, and Fay are children and leaves. (LEVEL = 3 and LEVEL = 4)

**Note:** A *root node* is the highest node within an inverted tree. A *child node* is any nonroot node. A parent node is any node that has children. A leaf node is any node without children. The number of levels returned by a hierarchical query may be limited by available user memory.



# Formatting Hierarchical Reports Using LEVEL and LPAD

Create a report displaying company management levels, beginning with the highest level and indenting each of the following levels.

```
COLUMN org_chart FORMAT A12
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_')
       AS org_chart
FROM   employees
START WITH last_name='King'
CONNECT BY PRIOR employee_id=manager_id
```

ORACLE

7-11

Copyright © 2004, Oracle. All rights reserved.

## Formatting Hierarchical Reports Using LEVEL

The nodes in a tree are assigned level numbers from the root. Use the LPAD function in conjunction with the pseudocolumn LEVEL to display a hierarchical report as an indented tree.

In the example on the slide:

- `LPAD(char1, n [, char2])` returns `char1`, left-padded to length `n` with the sequence of characters in `char2`. The argument `n` is the total length of the return value as it is displayed on your terminal screen.
- `LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_')` defines the display format.
- `char1` is the `LAST_NAME`, `n` the total length of the return value, is length of the `LAST_NAME + (LEVEL*2) - 2`, and `char2` is `'_'`.

In other words, this tells SQL to take the `LAST_NAME` and left-pad it with the `'_'` character until the length of the resultant string is equal to the value determined by `LENGTH(last_name) + (LEVEL*2) - 2`.

For King, `LEVEL = 1`. Therefore,  $(2 * 1) - 2 = 2 - 2 = 0$ . So King does not get padded with any `'_'` character and is displayed in column 1.

For Kochhar, `LEVEL = 2`. Therefore,  $(2 * 2) - 2 = 4 - 2 = 2$ . So Kochhar gets padded with 2 `'_'` characters and is displayed indented.

The rest of the records in the `EMPLOYEES` table are displayed similarly.

## Formatting Hierarchical Reports Using LEVEL (continued)

ORG_CHART	
King	
King	
_Kochhar	
_Greenber g	
_Faviet	
Chen	
Sciarr a	
Urman	
Popp	
Whalen	
Mavris	
Baer	
Higgins	
Gietz	
...	
_Kumar	
_Zlotkey	
_Abel	
_Hutton	
_Taylor	
Livingst on	
Grant	
Johnson	
Hartstein	
Fay	

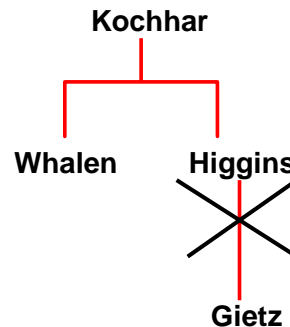
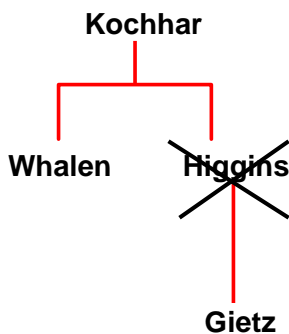
108 rows selected.

# Pruning Branches

Use the WHERE clause  
to eliminate a node.

Use the CONNECT BY clause  
to eliminate a branch.

```
WHERE last_name != 'Higgins'
CONNECT BY PRIOR
employee_id = manager_id
AND last_name != 'Higgins'
```



ORACLE

7-13

Copyright © 2004, Oracle. All rights reserved.

## Pruning Branches

You can use the WHERE and CONNECT BY clauses to prune the tree; that is, to control which nodes or rows are displayed. The predicate you use acts as a Boolean condition.

### Examples

Starting at the root, walk from the top down, and eliminate employee Higgins in the result, but process the child rows.

```
SELECT department_id, employee_id, last_name, job_id, salary
FROM employees
WHERE last_name != 'Higgins'
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id;
```

Starting at the root, walk from the top down, and eliminate employee Higgins and all child rows.

```
SELECT department_id, employee_id, last_name, job_id, salary
FROM employees
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'Higgins';
```

## Summary

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

- **You can use hierarchical queries to view a hierarchical relationship between rows in a table.**
- **You specify the direction and starting point of the query.**
- **You can eliminate nodes or branches by pruning.**

ORACLE

7-14

Copyright © 2004, Oracle. All rights reserved.

### Summary

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table. The `LEVEL` pseudocolumn counts how far down a hierarchical tree you have traveled. You can specify the direction of the query using the `CONNECT BY PRIOR` clause. You can specify the starting point using the `START WITH` clause. You can use the `WHERE` and `CONNECT BY` clauses to prune the tree branches.

## Practice 7: Overview

**This practice covers the following topics:**

- **Distinguishing hierarchical queries from nonhierarchical queries**
- **Walking through a tree**
- **Producing an indented report by using the `LEVEL` pseudocolumn**
- **Pruning the tree structure**
- **Sorting the output**

ORACLE®

7-15

Copyright © 2004, Oracle. All rights reserved.

### Practice 7: Overview

In this practice, you gain practical experience in producing hierarchical reports.

**Note:** Question 1 is a paper-based question.

## Practice 7

1. Look at the following output examples. Are they the result of a hierarchical query? Explain why or why not.

**Exhibit 1:**

EMPLOYEE_ID	LAST_NAME	MANAGER_ID	SALARY	DEPARTMENT_ID
100	King		24000	90
101	Kochhar	100	17000	90
102	De Haan	100	17000	90
201	Hartstein	100	13000	20
205	Higgins	101	12000	110
174	Abel	149	11000	80
149	Zlotkey	100	10500	80
103	Hunold	102	9000	60
■ ■ ■				
200	Whalen	101	4400	10
107	Lorentz	103	4200	60
141	Rajs	124	3500	50
142	Davies	124	3100	50
143	Matos	124	2600	50
144	Vargas	124	2500	50

20 rows selected.

**Exhibit 2:**

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
205	Higgins	110	Accounting
206	Gietz	110	Accounting
100	King	90	Executive
101	Kochhar	90	Executive
102	De Haan	90	Executive
149	Zlotkey	80	Sales
174	Abel	80	Sales
176	Taylor	80	Sales
103	Hunold	60	IT
104	Ernst	60	IT
107	Lorentz	60	IT

11 rows selected.

## Practice 7 (continued)

### Exhibit 3:

RANK	LAST_NAME
1	King
2	Kochhar
2	De Haan
3	Hunold
4	Ernst

2. Produce a report showing an organization chart for Mourgos's department. Print last names, salaries, and department IDs.

LAST_NAME	SALARY	DEPARTMENT_ID
Mourgos	5800	50
Rajs	3500	50
Davies	3100	50
Matos	2600	50
Vargas	2500	50
Walsh	3100	50
Feeney	3000	50
OConnell	2600	50
Grant	2600	50

9 rows selected.

3. Create a report that shows the hierarchy of the managers for the employee Lorentz. Display his immediate manager first.

LAST_NAME
Hunold
De Haan
King

### Practice 7 (continued)

4. Create an indented report showing the management hierarchy starting from the employee whose LAST\_NAME is Kochhar. Print the employee's last name, manager ID, and department ID. Give alias names to the columns as shown in the sample output.

NAME	MGR	DEPTNO
Kochhar	100	90
__Greenberg	101	100
___Faviet	108	100
___Chen	108	100
___Sciarra	108	100
___Urman	108	100
___Popp	108	100
__Whalen	101	10
__Mavris	101	40
__Baer	101	70
__Higgins	101	110
___Gietz	205	110

12 rows selected.

If you have time, complete the following exercise:

5. Produce a company organization chart that shows the management hierarchy. Start with the person at the top level, exclude all people with a job ID of IT\_PROG, and exclude De Haan and those employees who report to De Haan.

LAST_NAME	EMPLOYEE_ID	MANAGER_ID
King	100	
Kochhar	101	100
Greenberg	108	101
Faviet	109	108
Chen	110	108
Sciarra	111	108

...

LAST_NAME	EMPLOYEE_ID	MANAGER_ID
Livingston	177	149
Grant	178	149
Johnson	179	149
Hartstein	201	100
Fay	202	201

101 rows selected.



# 8

## Regular Expression Support

ORACLE®

Copyright © 2004, Oracle. All rights reserved.

# Objectives

**After completing this lesson, you should be able to use regular expression support in SQL to search, match, and replace strings all in terms of regular expressions.**



**ORACLE**

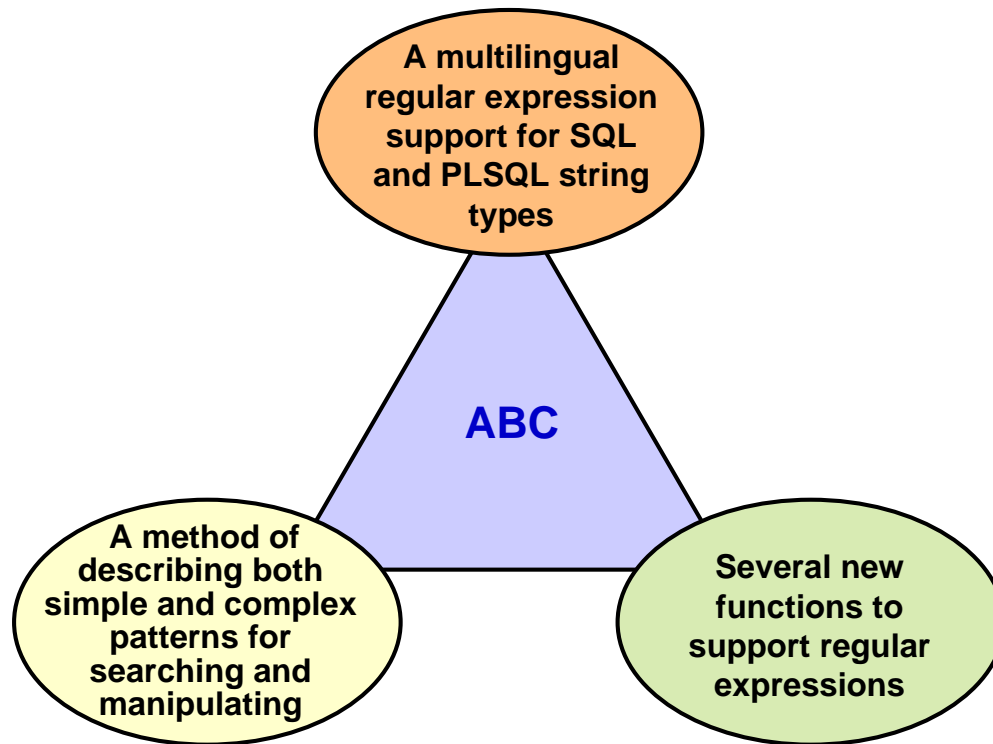
8-2

Copyright © 2004, Oracle. All rights reserved.

## Objectives

In this lesson you learn to use the regular expression support feature that has been introduced in Oracle Database 10g.

# Regular Expression Overview



ORACLE

8-3

Copyright © 2004, Oracle. All rights reserved.

## Regular Expression Overview

Oracle Database 10g introduces support for Regular Expressions. The implementation complies with the Portable Operating System for UNIX (POSIX) standard, controlled by the Institute of Electrical and Electronics Engineers (IEEE), for ASCII data matching semantics and syntax. Oracle's multilingual capabilities extend the matching capabilities of the operators beyond the POSIX standard. Regular expressions are a method of describing both simple and complex patterns for searching and manipulating.

String manipulation and searching contribute to a large percentage of the logic within a Web-based application. Usage ranges from the simple: find the word "San Francisco" in a specified text; to the complex extract of all URLs from the text; to the more complex: find all words whose every second character is a vowel.

When coupled with native SQL, the use of regular expressions allows for very powerful search and manipulation operations on any data stored in an Oracle database. You can use this feature to easily solve problems that would otherwise be very complex to program.

# Meta Characters

Symbol	Description
*	Matches zero or more occurrences
	Alteration operator for specifying alternative matches
^/\$	Matches the start-of-line/end-of-line
[ ]	Bracket expression for a matching list matching any one of the expressions represented in the list
{m}	Matches exactly <i>m</i> times
{m,n}	Matches at least <i>m</i> times but no more than <i>n</i> times
[ : ]	Specifies a character class and matches any character in that class
\	Can have 4 different meanings: 1. Stand for itself. 2. Quote the next character. 3. Introduce an operator. 4. Do nothing.
+	Matches one or more occurrence
?	Matches zero or one occurrence
.	Matches any character in the supported character set, except NULL
()	Grouping expression, treated as a single subexpression
[==]	Specifies equivalence classes
\n	Back-reference expression
[..]	Specifies one collation element, such as a multicharacter element

ORACLE

8-4

Copyright © 2004, Oracle. All rights reserved.

## Meta Characters

Meta characters are special characters that have a special meaning, such as a wildcard character, a repeating character, a nonmatching character, or a range of characters. You can use several predefined meta character symbols in the pattern matching.

## Using Meta Characters

Problem: Find 'abc' within a string:

Solution:

Matches: abc

Does not match: 'def'

1

Problem: To find 'a' followed by any character, followed by 'c'

Meta Character: any character is defined by '.'

Solution:

Matches: abc

Matches: adc

Matches: alc

Matches: a&c

Does not match: abb

2

Problem: To find one or more occurrences of 'a'

Meta Character: Use '+' sign to match one or more of the previous characters

Solution:

Matches: a

Matches: aa

Does not match: bbb

3

ORACLE

8-5

Copyright © 2004, Oracle. All rights reserved.

### Using Meta Characters

1. In the first example, a simple match is performed.
2. In the second example, the any character is defined as a '.'. This example searches for the character "a" followed by any character, followed by the character "c".
3. The third example searches for one or more occurrences of the letter "a." The "+" character is used here to indicate a match of one or more of the previous characters.

You can search for nonmatching character lists too. A nonmatching character list allows you to define a set of characters for which a match is invalid. For example, to find anything but the characters "a," "b," or "c," you can define the "^" to indicate a nonmatch.

Expression: [^abc]

Matches: abcdef

Matches: ghi

Does not match: abc

To match any letter not between "a" and "i," you can use:

Expression: [^a-i]

Matches: hijk

Matches: lmn

Does not match: abcdefghi

## Using Meta Characters (continued)

Meta Character Syntax	Operator Name	Description
.	Any Character – Dot	Match any character
+	One or More – Plus Quantifier	Match one or more occurrences of the preceding subexpression
?	Zero or One – Question Mark Quantifier	Match zero or one occurrence of the preceding subexpression
*	Zero or More – Star Quantifier	Match zero or more occurrences of the preceding subexpression
{ <i>m</i> } { <i>m</i> ,} { <i>m</i> , <i>n</i> }	Interval – Exact Count	Match <ul style="list-style-type: none"> <li>• exactly <i>m</i> occurrences</li> <li>• at least <i>m</i> occurrences</li> <li>• at least <i>m</i>, but not more than <i>n</i> occurrences of the preceding subexpression</li> </ul>
[...]	Matching Character List	Match any character in list ...
[^...]	Non-Matching Character List	Match any character not in list ...
	Or	'a b' matches character 'a' or 'b'.
(...)	Subexpression or Grouping	Treat expression ... as a unit.
\n	Back reference	Match the <i>n</i> <sup>th</sup> preceding subexpression, where <i>n</i> is an integer from 1 to 9
\	Escape Character	Treat the subsequent meta character in the expression as a literal.
^	Beginning of Line Anchor	Match the subsequent expression when it occurs at the beginning of a line.
\$	End of Line Anchor	Match the preceding expression only when it occurs at the end of a line.
[ <i>:class:</i> ]	POSIX Character Class	Match any character belonging to the specified character <i>class</i> .

## Regular Expression Functions

Function Name	Description
<b>REGEXP_LIKE</b>	Similar to the <b>LIKE</b> operator, but performs regular expression matching instead of simple pattern matching
<b>REGEXP_REPLACE</b>	Searches for a regular expression pattern and replaces it with a replacement string
<b>REGEXP_INSTR</b>	Searches for a given string for a regular expression pattern and returns the position where the match is found
<b>REGEXP_SUBSTR</b>	Searches for a regular expression pattern within a given string and returns the matched substring

ORACLE®

8-7

Copyright © 2004, Oracle. All rights reserved.

### Regular Expression Functions

The Oracle Database 10g provides a set of SQL functions that you can use to search and manipulate strings using regular expressions. You can use these functions on any data type that holds character data such as CHAR, NCHAR, CLOB, NCLOB, NVARCHAR2, and VARCHAR2. A regular expression must be enclosed or wrapped between single quotation marks. Doing so ensures that the entire expression is interpreted by the SQL function and can improve the readability of your code.

**REGEXP\_LIKE:** This function searches a character column for a pattern. Use this function in the WHERE clause of a query to return rows matching the regular expression you specify.

**REGEXP\_REPLACE:** This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern you specify.

**REGEXP\_INSTR:** This function searches a string for a given occurrence of a regular expression pattern. You specify which occurrence you want to find and the start position to search from. This function returns an integer indicating the position in the string where the match is found.

**REGEXP\_SUBSTR:** This function returns the actual substring matching the regular expression pattern you specify.

# The REGEXP Function Syntax

```
REGEXP_LIKE (srcstr, pattern [,match_option])
```

```
REGEXP_INSTR (srcstr, pattern [, position [, occurrence  
[, return_option [, match_option]]]])
```

```
REGEXP_SUBSTR (srcstr, pattern [, position  
[, occurrence [, match_option]]])
```

```
REGEXP_REPLACE(srcstr, pattern [,replacestr [, position  
[, occurrence [, match_option]]]])
```

ORACLE

8-8

Copyright © 2004, Oracle. All rights reserved.

## The REGEXP Function Syntax

The following table contains descriptions of the terms shown in the syntax on the slide.

srcstr	Search value
pattern	Regular expression
occurrence	Occurrence to search for
position	Search starting position
return_option	Start or end position of occurrence
replacestr	Character string replacing pattern
match_option	Option to change default matching; it can include one or more of the following values: “c” —uses case-sensitive matching (default) “i” —uses case-insensitive matching “n” —allows match-any-character operator “m” —treats source string as multiple line



## Performing Basic Searches

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

FIRST_NAME	LAST_NAME
Steven	King
Steven	Markle
Stephen	Stiles

ORACLE

8-9

Copyright © 2004, Oracle. All rights reserved.

### Example of REGEXP\_LIKE

In this query, against the EMPLOYEES table, all employees with first names containing either Steven or Stephen are displayed. In the expression used,

'^Ste(v|ph)en\$' :

- ^ indicates the beginning of the sentence
- \$ indicates the end of the sentence
- | indicates either/or

## Checking the Presence of a Pattern

```
SELECT street_address,  
       REGEXP_INSTR(street_address,'^[[:alpha:]]')  
FROM   locations  
WHERE  
       REGEXP_INSTR(street_address,'^[[:alpha:]]')> 1;
```

STREET_ADDRESS	REGEXP_INSTR(STREET_ADDRESS,'^[[:ALPHA:]]')
Magdalen Centre, The Oxford Science Park	9
Schwanthalerstr. 7031	16
Rua Frei Caneca 1360	4
Murtenstrasse 921	14
Pieter Breughelstraat 837	7
Mariano Escobedo 9991	8

ORACLE

8-10

Copyright © 2004, Oracle. All rights reserved.

### Checking the Presence of a Pattern

In this example, the `REGEXP_INSTR` function is used to search the street address to find the location of the first nonalphabetic character, regardless of whether it is in upper or lower case. The search is performed only on those addresses that do not start with a number. Note that `[<class>:]` implies a character class and matches any character from within that class; `[[:alpha:]]` matches with any alphabetic character. The results are displayed.

In the expression used in the query `'^[[:alpha:]]'`:

- `[` starts the expression
- `^` indicates NOT
- `[[:alpha:]]` indicates alpha character class
- `]` ends the expression

**Note:** The POSIX character class operator enables you to search for an expression within a character list that is a member of a specific POSIX character class. You can use this operator to search for specific formatting, such as uppercase characters, or you can search for special characters such as digits or punctuation characters. The full set of POSIX character classes is supported. Use the syntax `[<class>:]` where *class* is the name of the POSIX character class to search for. The following regular expression searches for one or more consecutive uppercase characters : `[[:upper:]]+`.

## Example of Extracting Substrings

```
SELECT REGEXP_SUBSTR(street_address , ' [^ ]+ ')  
"Road" FROM locations;
```

Road
Via
Calle
Jabberwocky
Interiors
Zagora
Charade
...

ORACLE

8-11

Copyright © 2004, Oracle. All rights reserved.

### Example of Extracting a Substring

In this example, the road names are extracted from the LOCATIONS table. To do this, the contents in the STREET\_ADDRESS column that are before the first space are returned using the REGEXP\_SUBSTR function. In the expression used in the query ' [^ ]+ ':

- [ starts the expression
- ^ indicates NOT
- indicates space
- ] ends the expression
- + indicates 1 or more
- indicates space

# Replacing Patterns

```
SELECT REGEXP_REPLACE( country_name, '(.)',  
                        '\1 ') "REGEXP_REPLACE"  
FROM countries;
```

REGEXP_REPLACE(COUNTRY_NAME,'(',')','\1')
Argentina
Australia
Belgium
Brazil
Canada
Switzerland
China
...

ORACLE®

## Replacing Patterns

This example examines examines COUNTRY\_NAME. The Oracle database reformats this pattern with a space after each non-null character in the string. The results are shown.

## Regular Expressions and Check Constraints

```
ALTER TABLE emp8
ADD CONSTRAINT email_addr
CHECK(REGEXP_LIKE(email,'@'))NOVALIDATE ;
```

1

```
INSERT INTO emp8 VALUES
(500,'Christian','Patel',
'ChrisP2creme.com', 1234567890,
'12-Jan-2004', 'HR_REP', 2000, null, 102, 40) ;
```

2

```
INSERT INTO emp8 VALUES
*
```

```
ERROR at line 1:
ORA-02290: check constraint (ORA20.EMAIL_ADDR) violated
```

ORACLE

### Regular Expressions and Check Constraints

Regular expressions can also be used in check constraints. In this example, a check constraint is added on the EMAIL column of the EMPLOYEES table. This will ensure that only strings containing an “@” symbol are accepted. The constraint is tested. The check constraint is violated because the e-mail address does not contain the required symbol. The NOVALIDATE clause ensures that existing data is not checked.

## Summary

**In this lesson, you should have learned how to use regular expression support in SQL and PL/SQL to search, match, and replace strings all in terms of regular expressions.**

ORACLE®

8-14

Copyright © 2004, Oracle. All rights reserved.

### Summary

In this lesson you have learned to use the regular expression support features that have been introduced in Oracle Database 10g.

## Practice 8: Overview

**This practice covers using regular expressions.**

ORACLE

8-15

Copyright © 2004, Oracle. All rights reserved.

### **Practice 8: Overview**

This practice covers searching and replacing data using regular expressions.

## Practice 8

1. Write a query to search the EMPLOYEES table for all employees whose first names start with “Ne” or “Na.”

FIRST_NAME	LAST_NAME
Nanette	Cambrault
Nancy	Greenberg
Neena	Kochhar
Nandita	Sarchand

2. Create a query that removes the spaces in the STREET\_ADDRESS column of LOCATIONS table in the display.

REGEXP_REPLACE(STREET_ADDRESS,',')
1297ViaColadiRie
93091CalledellaTesta
2017Shinjuku-ku
9450Kamiya-cho
2014JabberwockyRd
2011InteriorsBlvd
2007ZagoraSt
2004CharadeRd
147SpadinaAve
6092BoxwoodSt
40-5-12Laogianggen
1298Vileparle(E)
12-98VictoriaStreet
198ClementiNorth
8204ArthurSt
MagdalenCentre,TheOxfordSciencePark
9702ChesterRoad
Schwanthalerstr.7031
RuaFreiCaneca1360
20RuedesCorps-Saints
Murtenstrasse921
PieterBreughelstraat837
MarianoEscobedo9991

23 rows selected.



### Practice 8 (continued)

3. Create a query that displays “St” replaced by “Street” in the STREET\_ADDRESS column of LOCATIONS table. Be careful that you do not affect any rows that already have “Street” in them. Display only those rows which are affected.

REGEXP_REPLACE(STREET_ADDRESS,'ST\$','STREET')
2007 Zagora Street
6092 Boxwood Street
12-98 Victoria Street
8204 Arthur Street

