

---

## **Oracle Database 11g: SQL Fundamentals I**

**Electronic Presentation**

---

D49996GC20  
Edition 2.0  
October 2009

**ORACLE®**



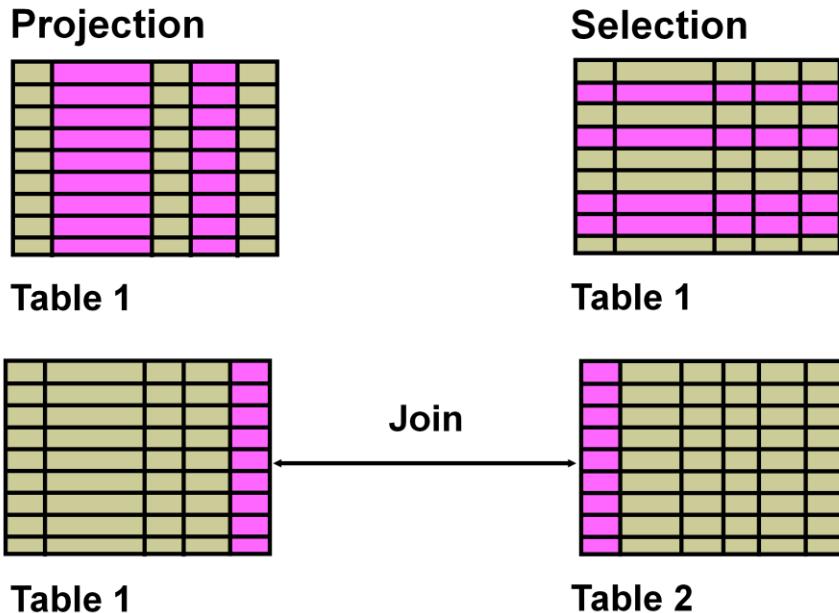
# 1

## Retrieving Data Using the SQL SELECT Statement

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

# Capabilities of SQL SELECT Statements



1 - 2

Copyright © 2009, Oracle. All rights reserved.

ORACLE®

## Capabilities of SQL SELECT Statements

A `SELECT` statement retrieves information from the database. With a `SELECT` statement, you can do the following:

- **Projection:** Select the columns in a table that are returned by a query. Select as few or as many of the columns as required.
  - **Selection:** Select the rows in a table that are returned by a query. Various criteria can be used to restrict the rows that are retrieved.
  - **Joins:** Bring together data that is stored in different tables by specifying the link between them. SQL joins are covered in more detail in the lesson titled “Displaying Data from Multiple Tables Using Joins.”

## Basic SELECT Statement

```
SELECT * | { [DISTINCT] column|expression [alias], ... }  
FROM    table;
```

- SELECT identifies the columns to be displayed.
- FROM identifies the table containing those columns.

ORACLE

1 - 3

Copyright © 2009, Oracle. All rights reserved.

### Basic SELECT Statement

In its simplest form, a SELECT statement must include the following:

- A SELECT clause, which specifies the columns to be displayed
- A FROM clause, which identifies the table containing the columns that are listed in the SELECT clause

In the syntax:

SELECT	Is a list of one or more columns
*	Selects all columns
DISTINCT	Suppresses duplicates
column expression	Selects the named column or the expression
alias	Gives the selected columns different headings
FROM table	Specifies the table containing the columns

**Note:** Throughout this course, the words *keyword*, *clause*, and *statement* are used as follows:

- A *keyword* refers to an individual SQL element—for example, SELECT and FROM are keywords.
- A *clause* is a part of a SQL statement—for example, SELECT employee\_id, last\_name, and so on.
- A *statement* is a combination of two or more clauses—for example, SELECT \* FROM employees.

# Selecting All Columns

```
SELECT *
FROM departments;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

ORACLE

## Selecting All Columns

You can display all columns of data in a table by following the SELECT keyword with an asterisk (\*). In the example in the slide, the DEPARTMENTS table contains four columns: DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, and LOCATION\_ID. The table contains eight rows, one for each department.

You can also display all columns in the table by listing all the columns after the SELECT keyword. For example, the following SQL statement (like the example in the slide) displays all columns and all rows of the DEPARTMENTS table:

```
SELECT department_id, department_name, manager_id, location_id
FROM departments;
```

**Note:** In SQL Developer, you can enter your SQL statement in a SQL Worksheet and click the “Execute Statement” icon or press [F9] to execute the statement. The output displayed on the Results tabbed page appears as shown in the slide.

# Selecting Specific Columns

```
SELECT department_id, location_id  
FROM departments;
```

	DEPARTMENT_ID	LOCATION_ID
1	10	1700
2	20	1800
3	50	1500
4	60	1400
5	80	2500
6	90	1700
7	110	1700
8	190	1700

ORACLE

1 - 5

Copyright © 2009, Oracle. All rights reserved.

## Selecting Specific Columns

You can use the SELECT statement to display specific columns of the table by specifying the column names, separated by commas. The example in the slide displays all the department numbers and location numbers from the DEPARTMENTS table.

In the SELECT clause, specify the columns that you want in the order in which you want them to appear in the output. For example, to display location before department number (from left to right), you use the following statement:

```
SELECT location_id, department_id  
FROM departments;
```

	LOCATION_ID	DEPARTMENT_ID
1	1700	10
2	1800	20
3	1500	50
4	1400	60

• • •

# Writing SQL Statements

- SQL statements are not case sensitive.
- SQL statements can be entered on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.
- In SQL Developer, SQL statements can be optionally terminated by a semicolon (;). Semicolons are required when you execute multiple SQL statements.
- In SQL\*Plus, you are required to end each SQL statement with a semicolon (;).

ORACLE

1 - 6

Copyright © 2009, Oracle. All rights reserved.

## Writing SQL Statements

By using the following simple rules and guidelines, you can construct valid statements that are both easy to read and edit:

- SQL statements are not case sensitive (unless indicated).
- SQL statements can be entered on one or many lines.
- Keywords cannot be split across lines or abbreviated.
- Clauses are usually placed on separate lines for readability and ease of editing.
- Indents should be used to make code more readable.
- Keywords typically are entered in uppercase; all other words, such as table names and column names are entered in lowercase.

## Executing SQL Statements

In SQL Developer, click the Run Script icon or press [F5] to run the command or commands in the SQL Worksheet. You can also click the Execute Statement icon or press [F9] to run a SQL statement in the SQL Worksheet. The Execute Statement icon executes the statement at the mouse pointer in the Enter SQL Statement box while the Run Script icon executes all the statements in the Enter SQL Statement box. The Execute Statement icon displays the output of the query on the Results tabbed page, whereas the Run Script icon emulates the SQL\*Plus display and shows the output on the Script Output tabbed page.

# Column Heading Defaults

- SQL Developer:
  - Default heading alignment: Left-aligned
  - Default heading display: Uppercase
- SQL\*Plus:
  - Character and Date column headings are left-aligned.
  - Number column headings are right-aligned.
  - Default heading display: Uppercase

ORACLE

1 - 7

Copyright © 2009, Oracle. All rights reserved.

## Column Heading Defaults

In SQL Developer, column headings are displayed in uppercase and are left-aligned.

```
SELECT last_name, hire_date, salary  
FROM employees;
```

	LAST_NAME	HIRE_DATE	SALARY
1	Whalen	17-SEP-87	4400
2	Hartstein	17-FEB-96	13000
3	Fay	17-AUG-97	6000
4	Higgins	07-JUN-94	12000
5	Gietz	07-JUN-94	8300
6	King	17-JUN-87	24000
7	Kochhar	21-SEP-89	17000
8	De Haan	13-JAN-93	17000
9	Hunold	03-JAN-90	9000

...

You can override the column heading display with an alias. Column aliases are covered later in this lesson.

# Arithmetic Expressions

Create expressions with number and date data by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

ORACLE

1 - 8

Copyright © 2009, Oracle. All rights reserved.

## Arithmetic Expressions

You may need to modify the way in which data is displayed, or you may want to perform calculations, or look at what-if scenarios. All these are possible using arithmetic expressions. An arithmetic expression can contain column names, constant numeric values, and the arithmetic operators.

### Arithmetic Operators

The slide lists the arithmetic operators that are available in SQL. You can use arithmetic operators in any clause of a SQL statement (except the FROM clause).

**Note:** With the DATE and TIMESTAMP data types, you can use the addition and subtraction operators only.

# Using Arithmetic Operators

```
SELECT last_name, salary, salary + 300  
FROM employees;
```

	LAST_NAME	SALARY	SALARY+300
1	Whalen	4400	4700
2	Hartstein	13000	13300
3	Fay	6000	6300
4	Higgins	12000	12300
5	Gietz	8300	8600
6	King	24000	24300
7	Kochhar	17000	17300
8	De Haan	17000	17300
9	Hunold	9000	9300
10	Ernst	6000	6300

...

ORACLE

## Using Arithmetic Operators

The example in the slide uses the addition operator to calculate a salary increase of \$300 for all employees. The slide also displays a SALARY+300 column in the output.

Note that the resultant calculated column, SALARY+300, is not a new column in the EMPLOYEES table; it is for display only. By default, the name of a new column comes from the calculation that generated it—in this case, salary+300.

**Note:** The Oracle server ignores blank spaces before and after the arithmetic operator.

## Operator Precedence

If an arithmetic expression contains more than one operator, multiplication and division are evaluated first. If operators in an expression are of the same priority, evaluation is done from left to right.

You can use parentheses to force the expression that is enclosed by the parentheses to be evaluated first.

### Rules of Precedence:

- Multiplication and division occur before addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to override the default precedence or to clarify the statement.

# Operator Precedence

```
SELECT last_name, salary, 12*salary+100  
FROM employees;
```

1

	LAST_NAME	SALARY	12*SALARY+100
1	Whalen	4400	52900
2	Hartstein	13000	156100
3	Fay	6000	72100

```
SELECT last_name, salary, 12*(salary+100)  
FROM employees;
```

2

	LAST_NAME	SALARY	12*(SALARY+100)
1	Whalen	4400	54000
2	Hartstein	13000	157200
3	Fay	6000	73200

ORACLE

1 - 10

Copyright © 2009, Oracle. All rights reserved.

## Operator Precedence (continued)

The first example in the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation by multiplying the monthly salary with 12, plus a one-time bonus of \$100. Note that multiplication is performed before addition.

**Note:** Use parentheses to reinforce the standard order of precedence and to improve clarity. For example, the expression in the slide can be written as  $(12 * \text{salary}) + 100$  with no change in the result.

## Using Parentheses

You can override the rules of precedence by using parentheses to specify the desired order in which the operators are to be executed.

The second example in the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation as follows: adding a monthly bonus of \$100 to the monthly salary, and then multiplying that subtotal with 12. Because of the parentheses, addition takes priority over multiplication.

## Defining a Null Value

- Null is a value that is unavailable, unassigned, unknown, or inapplicable.
- Null is not the same as zero or a blank space.

```
SELECT last_name, job_id, salary, commission_pct  
FROM employees;
```

	LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
1	Whalen	AD_ASST	4400	(null)
2	Hartstein	MK_MAN	13000	(null)
...				
17	Zlotkey	SA_MAN	10500	0.2
18	Abel	SA_REP	11000	0.3
19	Taylor	SA_REP	8600	0.2
20	Grant	SA_REP	7000	0.15

ORACLE

## Defining a Null Value

If a row lacks a data value for a particular column, that value is said to be *null* or to contain a null.

Null is a value that is unavailable, unassigned, unknown, or inapplicable. Null is not the same as zero or a blank space. Zero is a number and blank space is a character.

Columns of any data type can contain nulls. However, some constraints (NOT NULL and PRIMARY KEY) prevent nulls from being used in the column.

In the COMMISSION\_PCT column in the EMPLOYEES table, notice that only a sales manager or sales representative can earn a commission. Other employees are not entitled to earn commissions. A null represents that fact.

**Note:** By default, SQL Developer uses the literal, (null), to identify null values. However, you can set it to something more relevant to you. To do so, select Preferences from the Tools menu. In the Preferences dialog box, expand the Database node. Click Advanced Parameters and on the right pane, for the “Display Null value As,” enter the appropriate value.

# Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

```
SELECT last_name, 12*salary*commission_pct  
FROM employees;
```

LAST_NAME	12*SALARY*COMMISSION_PCT
1 Whalen	(null)
2 Hartstein	(null)
3 Fay	(null)
...	
17 Zlotkey	25200
18 Abel	39600
19 Taylor	20640
20 Grant	12600

ORACLE

## Null Values in Arithmetic Expressions

If any column value in an arithmetic expression is null, the result is null. For example, if you attempt to perform division by zero, you get an error. However, if you divide a number by null, the result is a null or unknown.

In the example in the slide, employee Whalen does not get any commission. Because the COMMISSION\_PCT column in the arithmetic expression is null, the result is null.

For more information, see the section on “Basic Elements of Oracle SQL” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

# Defining a Column Alias

A column alias:

- Renames a column heading
- Is useful with calculations
- Immediately follows the column name (There can also be the optional AS keyword between the column name and the alias.)
- Requires double quotation marks if it contains spaces or special characters, or if it is case-sensitive

ORACLE

1 - 13

Copyright © 2009, Oracle. All rights reserved.

## Defining a Column Alias

When displaying the result of a query, SQL Developer normally uses the name of the selected column as the column heading. This heading may not be descriptive and, therefore, may be difficult to understand. You can change a column heading by using a column alias.

Specify the alias after the column in the SELECT list using blank space as a separator. By default, alias headings appear in uppercase. If the alias contains spaces or special characters (such as # or \$), or if it is case-sensitive, enclose the alias in double quotation marks ("").

# Using Column Aliases

```
SELECT last_name AS name, commission_pct comm  
FROM employees;
```

	NAME	COMM
1	Whalen	(null)
2	Hartstein	(null)
3	Fay	(null)

```
SELECT last_name "Name" , salary*12 "Annual Salary"  
FROM employees;
```

	Name	Annual Salary
1	Whalen	52800
2	Hartstein	156000
3	Fay	72000

ORACLE

## Using Column Aliases

The first example displays the names and the commission percentages of all the employees. Note that the optional AS keyword has been used before the column alias name. The result of the query is the same whether the AS keyword is used or not. Also, note that the SQL statement has the column aliases, name and comm, in lowercase, whereas the result of the query displays the column headings in uppercase. As mentioned in the preceding slide, column headings appear in uppercase by default.

The second example displays the last names and annual salaries of all the employees. Because Annual Salary contains a space, it has been enclosed in double quotation marks. Note that the column heading in the output is exactly the same as the column alias.

# Concatenation Operator

A concatenation operator:

- Links columns or character strings to other columns
- Is represented by two vertical bars (||)
- Creates a resultant column that is a character expression

```
SELECT    last_name||job_id AS "Employees"  
FROM      employees;
```

Employees
1 AbelSA_REP
2 DaviesST_CLERK
3 De HaanAD_VP
4 ErnstIT_PROG
5 FayMK_REP
6 GietzAC_ACCOUNT
...

ORACLE

## Concatenation Operator

You can link columns to other columns, arithmetic expressions, or constant values to create a character expression by using the concatenation operator (||). Columns on either side of the operator are combined to make a single output column.

In the example, LAST\_NAME and JOB\_ID are concatenated, and given the alias Employees. Note that the last name of the employee and the job code are combined to make a single output column.

The AS keyword before the alias name makes the SELECT clause easier to read.

### Null Values with the Concatenation Operator

If you concatenate a null value with a character string, the result is a character string. LAST\_NAME || NULL results in LAST\_NAME.

**Note:** You can also concatenate date expressions with other expressions or columns.

## Literal Character Strings

- A literal is a character, a number, or a date that is included in the SELECT statement.
- Date and character literal values must be enclosed within single quotation marks.
- Each character string is output once for each row returned.

ORACLE

1 - 16

Copyright © 2009, Oracle. All rights reserved.

### Literal Character Strings

A literal is a character, a number, or a date that is included in the SELECT list. It is not a column name or a column alias. It is printed for each row returned. Literal strings of free-format text can be included in the query result and are treated the same as a column in the SELECT list.

The date and character literals *must* be enclosed within single quotation marks (' '); number literals need not be enclosed in a similar manner.

# Using Literal Character Strings

```
SELECT last_name ||' is a '||job_id  
      AS "Employee Details"  
FROM   employees;
```

Employee Details
1 Abel is a SA_REP
2 Davies is a ST_CLERK
3 De Haan is a AD_VP
4 Ernst is a IT_PROG
5 Fay is a MK_REP
6 Gietz is a AC_ACCOUNT
7 Grant is a SA_REP
8 Hartstein is a MK_MAN
9 Higgins is a AC_MGR
10 Hunold is a IT_PROG
...

ORACLE

1 - 17

Copyright © 2009, Oracle. All rights reserved.

## Using Literal Character Strings

The example in the slide displays the last names and job codes of all employees. The column has the heading Employee Details. Note the spaces between the single quotation marks in the SELECT statement. The spaces improve the readability of the output.

In the following example, the last name and salary for each employee are concatenated with a literal, to give the returned rows more meaning:

```
SELECT last_name ||': 1 Month salary = '||salary Monthly  
FROM   employees;
```

MONTHLY
1 Whalen: 1 Month salary = 4400
2 Hartstein: 1 Month salary = 13000
3 Fay: 1 Month salary = 6000
4 Higgins: 1 Month salary = 12000
5 Gietz: 1 Month salary = 8300
6 King: 1 Month salary = 24000
7 Kochhar: 1 Month salary = 17000
8 De Haan: 1 Month salary = 17000
...

## Alternative Quote (q) Operator

- Specify your own quotation mark delimiter.
- Select any delimiter.
- Increase readability and usability.

```
SELECT department_name || q'[ Department's Manager Id: ]'  
    || manager_id  
    AS "Department and Manager"  
FROM departments;
```

Department and Manager
1 Administration Department's Manager Id: 200
2 Marketing Department's Manager Id: 201
3 Shipping Department's Manager Id: 124
4 IT Department's Manager Id: 103
5 Sales Department's Manager Id: 149
6 Executive Department's Manager Id: 100
7 Accounting Department's Manager Id: 205
8 Contracting Department's Manager Id:

ORACLE

1 - 18

Copyright © 2009, Oracle. All rights reserved.

## Alternative Quote (q) Operator

Many SQL statements use character literals in expressions or conditions. If the literal itself contains a single quotation mark, you can use the quote (q) operator and select your own quotation mark delimiter.

You can choose any convenient delimiter, single-byte or multibyte, or any of the following character pairs: [ ], { }, ( ), or < >.

In the example shown, the string contains a single quotation mark, which is normally interpreted as a delimiter of a character string. By using the q operator, however, brackets [ ] are used as the quotation mark delimiters. The string between the brackets delimiters is interpreted as a literal character string.

# Duplicate Rows

The default display of queries is all rows, including duplicate rows.

1

```
SELECT department_id  
FROM employees;
```

	DEPARTMENT_ID
1	10
2	20
3	20
4	110
5	110
...	

2

```
SELECT DISTINCT department_id  
FROM employees;
```

	DEPARTMENT_ID
1	(null)
2	20
3	90
4	110
5	50
6	80
7	10
8	60

ORACLE

1 - 19

Copyright © 2009, Oracle. All rights reserved.

## Duplicate Rows

Unless you indicate otherwise, SQL displays the results of a query without eliminating the duplicate rows. The first example in the slide displays all the department numbers from the EMPLOYEES table. Note that the department numbers are repeated.

To eliminate duplicate rows in the result, include the DISTINCT keyword in the SELECT clause immediately after the SELECT keyword. In the second example in the slide, the EMPLOYEES table actually contains 20 rows, but there are only seven unique department numbers in the table.

You can specify multiple columns after the DISTINCT qualifier. The DISTINCT qualifier affects all the selected columns, and the result is every distinct combination of the columns.

```
SELECT DISTINCT department_id, job_id  
FROM employees;
```

	DEPARTMENT_ID	JOB_ID
1	110	AC_ACCOUNT
2	90	AD_VP
3	50	ST_CLERK

...

**Note:** You may also specify the keyword UNIQUE, which is a synonym for the keyword DISTINCT.

# Displaying the Table Structure

- Use the DESCRIBE command to display the structure of a table.
- Or, select the table in the Connections tree and use the Columns tab to view the table structure.

```
DESC[RIBE] tablename
```

The screenshot shows the Oracle SQL Developer interface. On the left, the 'Connections' tree shows a connection named 'myconnection' with a 'Tables' node expanded, revealing 'COUNTRIES' and 'DEPARTMENTS'. The 'DEPARTMENTS' node is selected. A red box highlights the 'Columns' tab in the top navigation bar of the main panel. The main panel displays the table structure in a grid:

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key	Comments
DEPARTMENT_ID	NUMBER(4,0)	No	(null)	1	1	Primary key column
DEPARTMENT_NAME	VARCHAR2(30 BYTE)	No	(null)	2	(null)	A not null column that
MANAGER_ID	NUMBER(6,0)	Yes	(null)	3	(null)	Manager_id of a dep
LOCATION_ID	NUMBER(4,0)	Yes	(null)	4	(null)	Location id where a

ORACLE

## Displaying the Table Structure

You can display the structure of a table by using the DESCRIBE command. The command displays the column names and the data types, and it shows you whether a column *must* contain data (that is, whether the column has a NOT NULL constraint).

In the syntax, *table name* is the name of any existing table, view, or synonym that is accessible to the user.

Using the SQL Developer GUI interface, you can select the table in the Connections tree and use the Columns tab to view the table structure.

**Note:** The DESCRIBE command is supported by both SQL\*Plus and SQL Developer.

# Using the DESCRIBE Command

```
DESCRIBE employees
```

```
DESCRIBE employees
Name          Null    Type
-----        -----  -----
EMPLOYEE_ID   NOT NULL NUMBER(6)
FIRST_NAME    VARCHAR2(20)
LAST_NAME     NOT NULL VARCHAR2(25)
EMAIL         NOT NULL VARCHAR2(25)
PHONE_NUMBER  VARCHAR2(20)
HIRE_DATE     NOT NULL DATE
JOB_ID        NOT NULL VARCHAR2(10)
SALARY         NUMBER(8,2)
COMMISSION_PCT NUMBER(2,2)
MANAGER_ID    NUMBER(6)
DEPARTMENT_ID NUMBER(4)

11 rows selected
```

ORACLE

1 - 21

Copyright © 2009, Oracle. All rights reserved.

## Using the DESCRIBE Command

The example in the slide displays information about the structure of the EMPLOYEES table using the DESCRIBE command.

In the resulting display, *Null* indicates that the values for this column may be unknown. NOT NULL indicates that a column must contain data. *Type* displays the data type for a column.

The data types are described in the following table:

Data Type	Description
NUMBER ( <i>p, s</i> )	Number value having a maximum number of digits <i>p</i> , with <i>s</i> digits to the right of the decimal point
VARCHAR2 ( <i>s</i> )	Variable-length character value of maximum size <i>s</i>
DATE	Date and time value between January 1, 4712 B.C. and December 31, A.D. 9999

# 2

## Restricting and Sorting Data

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

# Limiting Rows Using a Selection

## EMPLOYEES

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	200 Whalen	AD_ASST	10	
2	201 Hartstein	MK_MAN	20	
3	202 Fay	MK_REP	20	
4	205 Higgins	AC_MGR	110	
5	206 Gietz	AC_ACCOUNT	110	

...

“retrieve all  
employees in  
department 90”

The diagram illustrates a query selection process. At the top, there is a large table labeled "EMPLOYEES" containing five rows of employee data. Below it, a horizontal line with a bracket spans across the table, indicating the scope of the query. A vertical arrow points downwards from this bracket to a smaller table below, which contains only three rows of data, representing the result set for employees in department 90.

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100 King	AD_PRES	90	
2	101 Kochhar	AD_VP	90	
3	102 De Haan	AD_VP	90	

ORACLE

## Limiting Rows Using a Selection

In the example in the slide, assume that you want to display all the employees in department 90. The rows with a value of 90 in the DEPARTMENT\_ID column are the only ones that are returned. This method of restriction is the basis of the WHERE clause in SQL.

## **Limiting the Rows That Are Selected**

- Restrict the rows that are returned by using the WHERE clause:

```
SELECT * | {[DISTINCT] column|expression [alias],...}
FROM   table
[WHERE condition(s)];
```

- The WHERE clause follows the FROM clause.

ORACLE®

2 - 3

Copyright © 2009, Oracle. All rights reserved.

## **Limiting the Rows That Are Selected**

You can restrict the rows that are returned from the query by using the WHERE clause. A WHERE clause contains a condition that must be met and it directly follows the FROM clause. If the condition is true, the row meeting the condition is returned.

In the syntax:

*condition* Is composed of column names, expressions, constants, and a comparison operator. A condition specifies a combination of one or more expressions and logical (Boolean) operators, and returns a value of TRUE, FALSE, or UNKNOWN.

The WHERE clause can compare values in columns, literal, arithmetic expressions, or functions. It consists of three elements:

- Column name
  - Comparison condition
  - Column name, constant, or list of values

## Using the WHERE Clause

```
SELECT employee_id, last_name, job_id, department_id  
FROM   employees  
WHERE  department_id = 90 ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100 King	AD_PRES		90
2	101 Kochhar	AD_VP		90
3	102 De Haan	AD_VP		90

ORACLE

## Using the WHERE Clause

In the example, the SELECT statement retrieves the employee ID, last name, job ID, and department number of all employees who are in department 90.

**Note:** You cannot use column alias in the WHERE clause.

# Character Strings and Dates

- Character strings and date values are enclosed with single quotation marks.
- Character values are case-sensitive and date values are format-sensitive.
- The default date display format is DD-MON-RR.

```
SELECT last_name, job_id, department_id  
FROM   employees  
WHERE  last_name = 'Whalen' ;
```

```
SELECT last_name  
FROM   employees  
WHERE  hire_date = '17-FEB-96' ;
```

ORACLE

2 - 5

Copyright © 2009, Oracle. All rights reserved.

## Character Strings and Dates

Character strings and dates in the WHERE clause must be enclosed with single quotation marks (' '). Number constants, however, need not be enclosed with single quotation marks.

All character searches are case-sensitive. In the following example, no rows are returned because the EMPLOYEES table stores all the last names in mixed case:

```
SELECT last_name, job_id, department_id  
FROM   employees  
WHERE  last_name = 'WHALEN';
```

Oracle databases store dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds. The default date display is in the DD-MON-RR format.

**Note:** For details about the RR format and about changing the default date format, see the lesson titled “Using Single-Row Functions to Customize Output.” Also, you learn about the use of single-row functions such as UPPER and LOWER to override the case sensitivity in the same lesson.

# Comparison Operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
BETWEEN ... AND ...	Between two values (inclusive)
IN (set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

ORACLE

2 - 6

Copyright © 2009, Oracle. All rights reserved.

## Comparison Operators

Comparison operators are used in conditions that compare one expression with another value or expression. They are used in the WHERE clause in the following format:

### Syntax

... WHERE *expr operator value*

### Example

```
... WHERE hire_date = '01-JAN-95'  
... WHERE salary >= 6000  
... WHERE last_name = 'Smith'
```

Remember, an alias cannot be used in the WHERE clause.

**Note:** The symbols != and ^= can also represent the *not equal to* condition.

# Using Comparison Operators

```
SELECT last_name, salary  
FROM   employees  
WHERE  salary <= 3000 ;
```

	LAST_NAME	SALARY
1	Matos	2600
2	Vargas	2500

ORACLE

## Using Comparison Operators

In the example, the SELECT statement retrieves the last name and salary from the EMPLOYEES table for any employee whose salary is less than or equal to \$3,000. Note that there is an explicit value supplied to the WHERE clause. The explicit value of 3000 is compared to the salary value in the SALARY column of the EMPLOYEES table.

## Range Conditions Using the BETWEEN Operator

Use the BETWEEN operator to display rows based on a range of values:

```
SELECT last_name, salary  
FROM employees  
WHERE salary BETWEEN 2500 AND 3500;
```

Lower limit      Upper limit

	LAST_NAME	SALARY
1	Rajs	3500
2	Davies	3100
3	Matos	2600
4	Vargas	2500

ORACLE

2 - 8

Copyright © 2009, Oracle. All rights reserved.

## Range Conditions Using the BETWEEN Operator

You can display rows based on a range of values using the BETWEEN operator. The range that you specify contains a lower limit and an upper limit.

The SELECT statement in the slide returns rows from the EMPLOYEES table for any employee whose salary is between \$2,500 and \$3,500.

Values that are specified with the BETWEEN operator are inclusive. However, you must specify the lower limit first.

You can also use the BETWEEN operator on character values:

```
SELECT last_name  
FROM employees  
WHERE last_name BETWEEN 'King' AND 'Smith';
```

	LAST_NAME
1	King
2	Kochhar
3	Lorentz
4	Matos
5	Mourgos
6	Rajs

# Membership Condition Using the IN Operator

Use the IN operator to test for values in a list:

```
SELECT employee_id, last_name, salary, manager_id  
FROM   employees  
WHERE  manager_id IN (100, 101, 201) ;
```

	EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
1	201	Hartstein	13000	100
2	101	Kochhar	17000	100
3	102	De Haan	17000	100
4	124	Mourgos	5800	100
5	149	Zlotkey	10500	100
6	200	Whalen	4400	101
7	205	Higgins	12000	101
8	202	Fay	6000	201

ORACLE

2 - 9

Copyright © 2009, Oracle. All rights reserved.

## Membership Condition Using the IN Operator

To test for values in a specified set of values, use the IN operator. The condition defined using the IN operator is also known as the *membership condition*.

The slide example displays employee numbers, last names, salaries, and managers' employee numbers for all the employees whose manager's employee number is 100, 101, or 201.

**Note:** The set of values can be specified in any random order—for example, (201,100,101).

The IN operator can be used with any data type. The following example returns a row from the EMPLOYEES table, for any employee whose last name is included in the list of names in the WHERE clause:

```
SELECT employee_id, manager_id, department_id  
FROM   employees  
WHERE  last_name IN ('Hartstein', 'Vargas');
```

If characters or dates are used in the list, they must be enclosed with single quotation marks (' ').

**Note:** The IN operator is internally evaluated by the Oracle server as a set of OR conditions, such as a=value1 or a=value2 or a=value3. Therefore, using the IN operator has no performance benefits and is used only for logical simplicity.

## Pattern Matching Using the LIKE Operator

- Use the LIKE operator to perform wildcard searches of valid search string values.
- Search conditions can contain either literal characters or numbers:
  - % denotes zero or many characters.
  - \_ denotes one character.

```
SELECT first_name
FROM employees
WHERE first_name LIKE 'S%';
```

ORACLE

2 - 10

Copyright © 2009, Oracle. All rights reserved.

### Pattern Matching Using the LIKE Operator

You may not always know the exact value to search for. You can select rows that match a character pattern by using the LIKE operator. The character pattern-matching operation is referred to as a *wildcard* search. Two symbols can be used to construct the search string.

Symbol	Description
%	Represents any sequence of zero or more characters
_	Represents any single character

The SELECT statement in the slide returns the first name from the EMPLOYEES table for any employee whose first name begins with the letter “S.” Note the uppercase “S.” Consequently, names beginning with a lowercase “s” are not returned.

The LIKE operator can be used as a shortcut for some BETWEEN comparisons. The following example displays the last names and hire dates of all employees who joined between January, 1995 and December, 1995:

```
SELECT last_name, hire_date
FROM employees
WHERE hire_date LIKE '%95';
```

# Combining Wildcard Characters

- You can combine the two wildcard characters (%, \_) with literal characters for pattern matching:

```
SELECT last_name  
FROM   employees  
WHERE  last_name LIKE '_o%';
```

LAST_NAME
Kochhar
Lorentz
Mourgos

- You can use the ESCAPE identifier to search for the actual % and \_ symbols.

ORACLE

## Combining Wildcard Characters

The % and \_ symbols can be used in any combination with literal characters. The example in the slide displays the names of all employees whose last names have the letter “o” as the second character.

### ESCAPE Identifier

When you need to have an exact match for the actual % and \_ characters, use the ESCAPE identifier. This option specifies what the escape character is. If you want to search for strings that contain SA\_, you can use the following SQL statement:

```
SELECT employee_id, last_name, job_id  
FROM   employees WHERE job_id LIKE '%SA\_%' ESCAPE '\';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID
1	Zlotkey	SA_MAN
2	Abel	SA_REP
3	Taylor	SA_REP
4	Grant	SA_REP

The ESCAPE identifier identifies the backslash (\) as the escape character. In the SQL statement, the escape character precedes the underscore (\_). This causes the Oracle server to interpret the underscore literally.

# Using the NULL Conditions

Test for nulls with the IS NULL operator.

```
SELECT last_name, manager_id  
FROM employees  
WHERE manager_id IS NULL;
```

LAST_NAME	MANAGER_ID
King	(null)

ORACLE

## Using the NULL Conditions

The NULL conditions include the IS NULL condition and the IS NOT NULL condition.

The IS NULL condition tests for nulls. A null value means that the value is unavailable, unassigned, unknown, or inapplicable. Therefore, you cannot test with =, because a null cannot be equal or unequal to any value. The example in the slide retrieves the last names and managers of all employees who do not have a manager.

Here is another example: To display the last name, job ID, and commission for all employees who are *not* entitled to receive a commission, use the following SQL statement:

```
SELECT last_name, job_id, commission_pct  
FROM employees  
WHERE commission_pct IS NULL;
```

LAST_NAME	JOB_ID	COMMISSION_PCT
Whalen	AD_ASST	(null)
Hartstein	MK_MAN	(null)
Fay	MK_REP	(null)
Higgins	AC_MGR	(null)
Gietz	AC_ACCOUNT	(null)

...

# Defining Conditions Using the Logical Operators

Operator	Meaning
AND	Returns TRUE if <i>both</i> component conditions are true
OR	Returns TRUE if <i>either</i> component condition is true
NOT	Returns TRUE if the condition is false

ORACLE®

2 - 13

Copyright © 2009, Oracle. All rights reserved.

## Defining Conditions Using the Logical Operators

A logical condition combines the result of two component conditions to produce a single result based on those conditions or it inverts the result of a single condition. A row is returned only if the overall result of the condition is true.

Three logical operators are available in SQL:

- AND
- OR
- NOT

All the examples so far have specified only one condition in the WHERE clause. You can use several conditions in a single WHERE clause using the AND and OR operators.

## Using the AND Operator

AND requires both the component conditions to be true:

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
AND    job_id LIKE '%MAN' ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	201	Hartstein	MK_MAN	13000
2	149	Zlotkey	SA_MAN	10500

ORACLE

## Using the AND Operator

In the example, both the component conditions must be true for any record to be selected. Therefore, only those employees who have a job title that contains the string ‘MAN’ *and* earn \$10,000 or more are selected.

All character searches are case-sensitive, that is, no rows are returned if ‘MAN’ is not uppercase. Further, character strings must be enclosed with quotation marks.

### AND Truth Table

The following table shows the results of combining two expressions with AND:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

## Using the OR Operator

OR requires either component condition to be true:

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
OR    job_id LIKE '%MAN' ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	201	Hartstein	MK_MAN	13000
2	205	Higgins	AC_MGR	12000
3	100	King	AD_PRES	24000
4	101	Kochhar	AD_VP	17000
5	102	De Haan	AD_VP	17000
6	124	Mourgos	ST_MAN	5800
7	149	Zlotkey	SA_MAN	10500
8	174	Abel	SA_REP	11000

ORACLE

2 - 15

Copyright © 2009, Oracle. All rights reserved.

## Using the OR Operator

In the example, either component condition can be true for any record to be selected. Therefore, any employee who has a job ID that contains the string ‘MAN’ or earns \$10,000 or more is selected.

### OR Truth Table

The following table shows the results of combining two expressions with OR:

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

## Using the NOT Operator

```
SELECT last_name, job_id  
FROM employees  
WHERE job_id  
    NOT IN ('IT_PROG', 'ST_CLERK', 'SA REP') ;
```

	LAST_NAME	JOB_ID
1	De Haan	AD_VP
2	Fay	MK_REP
3	Gietz	AC_ACCOUNT
4	Hartstein	MK_MAN
5	Higgins	AC_MGR
6	King	AD_PRES
7	Kochhar	AD_VP
8	Mourgos	ST_MAN
9	Whalen	AD_ASST
10	Zlotkey	SA_MAN

ORACLE

2 - 16

Copyright © 2009, Oracle. All rights reserved.

## Using the NOT Operator

The example in the slide displays the last name and job ID of all employees whose job ID is not IT\_PROG, ST\_CLERK, or SA\_REP.

### NOT Truth Table

The following table shows the result of applying the NOT operator to a condition:

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

**Note:** The NOT operator can also be used with other SQL operators, such as BETWEEN, LIKE, and NULL.

```
... WHERE job_id NOT IN ('AC_ACCOUNT', 'AD_VP')  
... WHERE salary NOT BETWEEN 10000 AND 15000  
... WHERE last_name NOT LIKE '%A%'  
... WHERE commission_pct IS NOT NULL
```

# Rules of Precedence

Operator	Meaning
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Not equal to
7	NOT logical condition
8	AND logical condition
9	OR logical condition

You can use parentheses to override rules of precedence.

ORACLE

2 - 17

Copyright © 2009, Oracle. All rights reserved.

## Rules of Precedence

The rules of precedence determine the order in which expressions are evaluated and calculated. The table in the slide lists the default order of precedence. However, you can override the default order by using parentheses around the expressions that you want to calculate first.

# Rules of Precedence

```
SELECT last_name, job_id, salary  
FROM   employees  
WHERE  job_id = 'SA_REP'  
OR     job_id = 'AD_PRES'  
AND    salary > 15000;
```

1

	LAST_NAME	JOB_ID	SALARY
1	King	AD_PRES	24000
2	Abel	SA_REP	11000
3	Taylor	SA_REP	8600
4	Grant	SA_REP	7000

```
SELECT last_name, job_id, salary  
FROM   employees  
WHERE  (job_id = 'SA_REP'  
OR     job_id = 'AD_PRES')  
AND    salary > 15000;
```

2

	LAST_NAME	JOB_ID	SALARY
1	King	AD_PRES	24000

ORACLE

## Rules of Precedence (continued)

### 1. Precedence of the AND Operator: Example

In this example, there are two conditions:

- The first condition is that the job ID is AD\_PRES *and* the salary is greater than \$15,000.
- The second condition is that the job ID is SA\_REP.

Therefore, the SELECT statement reads as follows:

“Select the row if an employee is a president *and* earns more than \$15,000, *or* if the employee is a sales representative.”

### 2. Using Parentheses: Example

In this example, there are two conditions:

- The first condition is that the job ID is AD\_PRES *or* SA\_REP.
- The second condition is that the salary is greater than \$15,000.

Therefore, the SELECT statement reads as follows:

“Select the row if an employee is a president *or* a sales representative, *and* if the employee earns more than \$15,000.”

## Using the ORDER BY Clause

- Sort the retrieved rows with the ORDER BY clause:
  - ASC: Ascending order, default
  - DESC: Descending order
- The ORDER BY clause comes last in the SELECT statement:

```
SELECT    last_name, job_id, department_id, hire_date
FROM      employees
ORDER BY  hire_date ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
1 King	AD_PRES	90	17-JUN-87
2 Whalen	AD_ASST	10	17-SEP-87
3 Kochhar	AD_VP	90	21-SEP-89
4 Hunold	IT_PROG	60	03-JAN-90
5 Ernst	IT_PROG	60	21-MAY-91
6 De Haan	AD_VP	90	13-JAN-93

...

ORACLE

## Using the ORDER BY Clause

The order of rows that are returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. However, if you use the ORDER BY clause, it must be the last clause of the SQL statement. Further, you can specify an expression, an alias, or a column position as the sort condition.

### Syntax

```
SELECT      expr
FROM        table
[WHERE      condition(s)]
[ORDER BY  {column, expr, numeric_position} [ASC|DESC]];
```

In the syntax:

ORDER BY	specifies the order in which the retrieved rows are displayed
ASC	orders the rows in ascending order (This is the default order.)
DESC	orders the rows in descending order

If the ORDER BY clause is not used, the sort order is undefined, and the Oracle server may not fetch rows in the same order for the same query twice. Use the ORDER BY clause to display the rows in a specific order.

**Note:** Use the keywords NULLS FIRST or NULLS LAST to specify whether returned rows containing null values should appear first or last in the ordering sequence.

# Sorting

- Sorting in descending order:

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
ORDER BY hire_date DESC ;
```

1

- Sorting by column alias:

```
SELECT employee_id, last_name, salary*12 annsal
FROM employees
ORDER BY annsal ;
```

2

ORACLE

2 - 20

Copyright © 2009, Oracle. All rights reserved.

## Sorting

The default sort order is ascending:

- Numeric values are displayed with the lowest values first (for example, 1 to 999).
- Date values are displayed with the earliest value first (for example, 01-JAN-92 before 01-JAN-95).
- Character values are displayed in the alphabetical order (for example, “A” first and “Z” last).
- Null values are displayed last for ascending sequences and first for descending sequences.
- You can also sort by a column that is not in the SELECT list.

### Examples:

1. To reverse the order in which the rows are displayed, specify the DESC keyword after the column name in the ORDER BY clause. The example in the slide sorts the result by the most recently hired employee.
2. You can also use a column alias in the ORDER BY clause. The slide example sorts the data by annual salary.

**Note:** The DESC keyword used here for sorting in descending order should not be confused with the DESC keyword used to describe table structures.

# Sorting

- Sorting by using the column's numeric position:

```
SELECT last_name, job_id, department_id, hire_date  
FROM employees  
ORDER BY 3;
```



- Sorting by multiple columns:

```
SELECT last_name, department_id, salary  
FROM employees  
ORDER BY department_id, salary DESC;
```



ORACLE

2 - 21

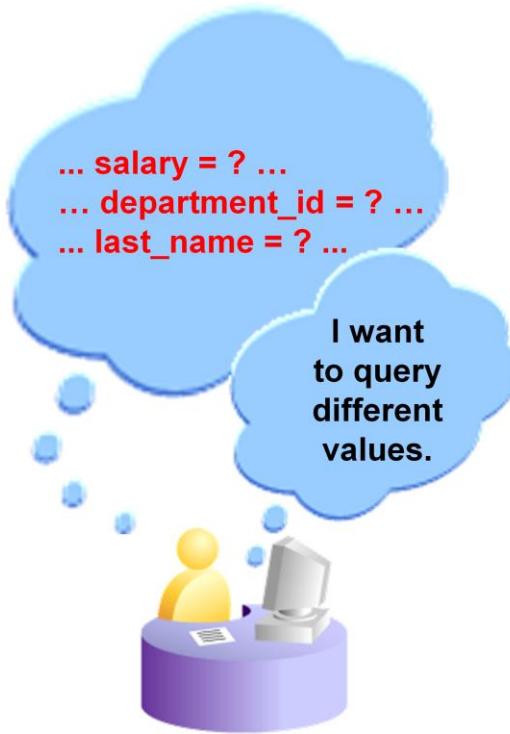
Copyright © 2009, Oracle. All rights reserved.

## Sorting (continued)

### Examples:

3. You can sort query results by specifying the numeric position of the column in the SELECT clause. The example in the slide sorts the result by the `department_id` as this column is at the third position in the SELECT clause.
4. You can sort query results by more than one column. The sort limit is the number of columns in the given table. In the ORDER BY clause, specify the columns and separate the column names using commas. If you want to reverse the order of a column, specify `DESC` after its name. The result of the query example shown in the slide is sorted by `department_id` in ascending order and also by `salary` in descending order.

# Substitution Variables



ORACLE

2 - 22

Copyright © 2009, Oracle. All rights reserved.

## Substitution Variables

So far, all the SQL statements were executed with predetermined columns, conditions, and their values. Suppose that you want a query that lists the employees with various jobs and not just those whose job\_ID is SA REP. You can edit the WHERE clause to provide a different value each time you run the command, but there is also an easier way.

By using a substitution variable in place of the exact values in the WHERE clause, you can run the same query for different values.

You can create reports that prompt users to supply their own values to restrict the range of data returned, by using substitution variables. You can embed *substitution variables* in a command file or in a single SQL statement. A variable can be thought of as a container in which values are temporarily stored. When the statement is run, the stored value is substituted.

# Substitution Variables

- Use substitution variables to:
  - Temporarily store values with single-ampersand (&) and double-ampersand (&&) substitution
- Use substitution variables to supplement the following:
  - WHERE conditions
  - ORDER BY clauses
  - Column expressions
  - Table names
  - Entire SELECT statements

ORACLE

2 - 23

Copyright © 2009, Oracle. All rights reserved.

## Substitution Variables (continued)

You can use single-ampersand (&) substitution variables to temporarily store values.

You can also predefine variables by using the DEFINE command. DEFINE creates and assigns a value to a variable.

### Restricted Ranges of Data: Examples

- Reporting figures only for the current quarter or specified date range
- Reporting on data relevant only to the user requesting the report
- Displaying personnel only within a given department

### Other Interactive Effects

Interactive effects are not restricted to direct user interaction with the WHERE clause. The same principles can also be used to achieve other goals, such as:

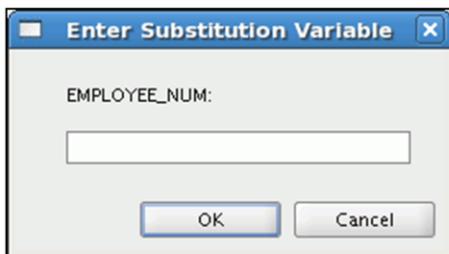
- Obtaining input values from a file rather than from a person
- Passing values from one SQL statement to another

**Note:** Both SQL Developer and SQL\* Plus support substitution variables and the DEFINE/UNDEFINE commands. Neither SQL Developer nor SQL\* Plus support validation checks (except for data type) on user input. If used in scripts that are deployed to users, substitution variables can be subverted for SQL injection attacks.

# Using the Single-Ampersand Substitution Variable

Use a variable prefixed with an ampersand (&) to prompt the user for a value:

```
SELECT employee_id, last_name, salary, department_id  
FROM   employees  
WHERE  employee_id = &employee_num ;
```



ORACLE

2 - 24

Copyright © 2009, Oracle. All rights reserved.

## Using the Single-Ampersand Substitution Variable

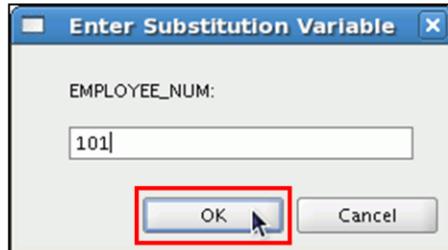
When running a report, users often want to restrict the data that is returned dynamically. SQL\*Plus or SQL Developer provides this flexibility with user variables. Use an ampersand (&) to identify each variable in your SQL statement. However, you do not need to define the value of each variable.

Notation	Description
<code>&amp;user_variable</code>	Indicates a variable in a SQL statement; if the variable does not exist, SQL*Plus or SQL Developer prompts the user for a value (the new variable is discarded after it is used.)

The example in the slide creates a SQL Developer substitution variable for an employee number. When the statement is executed, SQL Developer prompts the user for an employee number and then displays the employee number, last name, salary, and department number for that employee.

With the single ampersand, the user is prompted every time the command is executed if the variable does not exist.

# Using the Single-Ampersand Substitution Variable



	EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
1	101	Kochhar	17000	90

ORACLE

## Using the Single-Ampersand Substitution Variable (continued)

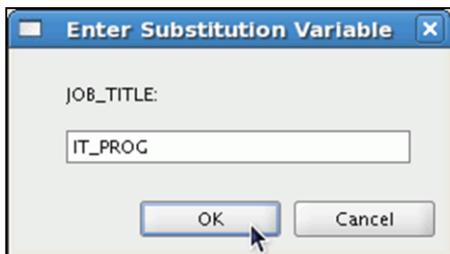
When SQL Developer detects that the SQL statement contains an ampersand, you are prompted to enter a value for the substitution variable that is named in the SQL statement.

After you enter a value and click the OK button, the results are displayed in the Results tab of your SQL Developer session.

## Character and Date Values with Substitution Variables

Use single quotation marks for date and character values:

```
SELECT last_name, department_id, salary*12
FROM   employees
WHERE  job_id = '&job_title' ;
```



	LAST_NAME	DEPARTMENT_ID	SALARY*12
1	Hunold	60	108000
2	Ernst	60	72000
3	Lorentz	60	50400

ORACLE

### Character and Date Values with Substitution Variables

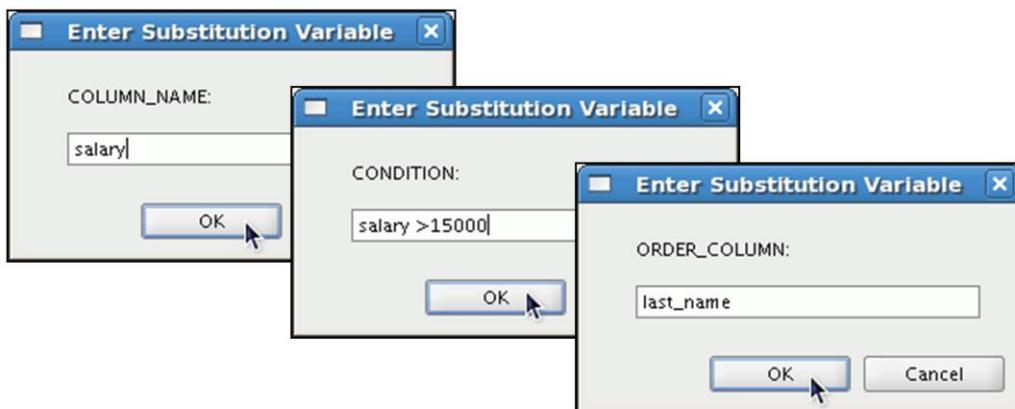
In a WHERE clause, date and character values must be enclosed with single quotation marks. The same rule applies to the substitution variables.

Enclose the variable with single quotation marks within the SQL statement itself.

The slide shows a query to retrieve the employee names, department numbers, and annual salaries of all employees based on the job title value of the SQL Developer substitution variable.

# Specifying Column Names, Expressions, and Text

```
SELECT employee_id, last_name, job_id, &column_name  
FROM employees  
WHERE &condition  
ORDER BY &order_column ;
```



ORACLE

2 - 27

Copyright © 2009, Oracle. All rights reserved.

## Specifying Column Names, Expressions, and Text

You can use the substitution variables not only in the WHERE clause of a SQL statement, but also as substitution for column names, expressions, or text.

### Example:

The example in the slide displays the employee number, last name, job title, and any other column that is specified by the user at run time, from the EMPLOYEES table. For each substitution variable in the SELECT statement, you are prompted to enter a value, and then click OK to proceed.

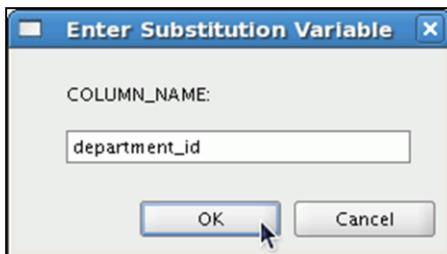
If you do not enter a value for the substitution variable, you get an error when you execute the preceding statement.

**Note:** A substitution variable can be used anywhere in the SELECT statement, except as the first word entered at the command prompt.

## Using the Double-Ampersand Substitution Variable

Use double ampersand (`&&`) if you want to reuse the variable value without prompting the user each time:

```
SELECT employee_id, last_name, job_id, &&column_name  
FROM employees  
ORDER BY &&column_name ;
```



	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	200	Whalen	AD_ASST	10
2	201	Hartstein	MK_MAN	20
3	202	Fay	MK_REP	20

...

ORACLE

### Using the Double-Ampersand Substitution Variable

You can use the double-ampersand (`&&`) substitution variable if you want to reuse the variable value without prompting the user each time. The user sees the prompt for the value only once. In the example in the slide, the user is asked to give the value for the variable, `column_name`, only once. The value that is supplied by the user (`department_id`) is used for both display and ordering of data. If you run the query again, you will not be prompted for the value of the variable.

SQL Developer stores the value that is supplied by using the `DEFINE` command; it uses it again whenever you reference the variable name. After a user variable is in place, you need to use the `UNDEFINE` command to delete it:

```
UNDEFINE column_name
```

## Using the DEFINE Command

- Use the `DEFINE` command to create and assign a value to a variable.
- Use the `UNDEFINE` command to remove a variable.

```
DEFINE employee_num = 200
SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num ;
UNDEFINE employee_num
```

ORACLE

2 - 29

Copyright © 2009, Oracle. All rights reserved.

### Using the `DEFINE` Command

The example shown creates a substitution variable for an employee number by using the `DEFINE` command. At run time, this displays the employee number, name, salary, and department number for that employee.

Because the variable is created using the SQL Developer `DEFINE` command, the user is not prompted to enter a value for the employee number. Instead, the defined variable value is automatically substituted in the `SELECT` statement.

The `EMPLOYEE_NUM` substitution variable is present in the session until the user undefines it or exits the SQL Developer session.

## Using the VERIFY Command

Use the VERIFY command to toggle the display of the substitution variable, both before and after SQL Developer replaces substitution variables with values:

The screenshot shows the Oracle SQL Developer interface. In the SQL Worksheet, the command `SET VERIFY ON` is highlighted with a red box. Below it is a query to select employee\_id, last\_name, and salary from the employees table where employee\_id equals &employee\_num. In the Script Output tab, the input is shown with the substitution variable &employee\_num replaced by the value 200. The output shows one row selected: EMPLOYEE\_ID 200, LAST\_NAME Whalen, and SALARY 4400.

```
SET VERIFY ON
SELECT employee_id, last_name, salary
FROM employees
WHERE employee_id = &employee_num;
```

EMPLOYEE_ID	LAST_NAME	SALARY
200	Whalen	4400

ORACLE

### Using the VERIFY Command

To confirm the changes in the SQL statement, use the VERIFY command. Setting SET VERIFY ON forces SQL Developer to display the text of a command after it replaces substitution variables with values. To see the VERIFY output, you should use the Run Script (F5) icon in the SQL Worksheet. SQL Developer displays the text of a command after it replaces substitution variables with values, in the Script Output tab as shown in the slide.

The example in the slide displays the new value of the EMPLOYEE\_ID column in the SQL statement followed by the output.

### SQL\*Plus System Variables

SQL\*Plus uses various system variables that control the working environment. One of the variables is VERIFY. To obtain a complete list of all the system variables, you can issue the SHOW ALL command on the SQL\*Plus command prompt.

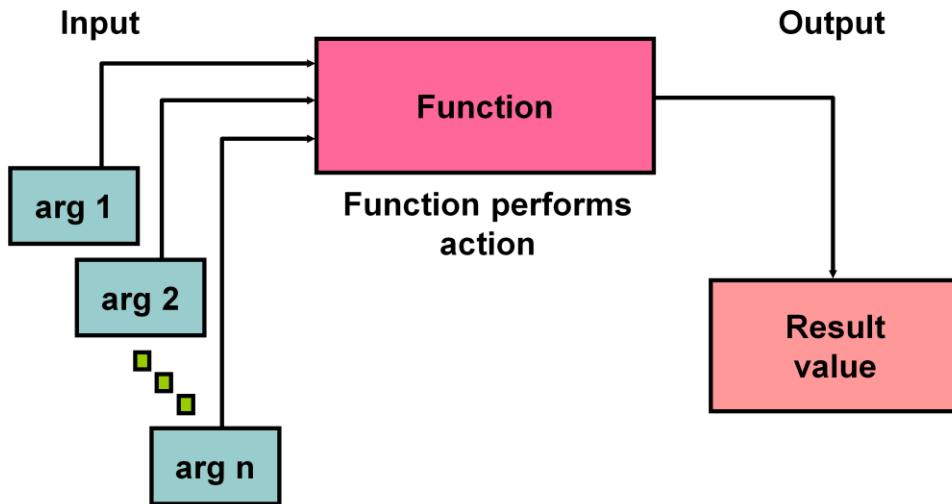


## **Using Single-Row Functions to Customize Output**

**ORACLE®**

Copyright © 2009, Oracle. All rights reserved.

# SQL Functions



ORACLE

3 - 2

Copyright © 2009, Oracle. All rights reserved.

## SQL Functions

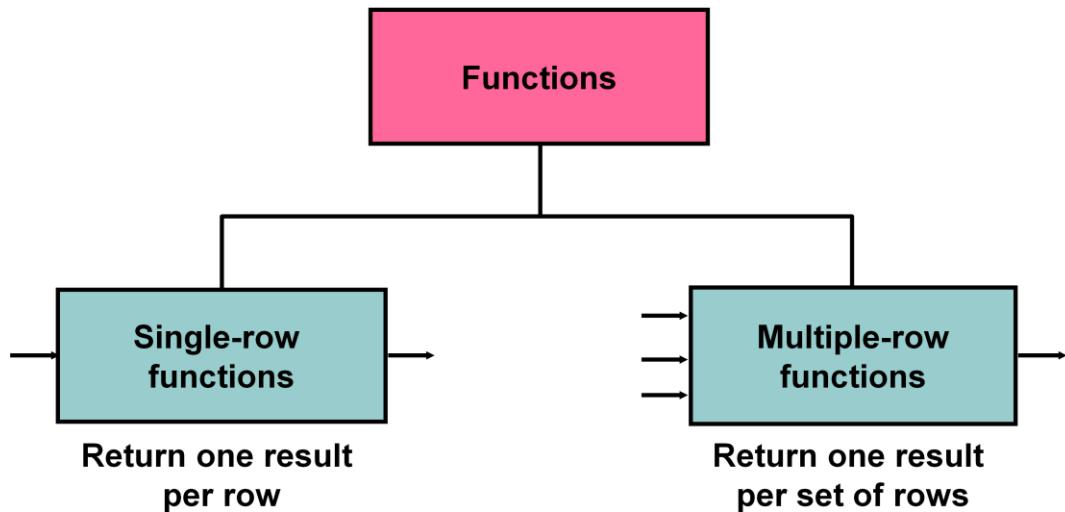
Functions are a very powerful feature of SQL. They can be used to do the following:

- Perform calculations on data
- Modify individual data items
- Manipulate output for groups of rows
- Format dates and numbers for display
- Convert column data types

SQL functions sometimes take arguments and always return a value.

**Note:** If you want to know whether a function is a SQL:2003 compliant function, refer to the *Oracle Compliance To Core SQL:2003* section in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

## Two Types of SQL Functions



3 - 3

Copyright © 2009, Oracle. All rights reserved.

ORACLE

## Two Types of SQL Functions

There are two types of functions:

- Single-row functions
- Multiple-row functions

### Single-Row Functions

These functions operate on single rows only and return one result per row. There are different types of single-row functions. This lesson covers the following functions:

- Character
- Number
- Date
- Conversion
- General

### Multiple-Row Functions

Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as *group functions* (covered in the lesson titled “Reporting Aggregated Data Using the Group Functions”).

**Note:** For more information and a complete list of available functions and their syntax, see the section on “Functions” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

# Single-Row Functions

Single-row functions:

- Manipulate data items
- Accept arguments and return one value
- Act on each row that is returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments that can be a column or an expression

```
function_name [(arg1, arg2, . . .)]
```

ORACLE

3 - 4

Copyright © 2009, Oracle. All rights reserved.

## Single-Row Functions

Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row that is returned by the query. An argument can be one of the following:

- User-supplied constant
- Variable value
- Column name
- Expression

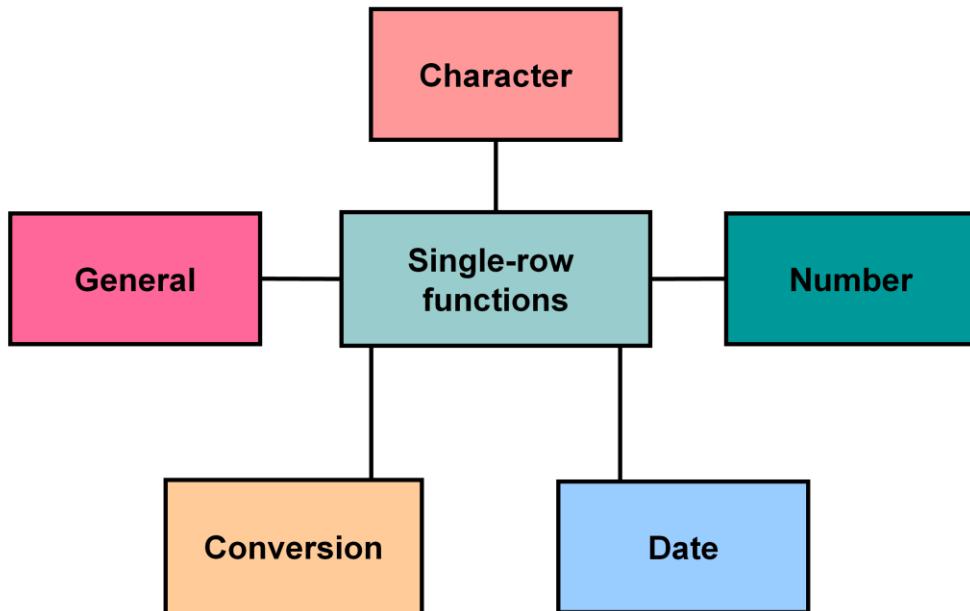
Features of single-row functions include:

- Acting on each row that is returned in the query
- Returning one result per row
- Possibly returning a data value of a different type than the one that is referenced
- Possibly expecting one or more arguments
- Can be used in SELECT, WHERE, and ORDER BY clauses; can be nested

In the syntax:

<i>function_name</i>	Is the name of the function
<i>arg1, arg2</i>	Is any argument to be used by the function. This can be represented by a column name or expression.

# Single-Row Functions



ORACLE

3 - 5

Copyright © 2009, Oracle. All rights reserved.

## Single-Row Functions (continued)

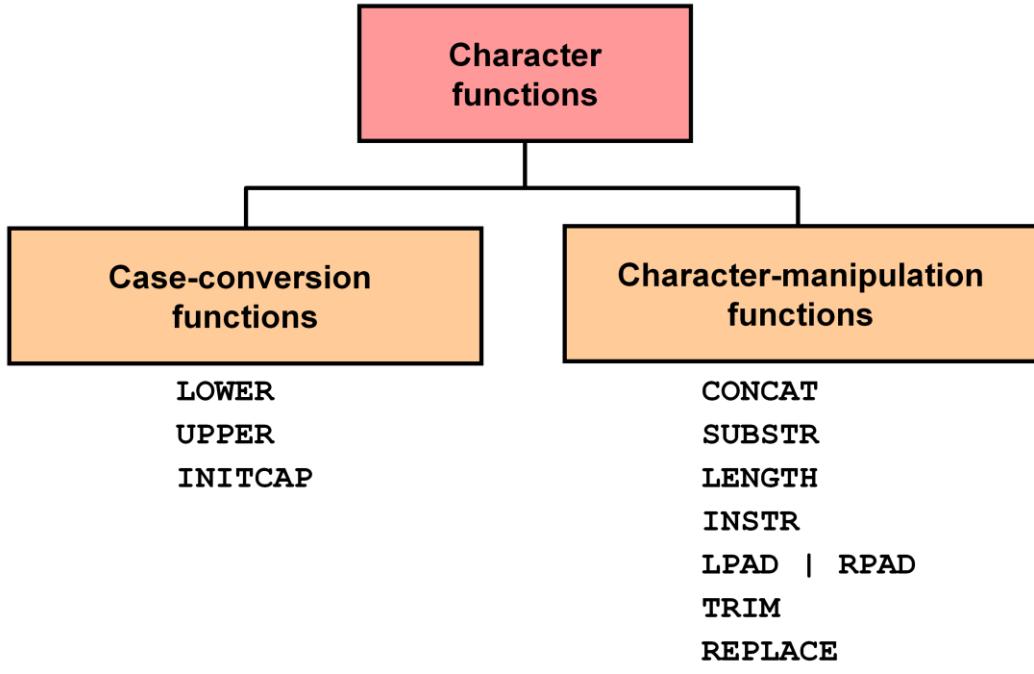
This lesson covers the following single-row functions:

- **Character functions:** Accept character input and can return both character and number values
- **Number functions:** Accept numeric input and return numeric values
- **Date functions:** Operate on values of the DATE data type (All date functions return a value of the DATE data type except the MONTHS\_BETWEEN function, which returns a number.)

The following single-row functions are discussed in the lesson titled “Using Conversion Functions and Conditional Expressions”:

- **Conversion functions:** Convert a value from one data type to another
- **General functions:**
  - NVL
  - NVL2
  - NULLIF
  - COALESCE
  - CASE
  - DECODE

# Character Functions



ORACLE

3 - 6

Copyright © 2009, Oracle. All rights reserved.

## Character Functions

Single-row character functions accept character data as input and can return both character and numeric values. Character functions can be divided into the following:

- Case-conversion functions
- Character-manipulation functions

Function	Purpose
LOWER ( <i>column expression</i> )	Converts alpha character values to lowercase
UPPER ( <i>column expression</i> )	Converts alpha character values to uppercase
INITCAP ( <i>column expression</i> )	Converts alpha character values to uppercase for the first letter of each word; all other letters in lowercase
CONCAT ( <i>column1 expression1, column2 expression2</i> )	Concatenates the first character value to the second character value; equivalent to concatenation operator (  )
SUBSTR ( <i>column expression, m[, n]</i> )	Returns specified characters from character value starting at character position <i>m</i> , <i>n</i> characters long (If <i>m</i> is negative, the count starts from the end of the character value. If <i>n</i> is omitted, all characters to the end of the string are returned.)



# Case-Conversion Functions

These functions convert the case for character strings:

Function	Result
LOWER('SQL Course')	sql course
UPPER('SQL Course')	SQL COURSE
INITCAP('SQL Course')	Sql Course

ORACLE

3 - 7

Copyright © 2009, Oracle. All rights reserved.

## Case-Conversion Functions

LOWER, UPPER, and INITCAP are the three case-conversion functions.

- LOWER: Converts mixed-case or uppercase character strings to lowercase
- UPPER: Converts mixed-case or lowercase character strings to uppercase
- INITCAP: Converts the first letter of each word to uppercase and the remaining letters to lowercase

```
SELECT 'The job id for '||UPPER(last_name)||' is '  
      ||LOWER(job_id) AS "EMPLOYEE DETAILS"  
FROM   employees;
```

EMPLOYEE DETAILS
1 The job id for ABEL is sa_rep
2 The job id for DAVIES is st_clerk
3 The job id for DE HAAN is ad_vp
4 The job id for ERNST is it_prog
5 The job id for FAY is mk_rep
6 The job id for GIETZ is ac_account

...

# Using Case-Conversion Functions

Display the employee number, name, and department number for employee Higgins:

```
SELECT employee_id, last_name, department_id  
FROM   employees  
WHERE  last_name = 'higgins';  
0 rows selected
```

```
SELECT employee_id, last_name, department_id  
FROM   employees  
WHERE  LOWER(last_name) = 'higgins';
```

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	205	Higgins	110

ORACLE

3 - 8

Copyright © 2009, Oracle. All rights reserved.

## Using Case-Conversion Functions

The slide example displays the employee number, name, and department number of employee Higgins.

The WHERE clause of the first SQL statement specifies the employee name as higgins. Because all the data in the EMPLOYEES table is stored in proper case, the name higgins does not find a match in the table, and no rows are selected.

The WHERE clause of the second SQL statement specifies that the employee name in the EMPLOYEES table is compared to higgins, converting the LAST\_NAME column to lowercase for comparison purposes. Because both names are now lowercase, a match is found and one row is selected. The WHERE clause can be rewritten in the following manner to produce the same result:

```
...WHERE last_name = 'Higgins'
```

The name in the output appears as it was stored in the database. To display the name in uppercase, use the UPPER function in the SELECT statement.

```
SELECT employee_id, UPPER(last_name), department_id  
FROM   employees  
WHERE  INITCAP(last_name) = 'Higgins';
```

# Character-Manipulation Functions

These functions manipulate character strings:

Function	Result
CONCAT('Hello', 'World')	HelloWorld
SUBSTR('HelloWorld', 1, 5)	Hello
LENGTH('HelloWorld')	10
INSTR('HelloWorld', 'W')	6
LPAD(salary, 10, '*')	*****24000
RPAD(salary, 10, '*')	24000*****
REPLACE ('JACK and JUE', 'J', 'BL')	BLACK and BLUE
TRIM('H' FROM 'HelloWorld')	elloWorld

ORACLE

3 - 9

Copyright © 2009, Oracle. All rights reserved.

## Character-Manipulation Functions

CONCAT, SUBSTR, LENGTH, INSTR, LPAD, RPAD, and TRIM are the character-manipulation functions that are covered in this lesson.

- CONCAT: Joins values together (You are limited to using two parameters with CONCAT.)
- SUBSTR: Extracts a string of determined length
- LENGTH: Shows the length of a string as a numeric value
- INSTR: Finds the numeric position of a named character
- LPAD: Returns an expression left-padded to the length of *n* characters with a character expression
- RPAD: Returns an expression right-padded to the length of *n* characters with a character expression
- TRIM: Trims leading or trailing characters (or both) from a character string (If *trim\_character* or *trim\_source* is a character literal, you must enclose it within single quotation marks.)

**Note:** You can use functions such as UPPER and LOWER with ampersand substitution. For example, use UPPER('&job\_title') so that the user does not have to enter the job title in a specific case.

# Using the Character-Manipulation Functions

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,
       job_id, LENGTH(last_name),
       INSTR(last_name, 'a') "Contains 'a'?"
  FROM employees
 WHERE SUBSTR(job_id, 4) = 'REP';
```

EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
202	PatFay	MK_REP	3	2
174	EllenAbel	SA_REP	4	0
176	JonathonTaylor	SA_REP	6	2
178	KimberelyGrant	SA_REP	5	3

1

2

3

1

2

3

ORACLE

## Using the Character-Manipulation Functions

The example in the slide displays employee first names and last names joined together, the length of the employee last name, and the numeric position of the letter “a” in the employee last name for all employees who have the string, REP, contained in the job ID starting at the fourth position of the job ID.

### Example:

Modify the SQL statement in the slide to display the data for those employees whose last names end with the letter “n.”

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,
       LENGTH(last_name), INSTR(last_name, 'a') "Contains 'a'?"
  FROM employees
 WHERE SUBSTR(last_name, -1, 1) = 'n';
```

EMPLOYEE_ID	NAME	LENGTH(LAST_NAME)	Contains 'a'?
102	LexDe Haan	7	5
200	JenniferWhalen	6	3
201	MichaelHartstein	9	2

# Number Functions

- ROUND: Rounds value to a specified decimal
- TRUNC: Truncates value to a specified decimal
- MOD: Returns remainder of division

Function	Result
ROUND(45.926, 2)	45.93
TRUNC(45.926, 2)	45.92
MOD(1600, 300)	100

ORACLE

3 - 11

Copyright © 2009, Oracle. All rights reserved.

## Number Functions

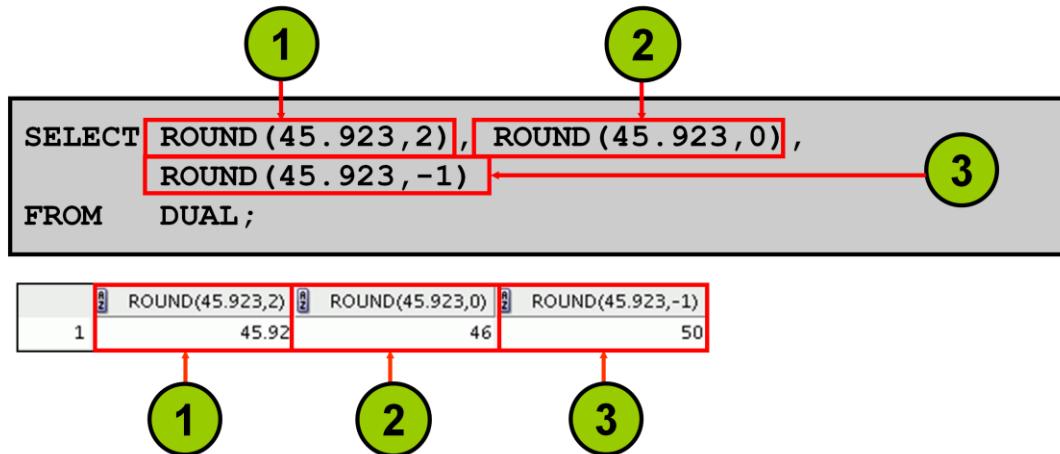
Number functions accept numeric input and return numeric values. This section describes some of the number functions.

Function	Purpose
ROUND( <i>column expression, n</i> )	Rounds the column, expression, or value to <i>n</i> decimal places or, if <i>n</i> is omitted, no decimal places (If <i>n</i> is negative, numbers to the left of decimal point are rounded.)
TRUNC( <i>column expression, n</i> )	Truncates the column, expression, or value to <i>n</i> decimal places or, if <i>n</i> is omitted, <i>n</i> defaults to zero
MOD( <i>m, n</i> )	Returns the remainder of <i>m</i> divided by <i>n</i>

**Note:** This list contains only some of the available number functions.

For more information, see the section on “Numeric Functions” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

## Using the ROUND Function



DUAL is a public table that you can use to view results from functions and calculations.

ORACLE

3 - 12

Copyright © 2009, Oracle. All rights reserved.

## Using the ROUND Function

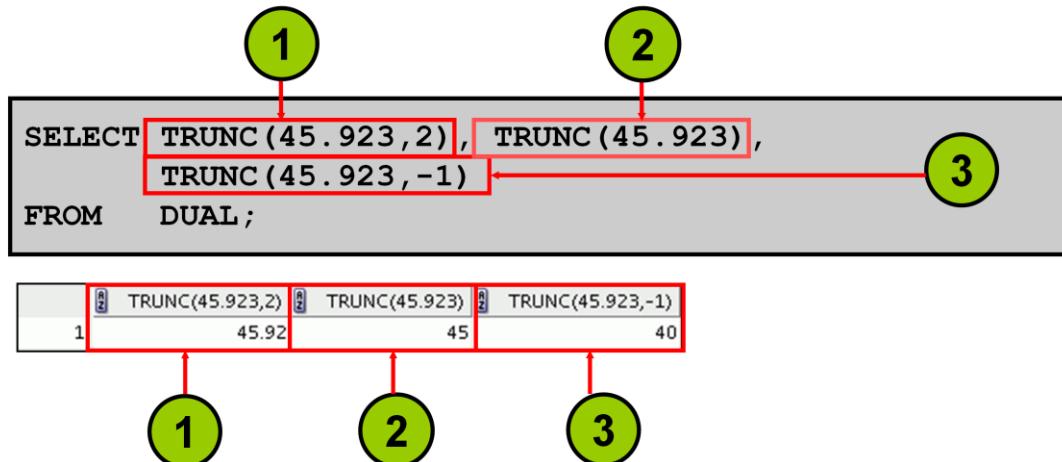
The ROUND function rounds the column, expression, or value to  $n$  decimal places. If the second argument is 0 or is missing, the value is rounded to zero decimal places. If the second argument is 2, the value is rounded to two decimal places. Conversely, if the second argument is  $-2$ , the value is rounded to two decimal places to the left (rounded to the nearest unit of 100).

The ROUND function can also be used with date functions. You will see examples later in this lesson.

### DUAL Table

The DUAL table is owned by the user SYS and can be accessed by all users. It contains one column, DUMMY, and one row with the value X. The DUAL table is useful when you want to return a value only once (for example, the value of a constant, pseudocolumn, or expression that is not derived from a table with user data). The DUAL table is generally used for completeness of the SELECT clause syntax, because both SELECT and FROM clauses are mandatory, and several calculations do not need to select from the actual tables.

## Using the TRUNC Function



ORACLE

3 - 13

Copyright © 2009, Oracle. All rights reserved.

## Using the TRUNC Function

The TRUNC function truncates the column, expression, or value to  $n$  decimal places.

The TRUNC function works with arguments similar to those of the ROUND function. If the second argument is 0 or is missing, the value is truncated to zero decimal places. If the second argument is 2, the value is truncated to two decimal places. Conversely, if the second argument is  $-2$ , the value is truncated to two decimal places to the left. If the second argument is  $-1$ , the value is truncated to one decimal place to the left.

Like the ROUND function, the TRUNC function can be used with date functions.

## Using the MOD Function

For all employees with the job title of Sales Representative, calculate the remainder of the salary after it is divided by 5,000.

```
SELECT last_name, salary, MOD(salary, 5000)
FROM   employees
WHERE  job_id = 'SA REP';
```

	LAST_NAME	SALARY	MOD(SALARY,5000)
1	Abel	11000	1000
2	Taylor	8600	3600
3	Grant	7000	2000

ORACLE

## Using the MOD Function

The MOD function finds the remainder of the first argument divided by the second argument. The slide example calculates the remainder of the salary after dividing it by 5,000 for all employees whose job ID is SA REP.

**Note:** The MOD function is often used to determine whether a value is odd or even. The MOD function is also the Oracle hash function.

## Working with Dates

- The Oracle Database stores dates in an internal numeric format: century, year, month, day, hours, minutes, and seconds.
- The default date display format is DD-MON-RR.
  - Enables you to store 21st-century dates in the 20th century by specifying only the last two digits of the year
  - Enables you to store 20th-century dates in the 21st century in the same way

```
SELECT last_name, hire_date  
FROM employees  
WHERE hire_date < '01-FEB-88';
```

LAST_NAME	HIRE_DATE
Whalen	17-SEP-87
King	17-JUN-87

ORACLE

## Working with Dates

The Oracle Database stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds.

The default display and input format for any date is DD-MON-RR. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D.

In the example in the slide, the HIRE\_DATE column output is displayed in the default format DD-MON-RR. However, dates are not stored in the database in this format. All the components of the date and time are stored. So, although a HIRE\_DATE such as 17-JUN-87 is displayed as day, month, and year, there is also *time* and *century* information associated with the date. The complete data might be June 17, 1987, 5:10:43 PM.

## RR Date Format

Current Year	Specified Date	RR Format	YY Format
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2001	27-OCT-95	1995	2095

		If the specified two-digit year is:	
		0–49	50–99
If two digits of the current year are:	0–49	The return date is in the current century	The return date is in the century before the current one
	50–99	The return date is in the century after the current one	The return date is in the current century

ORACLE

3 - 16

Copyright © 2009, Oracle. All rights reserved.

## RR Date Format

The RR date format is similar to the YY element, but you can use it to specify different centuries. Use the RR date format element instead of YY so that the century of the return value varies according to the specified two-digit year and the last two digits of the current year. The table in the slide summarizes the behavior of the RR element.

Current Year	Given Date	Interpreted (RR)	Interpreted (YY)
1994	27-OCT-95	1995	1995
1994	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2048	27-OCT-52	1952	2052
2051	27-OCT-47	2147	2047

Note the values shown in the last two rows of the above table. As we approach the middle of the century, then the RR behavior is probably not what you want.

# Using the SYSDATE Function

SYSDATE is a function that returns:

- Date
- Time

```
SELECT sysdate  
FROM   dual;
```

1	SYSDATE
1	10-JUN-09

ORACLE

## Using the SYSDATE Function

SYSDATE is a date function that returns the current database server date and time. You can use SYSDATE just as you would use any other column name. For example, you can display the current date by selecting SYSDATE from a table. It is customary to select SYSDATE from a public table called DUAL.

**Note:** SYSDATE returns the current date and time set for the operating system on which the database resides. Therefore, if you are in a place in Australia and connected to a remote database in a location in the United States (U.S.), the sysdate function will return the U.S. date and time. In that case, you can use the CURRENT\_DATE function that returns the current date in the session time zone.

The CURRENT\_DATE function and other related time zone functions are discussed in detail in the course titled *Oracle Database 11g: SQL Fundamentals II*.

## Arithmetic with Dates

- Add or subtract a number to or from a date for a resultant date value.
- Subtract two dates to find the number of days between those dates.
- Add hours to a date by dividing the number of hours by 24.

ORACLE

3 - 19

Copyright © 2009, Oracle. All rights reserved.

### Arithmetic with Dates

Because the database stores dates as numbers, you can perform calculations using arithmetic operators such as addition and subtraction. You can add and subtract number constants as well as dates.

You can perform the following operations:

Operation	Result	Description
date + number	Date	Adds a number of days to a date
date – number	Date	Subtracts a number of days from a date
date – date	Number of days	Subtracts one date from another
date + number/24	Date	Adds a number of hours to a date

# Using Arithmetic Operators with Dates

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS  
FROM employees  
WHERE department_id = 90;
```

	LAST_NAME	WEEKS
1	King	1147.102432208994708994708994708995
2	Kochhar	1028.959575066137566137566137566138
3	De Haan	856.102432208994708994708994708995

ORACLE

3 - 20

Copyright © 2009, Oracle. All rights reserved.

## Using Arithmetic Operators with Dates

The example in the slide displays the last name and the number of weeks employed for all employees in department 90. It subtracts the date on which the employee was hired from the current date (SYSDATE) and divides the result by 7 to calculate the number of weeks that a worker has been employed.

**Note:** SYSDATE is a SQL function that returns the current date and time. Your results may differ depending on the date and time set for the operating system of your local database when you run the SQL query.

If a more current date is subtracted from an older date, the difference is a negative number.

# Date-Manipulation Functions

Function	Result
MONTHS_BETWEEN	Number of months between two dates
ADD_MONTHS	Add calendar months to date
NEXT_DAY	Next day of the date specified
LAST_DAY	Last day of the month
ROUND	Round date
TRUNC	Truncate date

ORACLE

## Date-Manipulation Functions

Date functions operate on Oracle dates. All date functions return a value of the DATE data type except MONTHS\_BETWEEN, which returns a numeric value.

- MONTHS\_BETWEEN(*date1*, *date2*) : Finds the number of months between *date1* and *date2*. The result can be positive or negative. If *date1* is later than *date2*, the result is positive; if *date1* is earlier than *date2*, the result is negative. The noninteger part of the result represents a portion of the month.
- ADD\_MONTHS(*date*, *n*) : Adds *n* number of calendar months to *date*. The value of *n* must be an integer and can be negative.
- NEXT\_DAY(*date*, '*char*') : Finds the date of the next specified day of the week ('*char*') following *date*. The value of *char* may be a number representing a day or a character string.
- LAST\_DAY(*date*) : Finds the date of the last day of the month that contains *date*

The above list is a subset of the available date functions. ROUND and TRUNC number functions can also be used to manipulate the date values as shown below:

- ROUND(*date*[, '*fmt*']) : Returns *date* rounded to the unit that is specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is rounded to the nearest day.
- TRUNC(*date*[, '*fmt*']) : Returns *date* with the time portion of the day truncated to the unit that is specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is truncated to the nearest day.

# Using Date Functions

Function	Result
MONTHS_BETWEEN ( '01-SEP-95' , '11-JAN-94' )	19.6774194
ADD_MONTHS ( '31-JAN-96' , 1)	'29-FEB-96'
NEXT_DAY ('01-SEP-95' , 'FRIDAY')	'08-SEP-95'
LAST_DAY ('01-FEB-95')	'28-FEB-95'

ORACLE

## Using Date Functions

In the example in the slide, the ADD\_MONTHS function adds one month to the supplied date value “31-JAN-96” and returns “29-FEB-96.” The function recognizes the year 1996 as the leap year and, therefore, returns the last day of the February month. If you change the input date value to “31-JAN-95,” the function returns “28-FEB-95.”

For example, display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and the last day of the hire month for all employees who have been employed for fewer than 150 months.

```
SELECT employee_id, hire_date, MONTHS_BETWEEN (SYSDATE, hire_date) TENURE,  
ADD_MONTHS (hire_date, 6) REVIEW, NEXT_DAY (hire_date, 'FRIDAY'),  
LAST_DAY (hire_date)  
FROM employees WHERE MONTHS_BETWEEN (SYSDATE, hire_date) < 150;
```

#	EMPLOYEE_ID	HIRE_DATE	TENURE	REVIEW	NEXT_DA...	LAST_DAY...
1	202	17-AUG-97	141.79757989...	17-FEB-98	22-AUG-97	31-AUG-97
2	107	07-FEB-99	124.12016054...	07-AUG-99	12-FEB-99	28-FEB-99
3	124	16-NOV-99	114.82983796...	16-MAY-00	19-NOV-99	30-NOV-99
4	142	29-JAN-97	148.41048312...	29-JUL-97	31-JAN-97	31-JAN-97
5	143	15-MAR-98	134.86209602...	15-SEP-98	20-MAR-98	31-MAR-98
6	144	09-JUL-98	131.05564441...	09-JAN-99	10-JUL-98	31-JUL-98
7	149	29-JAN-00	112.41048312...	29-JUL-00	04-FEB-00	31-JAN-00
8	176	24-MAR-98	134.57177344...	24-SEP-98	27-MAR-98	31-MAR-98
9	178	24-MAY-99	120.57177344...	24-NOV-99	28-MAY-99	31-MAY-99

## Using ROUND and TRUNC Functions with Dates

Assume SYSDATE = '25-JUL-03':

Function	Result
ROUND(SYSDATE, 'MONTH')	01-AUG-03
ROUND(SYSDATE, 'YEAR')	01-JAN-04
TRUNC(SYSDATE, 'MONTH')	01-JUL-03
TRUNC(SYSDATE, 'YEAR')	01-JAN-03

ORACLE

3 - 23

Copyright © 2009, Oracle. All rights reserved.

## Using ROUND and TRUNC Functions with Dates

The ROUND and TRUNC functions can be used for number and date values. When used with dates, these functions round or truncate to the specified format model. Therefore, you can round dates to the nearest year or month. If the format model is month, dates 1-15 result in the first day of the current month. Dates 16-31 result in the first day of the next month. If the format model is year, months 1-6 result in January 1 of the current year. Months 7-12 result in January 1 of the next year.

### Example:

Compare the hire dates for all employees who started in 1997. Display the employee number, hire date, and starting month using the ROUND and TRUNC functions.

```
SELECT employee_id, hire_date,
       ROUND(hire_date, 'MONTH'), TRUNC(hire_date, 'MONTH')
  FROM   employees
 WHERE  hire_date LIKE '%97';
```

	EMPLOYEE_ID	HIRE_DATE	ROUND(HIRE_DATE,'MONTH')	TRUNC(HIRE_DATE,'MONTH')
1	202	17-AUG-97	01-SEP-97	01-AUG-97
2	142	29-JAN-97	01-FEB-97	01-JAN-97

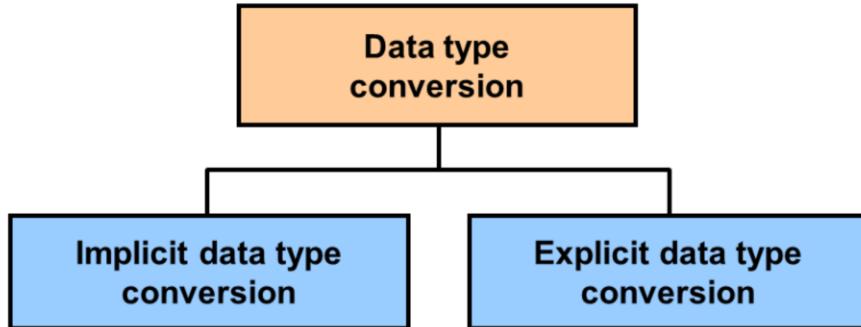
# **Using Conversion Functions and Conditional Expressions**



**ORACLE®**

Copyright © 2009, Oracle. All rights reserved.

# Conversion Functions



ORACLE

4 - 2

Copyright © 2009, Oracle. All rights reserved.

## Conversion Functions

In addition to Oracle data types, columns of tables in an Oracle Database can be defined by using the American National Standards Institute (ANSI), DB2, and SQL/DS data types. However, the Oracle server internally converts such data types to Oracle data types.

In some cases, the Oracle server receives data of one data type where it expects data of a different data type. When this happens, the Oracle server can automatically convert the data to the expected data type. This data type conversion can be done *implicitly* by the Oracle server or *explicitly* by the user.

Implicit data type conversions work according to the rules explained in the following slides.

Explicit data type conversions are performed by using the conversion functions. Conversion functions convert a value from one data type to another. Generally, the form of the function names follows the convention *data type TO data type*. The first data type is the input data type and the second data type is the output.

**Note:** Although implicit data type conversion is available, it is recommended that you do the explicit data type conversion to ensure the reliability of your SQL statements.

## Implicit Data Type Conversion

In expressions, the Oracle server can automatically convert the following:

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

ORACLE®

4 - 3

Copyright © 2009, Oracle. All rights reserved.

### Implicit Data Type Conversion

Oracle server can automatically perform data type conversion in an expression. For example, the expression `hire_date > '01-JAN-90'` results in the implicit conversion from the string '`01-JAN-90`' to a date. Therefore, a VARCHAR2 or CHAR value can be implicitly converted to a number or date data type in an expression.

## Implicit Data Type Conversion

For expression evaluation, the Oracle server can automatically convert the following:

From	To
NUMBER	VARCHAR2 or CHAR
DATE	VARCHAR2 or CHAR

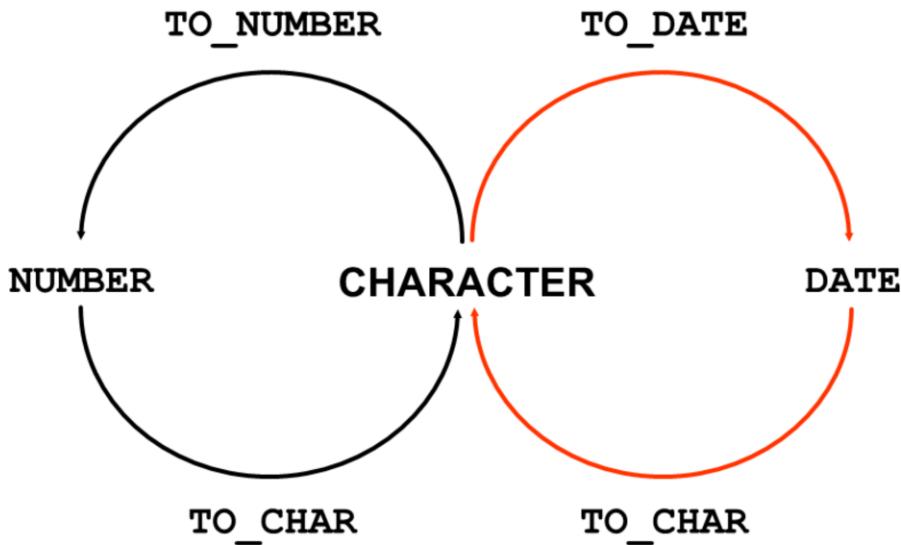
ORACLE®

### Implicit Data Type Conversion (continued)

In general, the Oracle server uses the rule for expressions when a data type conversion is needed. For example, the expression `grade = 2` results in the implicit conversion of the number 2 to the string “2” because grade is a CHAR (2) column.

**Note:** CHAR to NUMBER conversions succeed only if the character string represents a valid number.

# Explicit Data Type Conversion



ORACLE

4 - 5

Copyright © 2009, Oracle. All rights reserved.

## Explicit Data Type Conversion

SQL provides three functions to convert a value from one data type to another:

Function	Purpose
<code>TO_CHAR(number date, [ fmt ], [nlsparams])</code>	<p>Converts a number or date value to a VARCHAR2 character string with the format model <i>fmt</i></p> <p><b>Number conversion:</b> The <i>nlsparams</i> parameter specifies the following characters, which are returned by number format elements:</p> <ul style="list-style-type: none"><li>• Decimal character</li><li>• Group separator</li><li>• Local currency symbol</li><li>• International currency symbol</li></ul> <p>If <i>nlsparams</i> or any other parameter is omitted, this function uses the default parameter values for the session.</p>

## Using the TO\_CHAR Function with Dates

```
TO_CHAR(date, 'format_model')
```

The format model:

- Must be enclosed with single quotation marks
- Is case-sensitive
- Can include any valid date format element
- Has an `fm` element to remove padded blanks or suppress leading zeros
- Is separated from the date value by a comma

ORACLE

4 - 6

Copyright © 2009, Oracle. All rights reserved.

## Using the TO\_CHAR Function with Dates

`TO_CHAR` converts a datetime data type to a value of `VARCHAR2` data type in the format specified by the *format\_model*. A format model is a character literal that describes the format of datetime stored in a character string. For example, the datetime format model for the string '11-Nov-1999' is 'DD-Mon-YYYY'. You can use the `TO_CHAR` function to convert a date from its default format to the one that you specify.

### Guidelines

- The format model must be enclosed with single quotation marks and is case-sensitive.
- The format model can include any valid date format element. But be sure to separate the date value from the format model with a comma.
- The names of days and months in the output are automatically padded with blanks.
- To remove padded blanks or to suppress leading zeros, use the fill mode `fm` element.

```
SELECT employee_id, TO_CHAR(hire_date, 'MM/YY') Month_Hired  
FROM   employees  
WHERE  last_name = 'Higgins';
```

	EMPLOYEE_ID	MONTH_HIRED
1		205 06/94

## Elements of the Date Format Model

Element	Result
YYYY	Full year in numbers
YEAR	Year spelled out (in English)
MM	Two-digit value for the month
MONTH	Full name of the month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day of the week
DD	Numeric day of the month

ORACLE®

## Elements of the Date Format Model

- Time elements format the time portion of the date:

HH24:MI:SS AM	15:45:32 PM
---------------	-------------

- Add character strings by enclosing them with double quotation marks:

DD "of" MONTH	12 of OCTOBER
---------------	---------------

- Number suffixes spell out numbers:

ddspth	fourteenth
--------	------------

ORACLE®

## Elements of the Date Format Model

Use the formats that are listed in the following tables to display time information and literals, and to change numerals to spelled numbers.

Element	Description
AM or PM	Meridian indicator
A.M. or P.M.	Meridian indicator with periods
HH or HH12 or HH24	Hour of day, or hour (1–12), or hour (0–23)
MI	Minute (0–59)
SS	Second (0–59)
SSSS	Seconds past midnight (0–86399)

## Using the TO\_CHAR Function with Dates

```
SELECT last_name,  
       TO_CHAR(hire_date, 'fmDD Month YYYY')  
          AS HIREDATE  
FROM   employees;
```

	LAST_NAME	HIREDATE
1	Whalen	17 September 1987
2	Hartstein	17 February 1996
3	Fay	17 August 1997
4	Higgins	7 June 1994
5	Gietz	7 June 1994
6	King	17 June 1987
7	Kochhar	21 September 1989
8	De Haan	13 January 1993
9	Hunold	3 January 1990
10	Ernst	21 May 1991
...		

ORACLE

4 - 9

Copyright © 2009, Oracle. All rights reserved.

## Using the TO\_CHAR Function with Dates

The SQL statement in the slide displays the last names and hire dates for all the employees. The hire date appears as 17 June 1987.

### Example:

Modify the example in the slide to display the dates in a format that appears as “Seventeenth of June 1987 12:00:00 AM.”

```
SELECT last_name,  
       TO_CHAR(hire_date,  
              'fmDdspth "of" Month YYYY fmHH:MI:SS AM')  
          AS HIREDATE  
FROM   employees;
```

	LAST_NAME	HIREDATE
1	Whalen	Seventeenth of September 1987 12:00:00 AM
2	Hartstein	Seventeenth of February 1996 12:00:00 AM
...		

Notice that the month follows the format model specified; in other words, the first letter is capitalized and the rest are in lowercase.

## Using the TO\_CHAR Function with Numbers

```
TO_CHAR(number, 'format_model')
```

These are some of the format elements that you can use with the TO\_CHAR function to display a number value as a character:

Element	Result
9	Represents a number
0	Forces a zero to be displayed
\$	Places a floating dollar sign
L	Uses the floating local currency symbol
.	Prints a decimal point
,	Prints a comma as a thousands indicator

ORACLE®

## Using the TO\_CHAR Function with Numbers

When working with number values, such as character strings, you should convert those numbers to the character data type using the TO\_CHAR function, which translates a value of NUMBER data type to VARCHAR2 data type. This technique is especially useful with concatenation.

## Using the TO\_CHAR Function with Numbers

```
SELECT TO_CHAR(salary, '$99,999.00') SALARY  
FROM   employees  
WHERE  last_name = 'Ernst';
```

	SALARY
1	\$6,000.00

ORACLE®

4 - 11

Copyright © 2009, Oracle. All rights reserved.

## Using the TO\_CHAR Function with Numbers (continued)

- The Oracle server displays a string of number signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model.
- The Oracle server rounds the stored decimal value to the number of decimal places provided in the format model.

## Using the TO\_NUMBER and TO\_DATE Functions

- Convert a character string to a number format using the TO\_NUMBER function:

```
TO_NUMBER(char[, 'format_model'])
```

- Convert a character string to a date format using the TO\_DATE function:

```
TO_DATE(char[, 'format_model'])
```

- These functions have an *fx* modifier. This modifier specifies the exact match for the character argument and date format model of a TO\_DATE function.

ORACLE

4 - 12

Copyright © 2009, Oracle. All rights reserved.

## Using the TO\_NUMBER and TO\_DATE Functions

You may want to convert a character string to either a number or a date. To accomplish this task, use the TO\_NUMBER or TO\_DATE functions. The format model that you select is based on the previously demonstrated format elements.

The *fx* modifier specifies the exact match for the character argument and date format model of a TO\_DATE function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without *fx*, the Oracle server ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without *fx*, the numbers in the character argument can omit leading zeros.

## Using the TO\_CHAR and TO\_DATE Function with the RR Date Format

To find employees hired before 1990, use the RR date format, which produces the same results whether the command is run in 1999 or now:

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM   employees
WHERE  hire_date < TO_DATE('01-Jan-90', 'DD-Mon-RR');
```

	LAST_NAME	TO_CHAR(HIRE_DATE,'DD-MON-YYYY')
1	Whalen	17-Sep-1987
2	King	17-Jun-1987
3	Kochhar	21-Sep-1989

ORACLE

4 - 13

Copyright © 2009, Oracle. All rights reserved.

## Using the TO\_CHAR and TO\_DATE Function with the RR Date Format

To find employees who were hired before 1990, the RR format can be used. Because the current year is greater than 1999, the RR format interprets the year portion of the date from 1950 to 1999.

Alternatively, the following command, results in no rows being selected because the YY format interprets the year portion of the date in the current century (2090).

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-yyyy')
FROM   employees
WHERE  TO_DATE(hire_date, 'DD-Mon-yy') < '01-Jan-1990';
```

0 rows selected

## Nesting Functions

- Single-row functions can be nested to any level.
- Nested functions are evaluated from the deepest level to the least deep level.

**F3 (F2 (F1 (col,arg1) ,arg2) ,arg3)**

**Step 1 = Result 1**

**Step 2 = Result 2**

**Step 3 = Result 3**

**ORACLE**

## Nesting Functions

Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level. Some examples follow to show you the flexibility of these functions.

## Nesting Functions: Example

```
SELECT last_name,  
       UPPER(CONCAT(SUBSTR (LAST_NAME, 1, 8), '_US'))  
  FROM employees  
 WHERE department_id = 60;
```

	LAST_NAME	UPPER(CONCAT(SUBSTR(LAST_NAME,1,8),'_US'))
1	Hunold	HUNOLD_US
2	Ernst	ERNST_US
3	Lorentz	LORENTZ_US

ORACLE

4 - 15

Copyright © 2009, Oracle. All rights reserved.

## Nesting Functions (continued)

The example in the slide displays the last names of employees in department 60. The evaluation of the SQL statement involves three steps:

1. The inner function retrieves the first eight characters of the last name.

Result1 = SUBSTR (LAST\_NAME, 1, 8)

2. The outer function concatenates the result with \_US.

Result2 = CONCAT(Result1, '\_US')

3. The outermost function converts the results to uppercase.

The entire expression becomes the column heading because no column alias was given.

### Example:

Display the date of the next Friday that is six months from the hire date. The resulting date should appear as Friday, August 13th, 1999. Order the results by hire date.

```
SELECT      TO_CHAR(NEXT_DAY(ADD_MONTHS  
                           (hire_date, 6), 'FRIDAY'),  
                           'fmDay, Month ddth, YYYY')  
                         "Next 6 Month Review"  
  FROM        employees  
 ORDER BY    hire_date;
```

## General Functions

The following functions work with any data type and pertain to using nulls:

- NVL (expr1, expr2)
- NVL2 (expr1, expr2, expr3)
- NULLIF (expr1, expr2)
- COALESCE (expr1, expr2, ..., exprn)

ORACLE

4 - 16

Copyright © 2009, Oracle. All rights reserved.

## General Functions

These functions work with any data type and pertain to the use of null values in the expression list.

Function	Description
NVL	Converts a null value to an actual value
NVL2	If expr1 is not null, NVL2 returns expr2. If expr1 is null, NVL2 returns expr3. The argument expr1 can have any data type.
NULLIF	Compares two expressions and returns null if they are equal; returns the first expression if they are not equal
COALESCE	Returns the first non-null expression in the expression list

**Note:** For more information about the hundreds of functions available, see the section on “Functions” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

## NVL Function

Converts a null value to an actual value:

- Data types that can be used are date, character, and number.
- Data types must match:
  - NVL(*commission\_pct*, 0)
  - NVL(*hire\_date*, '01-JAN-97')
  - NVL(*job\_id*, 'No Job Yet')

ORACLE®

4 - 17

Copyright © 2009, Oracle. All rights reserved.

### NVL Function

To convert a null value to an actual value, use the NVL function.

#### Syntax

NVL (*expr1*, *expr2*)

In the syntax:

- *expr1* is the source value or expression that may contain a null
- *expr2* is the target value for converting the null

You can use the NVL function to convert any data type, but the return value is always the same as the data type of *expr1*.

#### NVL Conversions for Various Data Types

Data Type	Conversion Example
NUMBER	NVL( <i>number_column</i> , 9)
DATE	NVL( <i>date_column</i> , '01-JAN-95')
CHAR or VARCHAR2	NVL( <i>character_column</i> , 'Unavailable')

## Using the NVL Function

```
SELECT last_name, salary, NVL(commission_pct, 0),
       (salary*12) + (salary*12*NVL(commission_pct, 0)) AN_SAL
  FROM employees;
```

	LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
1	Whalen	4400	0	52800
2	Hartstein	13000	0	156000
3	Fay	6000	0	72000
4	Higgins	12000	0	144000
5	Gietz	8300	0	99600
6	King	24000	0	288000
7	Kochhar	17000	0	204000
8	De Haan	17000	0	204000
9	Hunold	9000	0	108000
10	Ernst	6000	0	72000

...

1

2

ORACLE

4 - 18

Copyright © 2009, Oracle. All rights reserved.

## Using the NVL Function

To calculate the annual compensation of all employees, you need to multiply the monthly salary by 12 and then add the commission percentage to the result:

```
SELECT last_name, salary, commission_pct,
       (salary*12) + (salary*12*commission_pct) AN_SAL
  FROM employees;
```

	LAST_NAME	SALARY	COMMISSION_PCT	AN_SAL
1	Whalen	4400	(null)	(null)
...				
16	Vargas	2500	(null)	(null)
17	Zlotkey	10500	0.2	151200
18	Abel	11000	0.3	171600
19	Taylor	8600	0.2	123840
20	Grant	7000	0.15	96600

Notice that the annual compensation is calculated for only those employees who earn a commission. If any column value in an expression is null, the result is null. To calculate values for all employees, you must convert the null value to a number before applying the arithmetic operator. In the example in the slide, the NVL function is used to convert null values to zero.

## Using the NVL2 Function

```
SELECT last_name, salary, commission_pct  
      NVL2(commission_pct,  
            'SAL+COMM', 'SAL') income  
FROM   employees WHERE department_id IN (50, 80);
```

LAST_NAME	SALARY	COMMISSION_PCT	INCOME
Mourgos	5800	(null)	SAL
Rajs	3500	(null)	SAL
Davies	3100	(null)	SAL
Matos	2600	(null)	SAL
Vargas	2500	(null)	SAL
Zlotkey	10500	0.2	SAL+COMM
Abel	11000	0.3	SAL+COMM
Taylor	8600	0.2	SAL+COMM



ORACLE

## Using the NVL2 Function

The NVL2 function examines the first expression. If the first expression is not null, the NVL2 function returns the second expression. If the first expression is null, the third expression is returned.

### Syntax

```
NVL2(expr1, expr2, expr3)
```

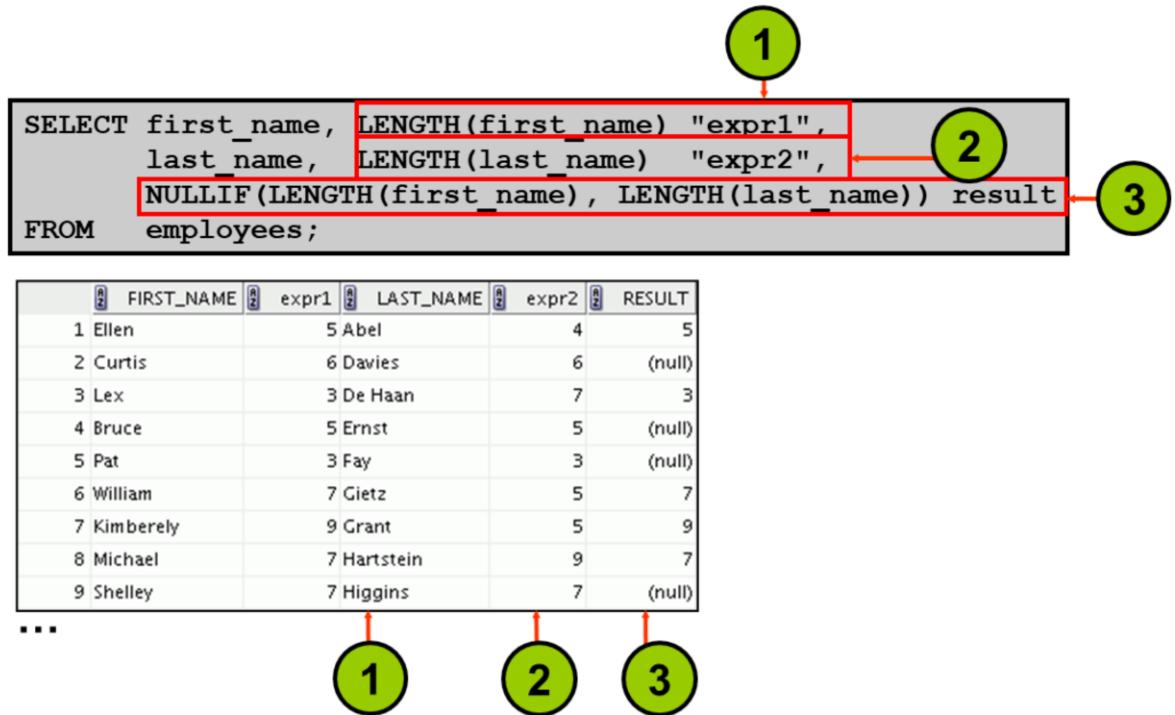
In the syntax:

- *expr1* is the source value or expression that may contain a null
- *expr2* is the value that is returned if *expr1* is not null
- *expr3* is the value that is returned if *expr1* is null

In the example shown in the slide, the COMMISSION\_PCT column is examined. If a value is detected, the text literal value of SAL+COMM is returned. If the COMMISSION\_PCT column contains a null value, the text literal value of SAL is returned.

**Note:** The argument *expr1* can have any data type. The arguments *expr2* and *expr3* can have any data types except LONG.

## Using the NULLIF Function



ORACLE

## Using the NULLIF Function

The `NULLIF` function compares two expressions.

### Syntax

```
NULLIF (expr1, expr2)
```

In the syntax:

- `NULLIF` compares `expr1` and `expr2`. If they are equal, the function returns null. If they are not, the function returns `expr1`. However, you cannot specify the literal `NULL` for `expr1`.

In the example shown in the slide, the length of the first name in the `EMPLOYEES` table is compared to the length of the last name in the `EMPLOYEES` table. When the lengths of the names are equal, a null value is displayed. When the lengths of the names are not equal, the length of the first name is displayed.

**Note:** The `NULLIF` function is logically equivalent to the following `CASE` expression. The `CASE` expression is discussed on a subsequent page:

```
CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END
```

## Using the COALESCE Function

- The advantage of the COALESCE function over the NVL function is that the COALESCE function can take multiple alternate values.
- If the first expression is not null, the COALESCE function returns that expression; otherwise, it does a COALESCE of the remaining expressions.

ORACLE®

4 - 21

Copyright © 2009, Oracle. All rights reserved.

## Using the COALESCE Function

The COALESCE function returns the first non-null expression in the list.

### Syntax

COALESCE (*expr1, expr2, ... exprn*)

In the syntax:

- *expr1* returns this expression if it is not null
- *expr2* returns this expression if the first expression is null and this expression is not null
- *exprn* returns this expression if the preceding expressions are null

Note that all expressions must be of the same data type.

## Using the COALESCE Function

```
SELECT last_name, employee_id,  
COALESCE(TO_CHAR(commission_pct),TO_CHAR(manager_id),  
'No commission and no manager')  
FROM employees;
```

LAST_NAME	EMPLOYEE_ID	COALESCE(TO_CHAR(COMMISI...)
Whalen	200 101	
Hartstein	201 100	
Fay	202 201	
Higgins	205 101	
Gietz	206 205	
King	100	No commission and no manager
***		
17 Zlotkey	149 .2	
18 Abel	174 .3	
19 Taylor	176 .2	
20 Grant	178 .15	

ORACLE

### Using the COALESCE Function (continued)

In the example shown in the slide, if the manager\_id value is not null, it is displayed. If the manager\_id value is null, the commission\_pct is displayed. If the manager\_id and commission\_pct values are null, “No commission and no manager” is displayed. Note that TO\_CHAR function is applied so that all expressions are of the same data type.

## Conditional Expressions

- Provide the use of the IF-THEN-ELSE logic within a SQL statement.
- Use two methods:
  - CASE expression
  - DECODE function

ORACLE®

4 - 23

Copyright © 2009, Oracle. All rights reserved.

### Conditional Expressions

The two methods that are used to implement conditional processing (IF-THEN-ELSE logic) in a SQL statement are the CASE expression and the DECODE function.

**Note:** The CASE expression complies with the ANSI SQL. The DECODE function is specific to Oracle syntax.

## CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
CASE expr WHEN comparison_expr1 THEN return_expr1
            [WHEN comparison_expr2 THEN return_expr2
            WHEN comparison_exprn THEN return_exprn
            ELSE else_expr]
END
```

ORACLE®

4 - 24

Copyright © 2009, Oracle. All rights reserved.

### CASE Expression

CASE expressions allow you to use the IF-THEN-ELSE logic in SQL statements without having to invoke procedures.

In a simple CASE expression, the Oracle server searches for the first WHEN . . . THEN pair for which expr is equal to comparison\_expr and returns return\_expr. If none of the WHEN . . . THEN pairs meet this condition, and if an ELSE clause exists, the Oracle server returns else\_expr. Otherwise, the Oracle server returns a null. You cannot specify the literal NULL for all the return\_exprs and the else\_expr.

The expressions expr and comparison\_expr must be of the same data type, which can be CHAR, VARCHAR2, NCHAR, or NVARCHAR2. All of the return values (return\_expr) must be of the same data type.

## Using the CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
SELECT last_name, job_id, salary,  
       CASE job_id WHEN 'IT_PROG' THEN 1.10*salary  
                     WHEN 'ST_CLERK' THEN 1.15*salary  
                     WHEN 'SA REP' THEN 1.20*salary  
                     ELSE salary END      "REVISED_SALARY"  
FROM employees;
```

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
1	Whalen	AD_ASST	4400	4400
...				
9	Hunold	IT_PROG	9000	9900
10	Ernst	IT_PROG	6000	6600
11	Lorentz	IT_PROG	4200	4620
12	Mourgos	ST_MAN	5800	5800
13	Rajs	ST_CLERK	3500	4025
14	Davies	ST_CLERK	3100	3565
...				
19	Taylor	SA REP	8600	10320
20	Grant	SA REP	7000	8400

ORACLE

## Using the CASE Expression

In the SQL statement in the slide, the value of JOB\_ID is decoded. If JOB\_ID is IT\_PROG, the salary increase is 10%; if JOB\_ID is ST\_CLERK, the salary increase is 15%; if JOB\_ID is SA REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be written with the DECODE function.

The following code is an example of the searched CASE expression. In a searched CASE expression, the search occurs from left to right until an occurrence of the listed condition is found, and then it returns the return expression. If no condition is found to be true, and if an ELSE clause exists, the return expression in the ELSE clause is returned; otherwise, a NULL is returned.

```
SELECT last_name, salary,  
       (CASE WHEN salary<5000 THEN 'Low'  
             WHEN salary<10000 THEN 'Medium'  
             WHEN salary<20000 THEN 'Good'  
             ELSE 'Excellent'  
        END) qualified_salary  
FROM employees;
```

## DECODE Function

Facilitates conditional inquiries by doing the work of a CASE expression or an IF-THEN-ELSE statement:

```
DECODE(col|expression, search1, result1
       [, search2, result2, ...]
       [, default])
```

ORACLE®

4 - 26

Copyright © 2009, Oracle. All rights reserved.

### DECODE Function

The DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic that is used in various languages. The DECODE function decodes *expression* after comparing it to each *search* value. If the expression is the same as *search*, *result* is returned.

If the default value is omitted, a null value is returned where a search value does not match any of the result values.

## Using the DECODE Function

```
SELECT last_name, job_id, salary,  
       DECODE(job_id, 'IT_PROG', 1.10*salary,  
              'ST_CLERK', 1.15*salary,  
              'SA REP',    1.20*salary,  
              salary)  
          REVISED_SALARY  
FROM   employees;
```

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...				
10	Ernst	IT_PROG	6000	6600
11	Lorentz	IT_PROG	4200	4620
12	Mourgos	ST_MAN	5800	5800
13	Rajs	ST_CLERK	3500	4025
...				
19	Taylor	SA REP	8600	10320
20	Grant	SA REP	7000	8400

ORACLE

4 - 27

Copyright © 2009, Oracle. All rights reserved.

## Using the DECODE Function

In the SQL statement in the slide, the value of JOB\_ID is tested. If JOB\_ID is IT\_PROG, the salary increase is 10%; if JOB\_ID is ST\_CLERK, the salary increase is 15%; if JOB\_ID is SA REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be expressed in pseudocode as an IF-THEN-ELSE statement:

```
IF job_id = 'IT_PROG'      THEN  salary = salary*1.10  
IF job_id = 'ST_CLERK'     THEN  salary = salary*1.15  
IF job_id = 'SA REP'       THEN  salary = salary*1.20  
ELSE salary = salary
```

## Using the DECODE Function

Display the applicable tax rate for each employee in department 80:

```
SELECT last_name, salary,  
       DECODE (TRUNC(salary/2000, 0),  
                0, 0.00,  
                1, 0.09,  
                2, 0.20,  
                3, 0.30,  
                4, 0.40,  
                5, 0.42,  
                6, 0.44,  
                0.45) TAX_RATE  
  FROM employees  
 WHERE department_id = 80;
```

ORACLE

4 - 28

Copyright © 2009, Oracle. All rights reserved.

### Using the DECODE Function (continued)

This slide shows another example using the DECODE function. In this example, you determine the tax rate for each employee in department 80 based on the monthly salary. The tax rates are as follows:

Monthly Salary Range	Tax Rate
\$0.00–1,999.99	00%
\$2,000.00–3,999.99	09%
\$4,000.00–5,999.99	20%
\$6,000.00–7,999.99	30%
\$8,000.00–9,999.99	40%
\$10,000.00–11,999.99	42%
\$12,200.00–13,999.99	44%
\$14,000.00 or greater	45%

LAST_NAME	SALARY	TAX_RATE
Zlotkey	10500	0.42
Abel	11000	0.42
Taylor	8600	0.4

# Reporting Aggregated Data Using the Group Functions

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

# What Are Group Functions?

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

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	110	12000
5	110	8300
6	90	24000
7	90	17000
8	90	17000
9	60	9000
10	60	6000
...		
18	80	11000
19	80	8600
20	(null)	7000

Maximum salary in  
EMPLOYEES table

MAX(SALARY)
24000

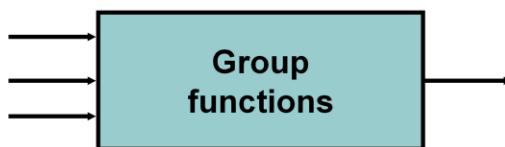
ORACLE

## What Are Group Functions?

Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may comprise the entire table or the table split into groups.

# Types of Group Functions

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- SUM
- VARIANCE



ORACLE

5 - 3

Copyright © 2009, Oracle. All rights reserved.

## Types of Group Functions

Each of the functions accepts an argument. The following table identifies the options that you can use in the syntax:

Function	Description
AVG ( [DISTINCT   <u>ALL</u> ] <i>n</i> )	Average value of <i>n</i> , ignoring null values
COUNT ( { *   [DISTINCT   <u>ALL</u> ] <i>expr</i> } )	Number of rows, where <i>expr</i> evaluates to something other than null (count all selected rows using *, including duplicates and rows with nulls)
MAX ( [DISTINCT   <u>ALL</u> ] <i>expr</i> )	Maximum value of <i>expr</i> , ignoring null values
MIN ( [DISTINCT   <u>ALL</u> ] <i>expr</i> )	Minimum value of <i>expr</i> , ignoring null values
STDDEV ( [DISTINCT   <u>ALL</u> ] <i>n</i> )	Standard deviation of <i>n</i> , ignoring null values
SUM ( [DISTINCT   <u>ALL</u> ] <i>n</i> )	Sum values of <i>n</i> , ignoring null values
VARIANCE ( [DISTINCT   <u>ALL</u> ] <i>n</i> )	Variance of <i>n</i> , ignoring null values

# Group Functions: Syntax

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

ORACLE

5 - 4

Copyright © 2009, Oracle. All rights reserved.

## Group Functions: Syntax

The group function is placed after the SELECT keyword. You may have multiple group functions separated by commas.

Guidelines for using the group functions:

- DISTINCT makes the function consider only nonduplicate values; ALL makes it consider every value, including duplicates. The default is ALL and, therefore, does not need to be specified.
- The data types for the functions with an expr argument may be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions ignore null values. To substitute a value for null values, use the NVL, NVL2, COALESCE, CASE, or DECODE functions.

# Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SELECT AVG(salary), MAX(salary),
       MIN(salary), SUM(salary)
  FROM employees
 WHERE job_id LIKE '%REP%';
```

	AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
1	8150	11000	6000	32600

ORACLE

## Using the AVG and SUM Functions

You can use the AVG, SUM, MIN, and MAX functions against the columns that can store numeric data. The example in the slide displays the average, highest, lowest, and sum of monthly salaries for all sales representatives.

## Using the MIN and MAX Functions

You can use MIN and MAX for numeric, character, and date data types.

```
SELECT MIN(hire_date), MAX(hire_date)  
FROM employees;
```

	MIN(HIRE_DATE)	MAX(HIRE_DATE)
1	17-JUN-87	29-JAN-00

ORACLE

5 - 6

Copyright © 2009, Oracle. All rights reserved.

## Using the MIN and MAX Functions

You can use the MAX and MIN functions for numeric, character, and date data types. The example in the slide displays the most junior and most senior employees.

The following example displays the employee last name that is first and the employee last name that is last in an alphabetic list of all employees:

```
SELECT MIN(last_name), MAX(last_name)  
FROM employees;
```

	MIN(LAST_NAME)	MAX(LAST_NAME)
1	Abel	Zlotkey

**Note:** The AVG, SUM, VARIANCE, and STDDEV functions can be used only with numeric data types. MAX and MIN cannot be used with LOB or LONG data types.

# Using the COUNT Function

COUNT (\*) returns the number of rows in a table:

1

```
SELECT COUNT(*)
FROM employees
WHERE department_id = 50;
```

AZ	COUNT(*)
1	5

COUNT (expr) returns the number of rows with non-null values for expr:

2

```
SELECT COUNT(commission_pct)
FROM employees
WHERE department_id = 80;
```

AZ	COUNT(COMMISSION_PCT)
1	3

ORACLE

## Using the COUNT Function

The COUNT function has three formats:

- COUNT (\*)
- COUNT (expr)
- COUNT (DISTINCT expr)

COUNT (\*) returns the number of rows in a table that satisfy the criteria of the SELECT statement, including duplicate rows and rows containing null values in any of the columns. If a WHERE clause is included in the SELECT statement, COUNT (\*) returns the number of rows that satisfy the condition in the WHERE clause.

In contrast, COUNT (expr) returns the number of non-null values that are in the column identified by expr.

COUNT (DISTINCT expr) returns the number of unique, non-null values that are in the column identified by expr.

### Examples:

1. The example in the slide displays the number of employees in department 50.
2. The example in the slide displays the number of employees in department 80 who can earn a commission.

## Using the DISTINCT Keyword

- COUNT(DISTINCT expr) returns the number of distinct non-null values of *expr*.
- To display the number of distinct department values in the EMPLOYEES table:

```
SELECT COUNT(DISTINCT department_id)
FROM employees;
```

1	COUNT(DISTINCTDEPARTMENT_ID)
1	7

ORACLE

## Using the DISTINCT Keyword

Use the DISTINCT keyword to suppress the counting of any duplicate values in a column.

The example in the slide displays the number of distinct department values that are in the EMPLOYEES table.

# Group Functions and Null Values

Group functions ignore null values in the column:

1

```
SELECT AVG(commission_pct)  
FROM employees;
```

	Avg(COMMISSION_PCT)
1	0.2125

The NVL function forces group functions to include null values:

2

```
SELECT AVG(NVL(commission_pct, 0))  
FROM employees;
```

	Avg(NVL(COMMISSION_PCT,0))
1	0.0425

ORACLE

5 - 9

Copyright © 2009, Oracle. All rights reserved.

## Group Functions and Null Values

All group functions ignore null values in the column.

However, the NVL function forces group functions to include null values.

### Examples:

1. The average is calculated based on *only* those rows in the table in which a valid value is stored in the COMMISSION\_PCT column. The average is calculated as the total commission that is paid to all employees divided by the number of employees receiving a commission (four).
2. The average is calculated based on *all* rows in the table, regardless of whether null values are stored in the COMMISSION\_PCT column. The average is calculated as the total commission that is paid to all employees divided by the total number of employees in the company (20).

# Creating Groups of Data

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	2500
5	50	2600
6	50	3100
7	50	3500
8	50	5800
9	60	9000
10	60	6000
11	60	4200
12	80	11000
13	80	8600
...		
18	110	8300
19	110	12000
20	(null)	7000

4400  
9500  
3500  
6400  
10033

Average salary in the EMPLOYEES table for each department

	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	20	9500
3	90	19333.333333333333...
4	110	10150
5	50	3500
6	80	10033.333333333333...
7	10	4400
8	60	6400

ORACLE

## Creating Groups of Data

Until this point in the discussion, all group functions have treated the table as one large group of information. At times, however, you need to divide the table of information into smaller groups. This can be done by using the GROUP BY clause.

## Creating Groups of Data: GROUP BY Clause Syntax

You can divide rows in a table into smaller groups by using the GROUP BY clause.

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

ORACLE

5 - 11

Copyright © 2009, Oracle. All rights reserved.

## Creating Groups of Data: GROUP BY Clause Syntax

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use the group functions to return summary information for each group.

In the syntax:

*group\_by\_expression*      Specifies the columns whose values determine the basis for grouping rows

### Guidelines

- If you include a group function in a SELECT clause, you cannot select individual results as well, *unless* the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups.
- You must include the *columns* in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.

## Using the GROUP BY Clause

All the columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

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

	DEPARTMENT_ID	Avg(SALARY)
1	(null)	7000
2	20	9500
3	90	19333.33333333333...
4	110	10150
5	50	3500
6	80	10033.33333333333...
7	10	4400
8	60	6400

ORACLE

## Using the GROUP BY Clause

When using the GROUP BY clause, make sure that all columns in the SELECT list that are not group functions are included in the GROUP BY clause. The example in the slide displays the department number and the average salary for each department. Here is how this SELECT statement, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the columns to be retrieved, as follows:
  - Department number column in the EMPLOYEES table
  - The average of all salaries in the group that you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The WHERE clause specifies the rows to be retrieved. Because there is no WHERE clause, all rows are retrieved by default.
- The GROUP BY clause specifies how the rows should be grouped. The rows are grouped by department number, so the AVG function that is applied to the salary column calculates the average salary for each department.

**Note:** To order the query results in ascending or descending order, include the ORDER BY clause in the query.

# Using the GROUP BY Clause

The GROUP BY column does not have to be in the SELECT list.

```
SELECT      AVG(salary)
FROM        employees
GROUP BY    department id ;
```

	Avg(Salary)
1	7000
2	9500
3	19333.3333333333333333333333...
4	10150
5	3500
6	10033.3333333333333333333333...
7	4400
8	6400

ORACLE®

5 - 13

Copyright © 2009, Oracle. All rights reserved.

## Using the GROUP BY Clause (continued)

The GROUP BY column does not have to be in the SELECT clause. For example, the SELECT statement in the slide displays the average salaries for each department without displaying the respective department numbers. Without the department numbers, however, the results do not look meaningful.

You can also use the group function in the ORDER BY clause:

```
SELECT      department_id, AVG(salary)
FROM        employees
GROUP BY    department_id
ORDER BY    AVG(salary);
```

DEPARTMENT_ID	Avg(Salary)
1	50
2	10
3	60

1

# Grouping by More Than One Column

EMPLOYEES

	DEPARTMENT_ID	JOB_ID	SALARY
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_CLERK	2500
5	50	ST_CLERK	2600
6	50	ST_CLERK	3100
7	50	ST_CLERK	3500
8	50	ST_MAN	5800
9	60	IT_PROG	9000
10	60	IT_PROG	6000
11	60	IT_PROG	4200
12	80	SA_REP	11000
13	80	SA_REP	8600
14	80	SA_MAN	10500
...			
19	110	AC_MGR	12000
20	(null)	SA_REP	7000

Add the salaries in the EMPLOYEES table for each job, grouped by department.

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	110	AC_ACCOUNT	8300
2	110	AC_MGR	12000
3	10	AD_ASST	4400
4	90	AD_PRES	24000
5	90	AD_VP	34000
6	60	IT_PROG	19200
7	20	MK_MAN	13000
8	20	MK_REP	6000
9	80	SA_MAN	10500
10	80	SA_REP	19600
11	(null)	SA_REP	7000
12	50	ST_CLERK	11700
13	50	ST_MAN	5800

ORACLE

## Grouping by More Than One Column

Sometimes, you need to see results for groups within groups. The slide shows a report that displays the total salary that is paid to each job title in each department.

The EMPLOYEES table is grouped first by the department number, and then by the job title within that grouping. For example, the four stock clerks in department 50 are grouped together, and a single result (total salary) is produced for all stock clerks in the group.

The following SELECT statement returns the result shown in the slide:

```
SELECT    department_id, job_id, sum(salary)
  FROM    employees
 GROUP BY department_id, job_id
 ORDER BY job_id;
```

# Using the GROUP BY Clause on Multiple Columns

```
SELECT      department_id, job_id, SUM(salary)
FROM        employees
WHERE       department_id > 40
GROUP BY    department_id, job_id
ORDER BY    department_id;
```

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	50	ST_CLERK	11700
2	50	ST_MAN	5800
3	60	IT_PROG	19200
4	80	SA_MAN	10500
5	80	SA_REP	19600
6	90	AD_PRES	24000
7	90	AD_VP	34000
8	110	AC_ACCOUNT	8300
9	110	AC_MGR	12000

ORACLE

5 - 15

Copyright © 2009, Oracle. All rights reserved.

## Using the Group By Clause on Multiple Columns

You can return summary results for groups and subgroups by listing multiple GROUP BY columns. The GROUP BY clause groups rows but does not guarantee the order of the result set. To order the groupings, use the ORDER BY clause.

In the example in the slide, the SELECT statement that contains a GROUP BY clause is evaluated as follows:

- The SELECT clause specifies the column to be retrieved:
  - Department ID in the EMPLOYEES table
  - Job ID in the EMPLOYEES table
  - The sum of all salaries in the group that you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The WHERE clause reduces the result set to those rows where department ID is greater than 40.
- The GROUP BY clause specifies how you must group the resulting rows:
  - First, the rows are grouped by the department ID.
  - Second, the rows are grouped by job ID in the department ID groups.
- The ORDER BY clause sorts the results by department ID.

**Note:** The SUM function is applied to the salary column for all job IDs in the result set in each department ID group. Also, note that the SA\_REP row is not returned. The department ID for this row is NULL and, therefore, does not meet the WHERE condition.

## Illegal Queries Using Group Functions

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause:

```
SELECT department_id, COUNT(last_name)  
FROM employees;
```

ORA-00937: not a single-group group function  
00937. 00000 - "not a single-group group function"

A GROUP BY clause must be added to  
count the last names for each  
department\_id.

```
SELECT department_id, job_id, COUNT(last_name)  
FROM employees  
GROUP BY department_id;
```

ORA-00979: not a GROUP BY expression  
00979. 00000 - "not a GROUP BY expression"

Either add job\_id in the GROUP BY or  
remove the job\_id column from the  
SELECT list.

ORACLE

## Illegal Queries Using Group Functions

Whenever you use a mixture of individual items (DEPARTMENT\_ID) and group functions (COUNT) in the same SELECT statement, you must include a GROUP BY clause that specifies the individual items (in this case, DEPARTMENT\_ID). If the GROUP BY clause is missing, the error message “not a single-group group function” appears and an asterisk (\*) points to the offending column. You can correct the error in the first example in the slide by adding the GROUP BY clause:

```
SELECT department_id, count(last_name)  
FROM employees  
GROUP BY department_id;
```

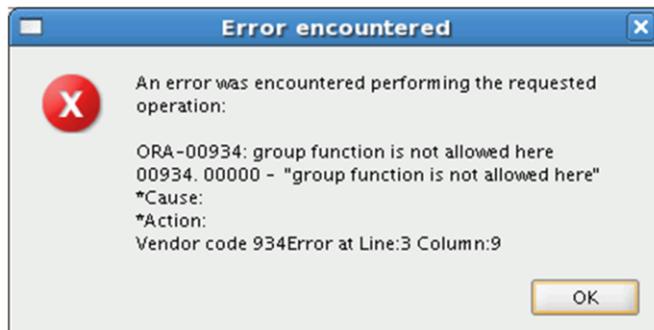
Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause. In the second example in the slide, job\_id is neither in the GROUP BY clause nor is it being used by a group function, so there is a “not a GROUP BY expression” error. You can correct the error in the second slide example by adding job\_id in the GROUP BY clause.

```
SELECT department_id, job_id, COUNT(last_name)  
FROM employees  
GROUP BY department_id, job_id;
```

## Illegal Queries Using Group Functions

- You cannot use the WHERE clause to restrict groups.
- You use the HAVING clause to restrict groups.
- You cannot use group functions in the WHERE clause.

```
SELECT      department_id, AVG(salary)
FROM        employees
WHERE       AVG(salary) > 8000
GROUP BY    department_id;
```



Cannot use the  
WHERE clause to  
restrict groups

ORACLE

5 - 17

Copyright © 2009, Oracle. All rights reserved.

## Illegal Queries Using Group Functions (continued)

The WHERE clause cannot be used to restrict groups. The SELECT statement in the example in the slide results in an error because it uses the WHERE clause to restrict the display of the average salaries of those departments that have an average salary greater than \$8,000.

However, you can correct the error in the example by using the HAVING clause to restrict groups:

```
SELECT      department_id, AVG(salary)
FROM        employees
GROUP BY    department_id
HAVING     AVG(salary) > 8000;
```

	DEPARTMENT_ID	AVG(SALARY)
1	20	9500
2	90	19333.3333333333...
3	110	10150
4	80	10033.3333333333...

# Restricting Group Results

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	2500
5	50	2600
6	50	3100
7	50	3500
8	50	5800
9	60	9000
10	60	6000
11	60	4200
12	80	11000
13	80	8600
...		
18	110	8300
19	110	12000
20	(null)	7000

The maximum salary per department when it is greater than \$10,000

	DEPARTMENT_ID	MAX(SALARY)
1	20	13000
2	90	24000
3	110	12000
4	80	11000

## Restricting Group Results

You use the HAVING clause to restrict groups in the same way that you use the WHERE clause to restrict the rows that you select. To find the maximum salary in each of the departments that have a maximum salary greater than \$10,000, you need to do the following:

1. Find the average salary for each department by grouping by department number.
2. Restrict the groups to those departments with a maximum salary greater than \$10,000.

## Restricting Group Results with the HAVING Clause

When you use the HAVING clause, the Oracle server restricts groups as follows:

1. Rows are grouped.
2. The group function is applied.
3. Groups matching the HAVING clause are displayed.

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY  group_by_expression]
[HAVING    group_condition]
[ORDER BY  column];
```

ORACLE

5 - 19

Copyright © 2009, Oracle. All rights reserved.

## Restricting Group Results with the HAVING Clause

You use the HAVING clause to specify the groups that are to be displayed, thus further restricting the groups on the basis of aggregate information.

In the syntax, *group\_condition* restricts the groups of rows returned to those groups for which the specified condition is true.

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

1. Rows are grouped.
2. The group function is applied to the group.
3. The groups that match the criteria in the HAVING clause are displayed.

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. Groups are formed and group functions are calculated before the HAVING clause is applied to the groups in the SELECT list.

**Note:** The WHERE clause restricts rows, whereas the HAVING clause restricts groups.

## Using the HAVING Clause

```
SELECT      department_id, MAX(salary)
FROM        employees
GROUP BY    department_id
HAVING      MAX(salary)>10000 ;
```

	DEPARTMENT_ID	MAX(SALARY)
1	20	13000
2	90	24000
3	110	12000
4	80	11000

ORACLE

5 - 20

Copyright © 2009, Oracle. All rights reserved.

## Using the HAVING Clause

The example in the slide displays the department numbers and maximum salaries for those departments with a maximum salary greater than \$10,000.

You can use the GROUP BY clause without using a group function in the SELECT list. If you restrict rows based on the result of a group function, you must have a GROUP BY clause as well as the HAVING clause.

The following example displays the department numbers and average salaries for those departments with a maximum salary greater than \$10,000:

```
SELECT      department_id, AVG(salary)
FROM        employees
GROUP BY    department_id
HAVING      max(salary)>10000;
```

	DEPARTMENT_ID	AVG(SALARY)
1	20	9500
2	90	19333.3333333333...
3	110	10150
4	80	10033.3333333333...

## Using the HAVING Clause

```
SELECT      job_id, SUM(salary) PAYROLL
FROM        employees
WHERE       job_id NOT LIKE '%REP%'
GROUP BY    job_id
HAVING      SUM(salary) > 13000
ORDER BY    SUM(salary);
```

JOB_ID	PAYROLL
1 IT_PROG	19200
2 AD_PRES	24000
3 AD_VP	34000

ORACLE

### Using the HAVING Clause (continued)

The example in the slide displays the job ID and total monthly salary for each job that has a total payroll exceeding \$13,000. The example excludes sales representatives and sorts the list by the total monthly salary.

## Nesting Group Functions

Display the maximum average salary:

```
SELECT MAX(AVG(salary))  
FROM employees  
GROUP BY department_id;
```

AZ MAX(AVG(SALARY))

ORACLE®

5 - 22

Copyright © 2009, Oracle. All rights reserved.

# Nesting Group Functions

Group functions can be nested to a depth of two functions. The example in the slide calculates the average salary for each department\_id and then displays the maximum average salary.

Note that GROUP BY clause is mandatory when nesting group functions.

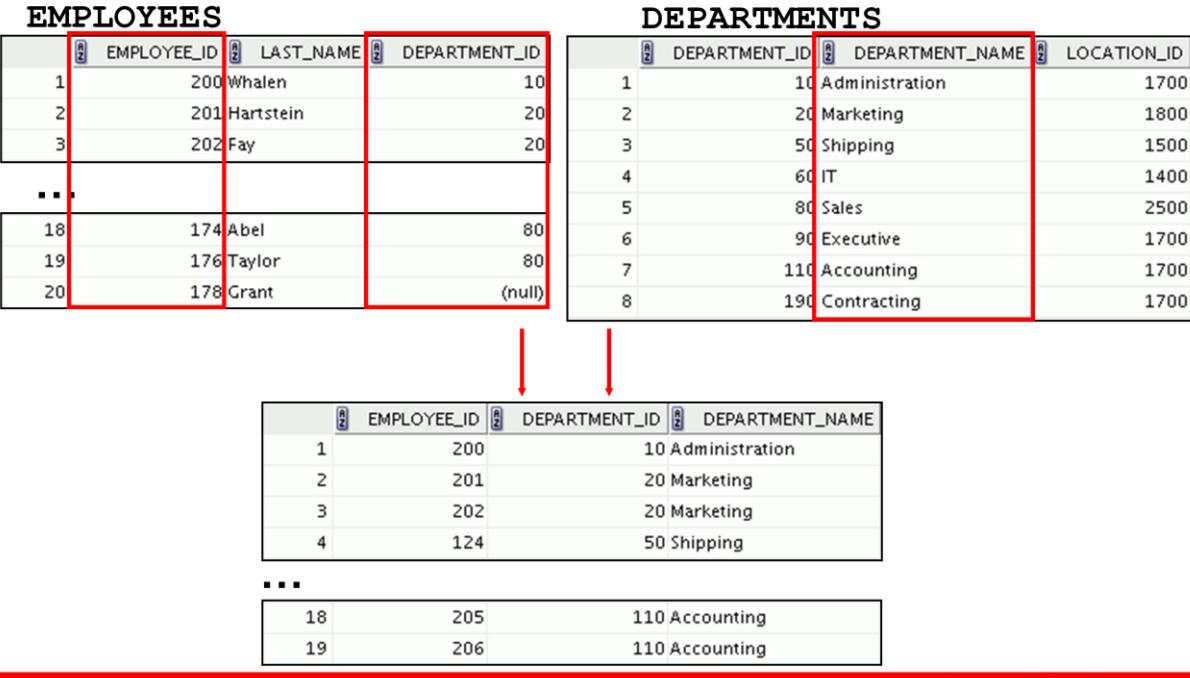


# **Displaying Data from Multiple Tables Using Joins**

**ORACLE®**

Copyright © 2009, Oracle. All rights reserved.

# Obtaining Data from Multiple Tables



ORACLE

## Obtaining Data from Multiple Tables

Sometimes you need to use data from more than one table. In the example in the slide, the report displays data from two separate tables:

- Employee IDs exist in the EMPLOYEES table.
- Department IDs exist in both the EMPLOYEES and DEPARTMENTS tables.
- Department names exist in the DEPARTMENTS table.

To produce the report, you need to link the EMPLOYEES and DEPARTMENTS tables, and access data from both of them.

# Types of Joins

Joins that are compliant with the SQL:1999 standard include the following:

- Natural joins:
  - NATURAL JOIN clause
  - USING clause
  - ON clause
- OUTER joins:
  - LEFT OUTER JOIN
  - RIGHT OUTER JOIN
  - FULL OUTER JOIN
- Cross joins

ORACLE

6 - 3

Copyright © 2009, Oracle. All rights reserved.

## Types of Joins

To join tables, you can use a join syntax that is compliant with the SQL:1999 standard.

### Note

- Before the Oracle9*i* release, the join syntax was different from the American National Standards Institute (ANSI) standards. The SQL:1999-compliant join syntax does not offer any performance benefits over the Oracle-proprietary join syntax that existed in the prior releases. For detailed information about the proprietary join syntax, see Appendix F: Oracle Join Syntax.
- The following slide discusses the SQL:1999 join syntax.

# Joining Tables Using SQL:1999 Syntax

Use a join to query data from more than one table:

```
SELECT    table1.column, table2.column
FROM      table1
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
  ON (table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.column_name = table2.column_name)] |
[CROSS JOIN table2];
```

ORACLE

6 - 4

Copyright © 2009, Oracle. All rights reserved.

## Joining Tables Using SQL:1999 Syntax

In the syntax:

- table1.column denotes the table and the column from which data is retrieved
- NATURAL JOIN joins two tables based on the same column name
- JOIN table2 USING column\_name performs an equijoin based on the column name
- JOIN table2 ON table1.column\_name = table2.column\_name performs an equijoin based on the condition in the ON clause
- LEFT/RIGHT/FULL OUTER is used to perform OUTER joins
- CROSS JOIN returns a Cartesian product from the two tables

For more information, see the section titled “SELECT” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

# Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Instead of full table name prefixes, use table aliases.
- Table alias gives a table a shorter name:
  - Keeps SQL code smaller, uses less memory
- Use column aliases to distinguish columns that have identical names, but reside in different tables.

ORACLE

6 - 5

Copyright © 2009, Oracle. All rights reserved.

## Qualifying Ambiguous Column Names

When joining two or more tables, you need to qualify the names of the columns with the table name to avoid ambiguity. Without the table prefixes, the DEPARTMENT\_ID column in the SELECT list could be from either the DEPARTMENTS table or the EMPLOYEES table. It is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, using the table prefix improves performance, because you tell the Oracle server exactly where to find the columns.

However, qualifying column names with table names can be time consuming, particularly if the table names are lengthy. Instead, you can use *table aliases*. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore, using less memory.

The table name is specified in full, followed by a space, and then the table alias. For example, the EMPLOYEES table can be given an alias of e, and the DEPARTMENTS table an alias of d.

### Guidelines

- Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
- If a table alias is used for a particular table name in the FROM clause, that table alias must be substituted for the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- The table alias is valid for only the current SELECT statement.

## Creating Natural Joins

- The NATURAL JOIN clause is based on all the columns in the two tables that have the same name.
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types, an error is returned.

ORACLE

6 - 6

Copyright © 2009, Oracle. All rights reserved.

### Creating Natural Joins

You can join tables automatically based on the columns in the two tables that have matching data types and names. You do this by using the NATURAL JOIN keywords.

**Note:** The join can happen on only those columns that have the same names and data types in both tables. If the columns have the same name but different data types, the NATURAL JOIN syntax causes an error.

# Retrieving Records with Natural Joins

```
SELECT department_id, department_name,
       location_id, city
  FROM departments
NATURAL JOIN locations ;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
1	60	IT	1400	Southlake
2	50	Shipping	1500	South San Francisco
3	10	Administration	1700	Seattle
4	90	Executive	1700	Seattle
5	110	Accounting	1700	Seattle
6	190	Contracting	1700	Seattle
7	20	Marketing	1800	Toronto
8	80	Sales	2500	Oxford

ORACLE

## Retrieving Records with Natural Joins

In the example in the slide, the LOCATIONS table is joined to the DEPARTMENT table by the LOCATION\_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

### Natural Joins with a WHERE Clause

Additional restrictions on a natural join are implemented by using a WHERE clause. The following example limits the rows of output to those with a department ID equal to 20 or 50:

```
SELECT department_id, department_name,
       location_id, city
  FROM departments
NATURAL JOIN locations
 WHERE department_id IN (20, 50);
```

## Creating Joins with the USING Clause

- If several columns have the same names but the data types do not match, use the USING clause to specify the columns for the equijoin.
- Use the USING clause to match only one column when more than one column matches.
- The NATURAL JOIN and USING clauses are mutually exclusive.

ORACLE

6 - 8

Copyright © 2009, Oracle. All rights reserved.

### Creating Joins with the USING Clause

Natural joins use all columns with matching names and data types to join the tables. The USING clause can be used to specify only those columns that should be used for an equijoin.

# Joining Column Names

EMPLOYEES

	EMPLOYEE_ID	DEPARTMENT_ID
1	200	10
2	201	20
3	202	20
4	205	110
5	206	110
6	100	90
7	101	90
8	102	90
9	103	60
10	104	60

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME
1	10	Administration
2	20	Marketing
3	50	Shipping
4	60	IT
5	80	Sales
6	90	Executive
7	110	Accounting
8	190	Contracting

Primary key

Foreign key

## Joining Column Names

To determine an employee's department name, you compare the value in the DEPARTMENT\_ID column in the EMPLOYEES table with the DEPARTMENT\_ID values in the DEPARTMENTS table. The relationship between the EMPLOYEES and DEPARTMENTS tables is an *equijoin*; that is, values in the DEPARTMENT\_ID column in both the tables must be equal. Frequently, this type of join involves primary and foreign key complements.

**Note:** Equijoins are also called *simple joins* or *inner joins*.

## Retrieving Records with the USING Clause

```
SELECT employee_id, last_name,  
       location_id, department_id  
FROM   employees JOIN departments  
USING (department_id) ;
```

	EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
1	200	Whalen	1700	10
2	201	Hartstein	1800	20
3	202	Fay	1800	20
4	144	Vargas	1500	50
5	143	Matos	1500	50
6	142	Davies	1500	50
7	141	Rajs	1500	50
8	124	Mourgos	1500	50
...				
18	206	Gietz	1700	110
19	205	Higgins	1700	110

ORACLE

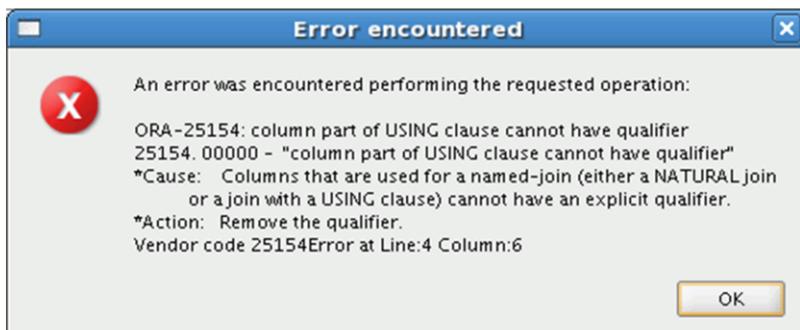
## Retrieving Records with the USING Clause

In the example in the slide, the DEPARTMENT\_ID columns in the EMPLOYEES and DEPARTMENTS tables are joined and thus the LOCATION\_ID of the department where an employee works is shown.

## Using Table Aliases with the USING Clause

- Do not qualify a column that is used in the USING clause.
- If the same column is used elsewhere in the SQL statement, do not alias it.

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



ORACLE

6 - 11

Copyright © 2009, Oracle. All rights reserved.

### Using Table Aliases with the USING clause

When joining with the USING clause, you cannot qualify a column that is used in the USING clause itself. Furthermore, if that column is used anywhere in the SQL statement, you cannot alias it. For example, in the query mentioned in the slide, you should not alias the location\_id column in the WHERE clause because the column is used in the USING clause.

The columns that are referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement. For example, the following statement is valid:

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

The columns that are common in both the tables, but not used in the USING clause, must be prefixed with a table alias; otherwise, you get the “column ambiguously defined” error.

In the following statement, manager\_id is present in both the employees and departments table; if manager\_id is not prefixed with a table alias, it gives a “column ambiguously defined” error.

The following statement is valid:

```
SELECT first_name, d.department_name, d.manager_id
FROM   employees e JOIN departments d USING (department_id)
WHERE  department_id = 50;
```

## Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- Use the `ON` clause to specify arbitrary conditions or specify columns to join.
- The join condition is separated from other search conditions.
- The `ON` clause makes code easy to understand.

ORACLE

6 - 12

Copyright © 2009, Oracle. All rights reserved.

### Creating Joins with the ON Clause

Use the `ON` clause to specify a join condition. With this, you can specify join conditions separate from any search or filter conditions in the `WHERE` clause.

## Retrieving Records with the ON Clause

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

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	200 Whalen	10	10	1700
2	201 Hartstein	20	20	1800
3	202 Fay	20	20	1800
4	144 Vargas	50	50	1500
5	143 Matos	50	50	1500
6	142 Davies	50	50	1500
7	141 Rajs	50	50	1500
8	124 Mourgos	50	50	1500
9	103 Hunold	60	60	1400
10	104 Ernst	60	60	1400
11	107 Lorentz	60	60	1400
...				

ORACLE

## Retrieving Records with the ON Clause

In this example, the DEPARTMENT\_ID columns in the EMPLOYEES and DEPARTMENTS table are joined using the ON clause. Wherever a department ID in the EMPLOYEES table equals a department ID in the DEPARTMENTS table, the row is returned. The table alias is necessary to qualify the matching column names.

You can also use the ON clause to join columns that have different names. The parenthesis around the joined columns, as in the example in the slide, (e.department\_id = d.department\_id) is optional. So, even ON e.department\_id = d.department\_id will work.

**Note:** When you use the Execute Statement icon to run the query, SQL Developer suffixes a '\_1' to differentiate between the two department\_ids.

## Creating Three-Way Joins with the ON Clause

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

	EMPLOYEE_ID	CITY	DEPARTMENT_NAME
1	100	Seattle	Executive
2	101	Seattle	Executive
3	102	Seattle	Executive
4	103	Southlake	IT
5	104	Southlake	IT
6	107	Southlake	IT
7	124	South San Francisco	Shipping
8	141	South San Francisco	Shipping
9	142	South San Francisco	Shipping
...			

ORACLE

## Creating Three-Way Joins with the ON Clause

A three-way join is a join of three tables. In SQL:1999-compliant syntax, joins are performed from left to right. So, the first join to be performed is EMPLOYEES JOIN DEPARTMENTS. The first join condition can reference columns in EMPLOYEES and DEPARTMENTS but cannot reference columns in LOCATIONS. The second join condition can reference columns from all three tables.

**Note:** The code example in the slide can also be accomplished with the USING clause:

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

## Applying Additional Conditions to a Join

Use the AND clause or the WHERE clause to apply additional conditions:

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
  FROM employees e JOIN departments d
 WHERE (e.department_id = d.department_id)
   AND e.manager_id = 149;
```

Or

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
  FROM employees e JOIN departments d
 WHERE (e.department_id = d.department_id)
   WHERE e.manager_id = 149;
```

ORACLE

6 - 15

Copyright © 2009, Oracle. All rights reserved.

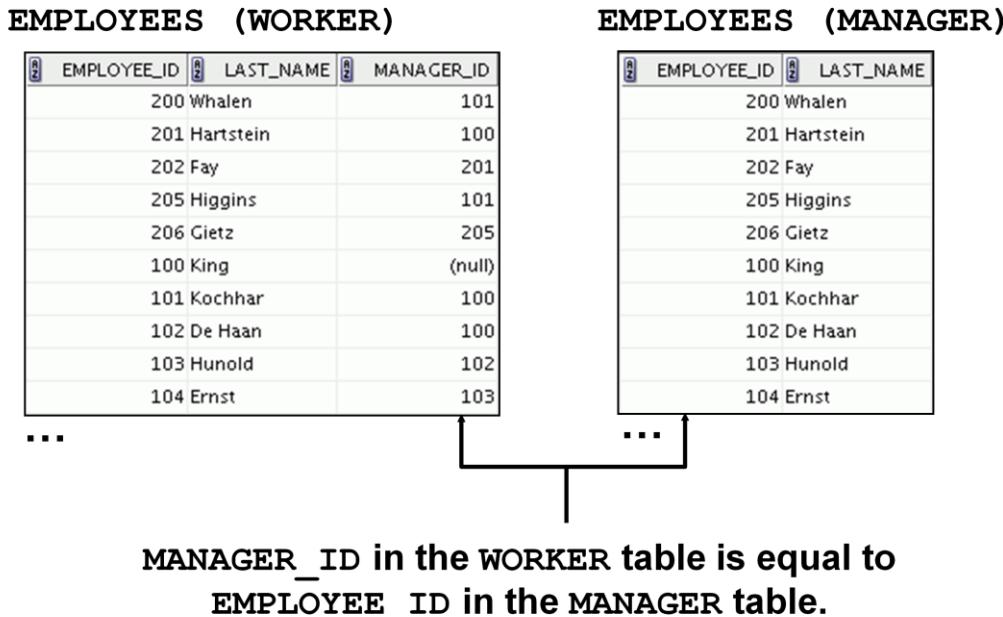
## Applying Additional Conditions to a Join

You can apply additional conditions to the join.

The example shown performs a join on the EMPLOYEES and DEPARTMENTS tables and, in addition, displays only employees who have a manager ID of 149. To add additional conditions to the ON clause, you can add AND clauses. Alternatively, you can use a WHERE clause to apply additional conditions.

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	174 Abel	80	80	2500
2	176 Taylor	80	80	2500

## Joining a Table to Itself



ORACLE

### Joining a Table to Itself

Sometimes you need to join a table to itself. To find the name of each employee's manager, you need to join the EMPLOYEES table to itself, or perform a self-join. For example, to find the name of Lorentz's manager, you need to:

- Find Lorentz in the EMPLOYEES table by looking at the LAST\_NAME column
- Find the manager number for Lorentz by looking at the MANAGER\_ID column. Lorentz's manager number is 103.
- Find the name of the manager with EMPLOYEE\_ID 103 by looking at the LAST\_NAME column. Hunold's employee number is 103, so Hunold is Lorentz's manager.

In this process, you look in the table twice. The first time you look in the table to find Lorentz in the LAST\_NAME column and the MANAGER\_ID value of 103. The second time you look in the EMPLOYEE\_ID column to find 103 and the LAST\_NAME column to find Hunold.

## Self-Joins Using the ON Clause

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

	EMP	MGR
1	Hunold	De Haan
2	Fay	Hartstein
3	Gietz	Higgins
4	Lorentz	Hunold
5	Ernst	Hunold
6	Zlotkey	King
7	Mourgos	King
...		

ORACLE

## Self-Joins Using the ON Clause

The ON clause can also be used to join columns that have different names, within the same table or in a different table.

The example shown is a self-join of the EMPLOYEES table, based on the EMPLOYEE\_ID and MANAGER\_ID columns.

**Note:** The parenthesis around the joined columns as in the example in the slide, (e.manager\_id = m.employee\_id) is **optional**. So, even ON e.manager\_id = m.employee\_id will work.

# Nonequijoins

EMPLOYEES

	LAST_NAME	SALARY
1	Whalen	4400
2	Hartstein	13000
3	Fay	6000
4	Higgins	12000
5	Gietz	8300
6	King	24000
7	Kochhar	17000
8	De Haan	17000
9	Hunold	9000
10	Ernst	6000
...		
19	Taylor	8600
20	Grant	7000

JOB\_GRADES

	GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

The JOB\_GRADES table defines the LOWEST\_SAL and HIGHEST\_SAL range of values for each GRADE\_LEVEL. Therefore, the GRADE\_LEVEL column can be used to assign grades to each employee.

ORACLE

## Nonequijoins

A nonequijoin is a join condition containing something other than an equality operator.

The relationship between the EMPLOYEES table and the JOB\_GRADES table is an example of a nonequijoin. The SALARY column in the EMPLOYEES table ranges between the values in the LOWEST\_SAL and HIGHEST\_SAL columns of the JOB\_GRADES table. Therefore, each employee can be graded based on their salary. The relationship is obtained using an operator other than the equality (=) operator.

## Retrieving Records with Nonequijoins

```
SELECT e.last_name, e.salary, j.grade_level
FROM   employees e JOIN job_grades j
ON     e.salary
      BETWEEN j.lowest_sal AND j.highest_sal;
```

	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Mourgos	5800	B
8	Ernst	6000	C
9	Fay	6000	C
10	Grant	7000	C
...			

ORACLE

## Retrieving Records with Nonequijoins

The example in the slide creates a nonequijoin to evaluate an employee's salary grade. The salary must be *between* any pair of the low and high salary ranges.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the JOB\_GRADES table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees' salaries lie within the limits provided by the job grade table. That is, no employee earns less than the lowest value contained in the LOWEST\_SAL column or more than the highest value contained in the HIGHEST\_SAL column.

**Note:** Other conditions (such as  $\leq$  and  $\geq$ ) can be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using the BETWEEN condition. The Oracle server translates the BETWEEN condition to a pair of AND conditions. Therefore, using BETWEEN has no performance benefits, but should be used only for logical simplicity.

Table aliases have been specified in the slide example for performance reasons, not because of possible ambiguity.

# Returning Records with No Direct Match Using OUTER Joins

DEPARTMENTS

	DEPARTMENT_NAME	DEPARTMENT_ID
1	Administration	10
2	Marketing	20
3	Shipping	50
4	IT	60
5	Sales	80
6	Executive	90
7	Accounting	110
8	Contracting	190

Equijoin with EMPLOYEES

	DEPARTMENT_ID	LAST_NAME
1	10	Whalen
2	20	Hartstein
3	20	Fay
4	110	Higgins
5	110	Gietz
6	90	King
7	90	Kochhar
8	90	De Haan
9	60	Hunold
10	60	Ernst
...		
18	80	Abel
19	80	Taylor

There are no employees in department 190.

Employee "Grant" has not been assigned a department ID.

ORACLE

## Returning Records with No Direct Match Using OUTER Joins

If a row does not satisfy a join condition, the row does not appear in the query result.

In the slide example, a simple equijoin condition is used on the EMPLOYEES and DEPARTMENTS tables to return the result on the right. The result set does not contain the following:

- Department ID 190, because there are no employees with that department ID recorded in the EMPLOYEES table
- The employee with the last name of Grant, because this employee has not been assigned a department ID

To return the department record that does not have any employees, or employees that do not have an assigned department, you can use an OUTER join.

## INNER Versus OUTER Joins

- In SQL:1999, the join of two tables returning only matched rows is called an INNER join.
- A join between two tables that returns the results of the INNER join as well as the unmatched rows from the left (or right) table is called a left (or right) OUTER join.
- A join between two tables that returns the results of an INNER join as well as the results of a left and right join is a full OUTER join.

ORACLE

6 - 21

Copyright © 2009, Oracle. All rights reserved.

### INNER Versus OUTER Joins

Joining tables with the NATURAL JOIN, USING, or ON clauses results in an INNER join. Any unmatched rows are not displayed in the output. To return the unmatched rows, you can use an OUTER join. An OUTER join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other table satisfy the join condition.

There are three types of OUTER joins:

- LEFT OUTER
- RIGHT OUTER
- FULL OUTER

## LEFT OUTER JOIN

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1 Whalen	10	Administration
2 Fay	20	Marketing
3 Hartstein	20	Marketing
4 Vargas	50	Shipping
5 Matos	50	Shipping
...		
16 Kochhar	90	Executive
17 King	90	Executive
18 Gietz	110	Accounting
19 Higgins	110	Accounting
20 Grant	(null)	(null)

ORACLE

## LEFT OUTER JOIN

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

## RIGHT OUTER JOIN

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1 Whalen	10	Administration
2 Hartstein	20	Marketing
3 Fay	20	Marketing
4 Davies	50	Shipping
5 Vargas	50	Shipping
6 Rajs	50	Shipping
7 Mourgos	50	Shipping
8 Matos	50	Shipping
...		
18 Higgins	110	Accounting
19 Gietz	110	Accounting
20 (null)	190	Contracting

ORACLE

### RIGHT OUTER JOIN

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

## FULL OUTER JOIN

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1 Whalen	10	Administration
2 Hartstein	20	Marketing
3 Fay	20	Marketing
4 Higgins	110	Accounting
...		
17 Zlotkey	80	Sales
18 Abel	80	Sales
19 Taylor	80	Sales
20 Grant	(null)	(null)
21 (null)	190	Contracting

ORACLE

### FULL OUTER JOIN

This query retrieves all rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

# Cartesian Products

- A Cartesian product is formed when:
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table
- Always include a valid join condition if you want to avoid a Cartesian product.

ORACLE

6 - 25

Copyright © 2009, Oracle. All rights reserved.

## Cartesian Products

When a join condition is invalid or omitted completely, the result is a *Cartesian product*, in which all combinations of rows are displayed. All rows in the first table are joined to all rows in the second table.

A Cartesian product tends to generate a large number of rows and the result is rarely useful. You should, therefore, always include a valid join condition unless you have a specific need to combine all rows from all tables.

Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.

# Generating a Cartesian Product

**EMPLOYEES (20 rows)**

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	200	Whalen	10
2	201	Hartstein	20
3	202	Fay	20
4	205	Higgins	110
...			
19	176	Taylor	80
20	178	Grant	(null)

**DEPARTMENTS (8 rows)**

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

**Cartesian product:**  
 **$20 \times 8 = 160$  rows**

	EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
1	200	10	1700
2	201	20	1700
...			
21	200	10	1800
22	201	20	1800
...			
159	176	80	1700
160	178	(null)	1700

**ORACLE**

## Generating a Cartesian Product

A Cartesian product is generated if a join condition is omitted. The example in the slide displays the employee last name and the department name from the EMPLOYEES and DEPARTMENTS tables. Because no join condition was specified, all rows (20 rows) from the EMPLOYEES table are joined with all rows (8 rows) in the DEPARTMENTS table, thereby generating 160 rows in the output.

## Creating Cross Joins

- The CROSS JOIN clause produces the cross-product of two tables.
- This is also called a Cartesian product between the two tables.

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

LAST_NAME	DEPARTMENT_NAME
Abel	Administration
Davies	Administration
De Haan	Administration
Ernst	Administration
Fay	Administration
...	
158 Vargas	Contracting
159 Whalen	Contracting
160 Zlotkey	Contracting

ORACLE

## Creating Cross Joins

The example in the slide produces a Cartesian product of the EMPLOYEES and DEPARTMENTS tables.

The CROSS JOIN technique can be applied to many situations usefully. For example, to return total labor cost by office by month, even if month X has no labor cost, you can do a cross join of Offices with a table of all Months.

It is a good practice to explicitly state CROSS JOIN in your SELECT when you intend to create a Cartesian product. Therefore, it is very clear that you intend for this to happen and it is not the result of missing joins.

# Using Subqueries to Solve Queries

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

# Using a Subquery to Solve a Problem

Who has a salary greater than Abel's?

Main query:



Which employees have salaries greater than Abel's salary?

Subquery:



What is Abel's salary?

ORACLE

7 - 2

Copyright © 2009, Oracle. All rights reserved.

## Using a Subquery to Solve a Problem

Suppose you want to write a query to find out who earns a salary greater than Abel's salary.

To solve this problem, you need *two* queries: one to find how much Abel earns, and a second query to find who earns more than that amount.

You can solve this problem by combining the two queries, placing one query *inside* the other query.

The inner query (or *subquery*) returns a value that is used by the outer query (or *main query*). Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.

# Subquery Syntax

```
SELECT    select_list
FROM      table
WHERE     expr operator
          (SELECT    select_list
           FROM      table);
```

- The subquery (inner query) executes *before* the main query (outer query).
- The result of the subquery is used by the main query.

ORACLE

7 - 3

Copyright © 2009, Oracle. All rights reserved.

## Subquery Syntax

A subquery is a SELECT statement that is embedded in the clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses, including the following:

- WHERE clause
- HAVING clause
- FROM clause

In the syntax:

*operator* includes a comparison condition such as *>*, *=*, or *IN*

**Note:** Comparison conditions fall into two classes: single-row operators (*>*, *=*, *>=*, *<*, *<>*, *<=*) and multiple-row operators (*IN*, *ANY*, *ALL*, *EXISTS*).

The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The subquery generally executes first, and its output is used to complete the query condition for the main (or outer) query.

# Using a Subquery

```
SELECT last_name, salary  
FROM   employees  
WHERE  salary > 11000  
       (SELECT salary  
        FROM   employees  
        WHERE  last_name = 'Abel');
```

	LAST_NAME	SALARY
1	Hartstein	13000
2	Higgins	12000
3	King	24000
4	Kochhar	17000
5	De Haan	17000

ORACLE

## Using a Subquery

In the slide, the inner query determines the salary of employee Abel. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than employee Abel.

## Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition for readability. (However, the subquery can appear on either side of the comparison operator.)
- Use single-row operators with single-row subqueries and multiple-row operators with multiple-row subqueries.

ORACLE

7 - 5

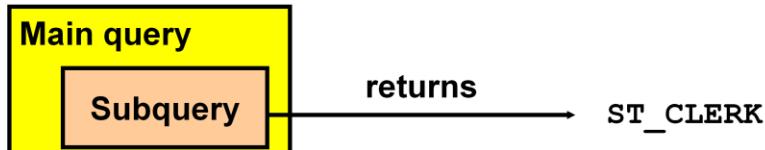
Copyright © 2009, Oracle. All rights reserved.

### Guidelines for Using Subqueries

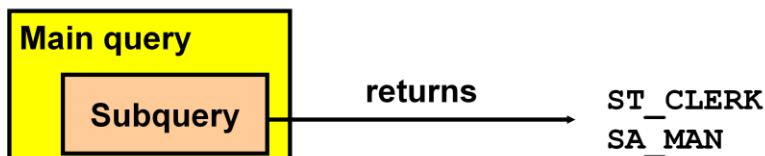
- A subquery must be enclosed in parentheses.
- Place the subquery on the right side of the comparison condition for readability. However, the subquery can appear on either side of the comparison operator.
- Two classes of comparison conditions are used in subqueries: single-row operators and multiple-row operators.

# Types of Subqueries

- Single-row subquery



- Multiple-row subquery



ORACLE

7 - 6

Copyright © 2009, Oracle. All rights reserved.

## Types of Subqueries

- **Single-row subqueries:** Queries that return only one row from the inner SELECT statement
- **Multiple-row subqueries:** Queries that return more than one row from the inner SELECT statement

**Note:** There are also multiple-column subqueries, which are queries that return more than one column from the inner SELECT statement. These are covered in the *Oracle Database 11g: SQL Fundamentals II* course.

# Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

ORACLE

7 - 7

Copyright © 2009, Oracle. All rights reserved.

## Single-Row Subqueries

A single-row subquery is one that returns one row from the inner SELECT statement. This type of subquery uses a single-row operator. The slide gives a list of single-row operators.

### Example:

Display the employees whose job ID is the same as that of employee 141:

```
SELECT last_name, job_id
  FROM employees
 WHERE job_id =
       (SELECT job_id
        FROM   employees
       WHERE  employee_id = 141);
```

	LAST_NAME	JOB_ID
1	Rajs	ST_CLERK
2	Davies	ST_CLERK
3	Matos	ST_CLERK
4	Vargas	ST_CLERK

# Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id = SA_REP
       (SELECT job_id
        FROM   employees
        WHERE  last_name = 'Taylor')
AND    salary > 8600
       (SELECT salary
        FROM   employees
        WHERE  last_name = 'Taylor');
```

#	LAST_NAME	JOB_ID	SALARY
1	Abel	SA_REP	11000

ORACLE

## Executing Single-Row Subqueries

A SELECT statement can be considered as a query block. The example in the slide displays employees who do the same job as “Taylor,” but earn more salary than him.

The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results SA\_REP and 8600, respectively. The outer query block is then processed and uses the values that were returned by the inner queries to complete its search conditions.

Both inner queries return single values (SA\_REP and 8600, respectively), so this SQL statement is called a single-row subquery.

**Note:** The outer and inner queries can get data from different tables.

## Using Group Functions in a Subquery

```
SELECT last_name, job_id, salary  
FROM   employees  
WHERE  salary = 2500  
       (SELECT MIN(salary)  
        FROM   employees);
```

	LAST_NAME	JOB_ID	SALARY
1	Vargas	ST_CLERK	2500

ORACLE

## Using Group Functions in a Subquery

You can display data from a main query by using a group function in a subquery to return a single row. The subquery is in parentheses and is placed after the comparison condition.

The example in the slide displays the employee last name, job ID, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (2500) to the outer query.

## HAVING Clause with Subqueries

- The Oracle server executes the subqueries first.
- The Oracle server returns results into the HAVING clause of the main query.

```
SELECT      department_id, MIN(salary)
FROM        employees
GROUP BY    department_id
HAVING      MIN(salary) > 2500
            (SELECT MIN(salary)
             FROM   employees
             WHERE  department_id = 50);
```

	DEPARTMENT_ID	MIN(SALARY)
1	(null)	7000
2	20	6000
3	90	17000
4	110	8300
5	80	8600
6	10	4400
7	60	4200

ORACLE

7 - 10

Copyright © 2009, Oracle. All rights reserved.

## HAVING Clause with Subqueries

You can use subqueries not only in the WHERE clause, but also in the HAVING clause. The Oracle server executes the subquery and the results are returned into the HAVING clause of the main query.

The SQL statement in the slide displays all the departments that have a minimum salary greater than that of department 50.

### Example:

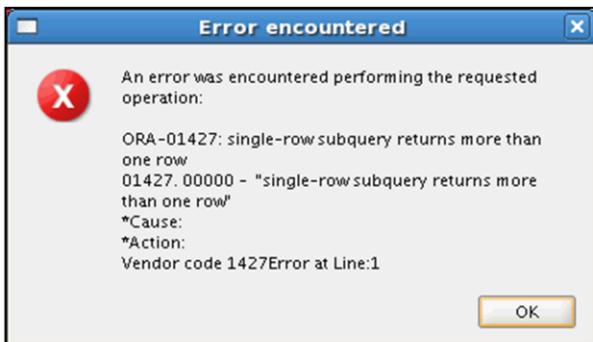
Find the job with the lowest average salary.

```
SELECT      job_id, AVG(salary)
FROM        employees
GROUP BY    job_id
HAVING      AVG(salary) = (SELECT      MIN(AVG(salary))
                           FROM        employees
                           GROUP BY    job_id);
```

	JOB_ID	AVG(SALARY)
1	ST_CLERK	2925

# What Is Wrong with This Statement?

```
SELECT employee_id, last_name  
FROM employees  
WHERE salary =  
      (SELECT MIN(salary)  
       FROM employees  
       GROUP BY department_id);
```



Single-row operator  
with multiple-row  
subquery

ORACLE

7 - 11

Copyright © 2009, Oracle. All rights reserved.

## What Is Wrong with This Statement?

A common error with subqueries occurs when more than one row is returned for a single-row subquery.

In the SQL statement in the slide, the subquery contains a GROUP BY clause, which implies that the subquery will return multiple rows, one for each group that it finds. In this case, the results of the subquery are 4400, 6000, 2500, 4200, 7000, 17000, and 8300.

The outer query takes those results and uses them in its WHERE clause. The WHERE clause contains an equal (=) operator, a single-row comparison operator that expects only one value. The = operator cannot accept more than one value from the subquery and, therefore, generates the error.

To correct this error, change the = operator to IN.

## No Rows Returned by the Inner Query

```
SELECT last_name, job_id  
FROM   employees  
WHERE  job_id =  
       (SELECT job_id  
        FROM   employees  
        WHERE  last_name = 'Haas');  
  
0 rows selected
```

Subquery returns no rows because there is no employee named “Haas.”

ORACLE

## No Rows Returned by the Inner Query

Another common problem with subqueries occurs when no rows are returned by the inner query. In the SQL statement in the slide, the subquery contains a WHERE clause. Presumably, the intention is to find the employee whose name is Haas. The statement is correct, but selects no rows when executed because there is no employee named Haas. Therefore, the subquery returns no rows. The outer query takes the results of the subquery (null) and uses these results in its WHERE clause. The outer query finds no employee with a job ID equal to null, and so returns no rows. If a job existed with a value of null, the row is not returned because comparison of two null values yields a null; therefore, the WHERE condition is not true.

# Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

Operator	Meaning
IN	Equal to any member in the list
ANY	Must be preceded by =, !=, >, <, <=, >=. Compares a value to each value in a list or returned by a query. Evaluates to FALSE if the query returns no rows.
ALL	Must be preceded by =, !=, >, <, <=, >=. Compares a value to every value in a list or returned by a query. Evaluates to TRUE if the query returns no rows.

ORACLE

7 - 13

Copyright © 2009, Oracle. All rights reserved.

## Multiple-Row Subqueries

Subqueries that return more than one row are called multiple-row subqueries. You use a multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values:

```
SELECT last_name, salary, department_id
  FROM employees
 WHERE salary IN (SELECT MIN(salary)
                   FROM employees
                   GROUP BY department_id);
```

### Example:

Find the employees who earn the same salary as the minimum salary for each department.

The inner query is executed first, producing a query result. The main query block is then processed and uses the values that were returned by the inner query to complete its search condition. In fact, the main query appears to the Oracle server as follows:

```
SELECT last_name, salary, department_id
  FROM employees
 WHERE salary IN (2500, 4200, 4400, 6000, 7000, 8300,
                  8600, 17000);
```

## Using the ANY Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
  FROM employees          9000, 6000, 4200
 WHERE salary < ANY
       (SELECT salary
        FROM   employees
        WHERE  job_id = 'IT_PROG')
 AND    job_id <> 'IT_PROG';
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	144	Vargas	ST_CLERK	2500
2	143	Matos	ST_CLERK	2600
3	142	Davies	ST_CLERK	3100
4	141	Rajs	ST_CLERK	3500
5	200	Whalen	AD_ASST	4400
...				
9	206	Gietz	AC_ACCOUNT	8300
10	176	Taylor	SA_REP	8600

ORACLE

### Using the ANY Operator in Multiple-Row Subqueries

The ANY operator (and its synonym, the SOME operator) compares a value to *each* value returned by a subquery. The slide example displays employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is \$9,000.

- <ANY means less than the maximum.
- >ANY means more than the minimum.
- =ANY is equivalent to IN.

## Using the ALL Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
  FROM employees      9000, 6000, 4200
 WHERE salary < ALL
          (SELECT salary
             FROM employees
            WHERE job_id = 'IT_PROG')
AND     job_id <> 'IT_PROG';
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	141	Rajs	ST_CLERK	3500
2	142	Davies	ST_CLERK	3100
3	143	Matos	ST_CLERK	2600
4	144	Vargas	ST_CLERK	2500

ORACLE

7 - 15

Copyright © 2009, Oracle. All rights reserved.

### Using the ALL Operator in Multiple-Row Subqueries

The ALL operator compares a value to *every* value returned by a subquery. The example in the slide displays employees whose salary is less than the salary of all employees with a job ID of IT\_PROG and whose job is not IT\_PROG.

>ALL means more than the maximum and <ALL means less than the minimum.

The NOT operator can be used with IN, ANY, and ALL operators.

## Null Values in a Subquery

```
SELECT emp.last_name
  FROM employees emp
 WHERE emp.employee_id NOT IN
       (SELECT mgr.manager_id
        FROM   employees mgr);
```

0 rows selected

ORACLE

7 - 16

Copyright © 2009, Oracle. All rights reserved.

### Null Values in a Subquery

The SQL statement in the slide attempts to display all the employees who do not have any subordinates. Logically, this SQL statement should have returned 12 rows. However, the SQL statement does not return any rows. One of the values returned by the inner query is a null value and, therefore, the entire query returns no rows.

The reason is that all conditions that compare a null value result in a null. So whenever null values are likely to be part of the results set of a subquery, do not use the NOT IN operator. The NOT IN operator is equivalent to <> ALL.

Notice that the null value as part of the results set of a subquery is not a problem if you use the IN operator. The IN operator is equivalent to =ANY. For example, to display the employees who have subordinates, use the following SQL statement:

```
SELECT emp.last_name
  FROM employees emp
 WHERE emp.employee_id IN
       (SELECT mgr.manager_id
        FROM   employees mgr);
```

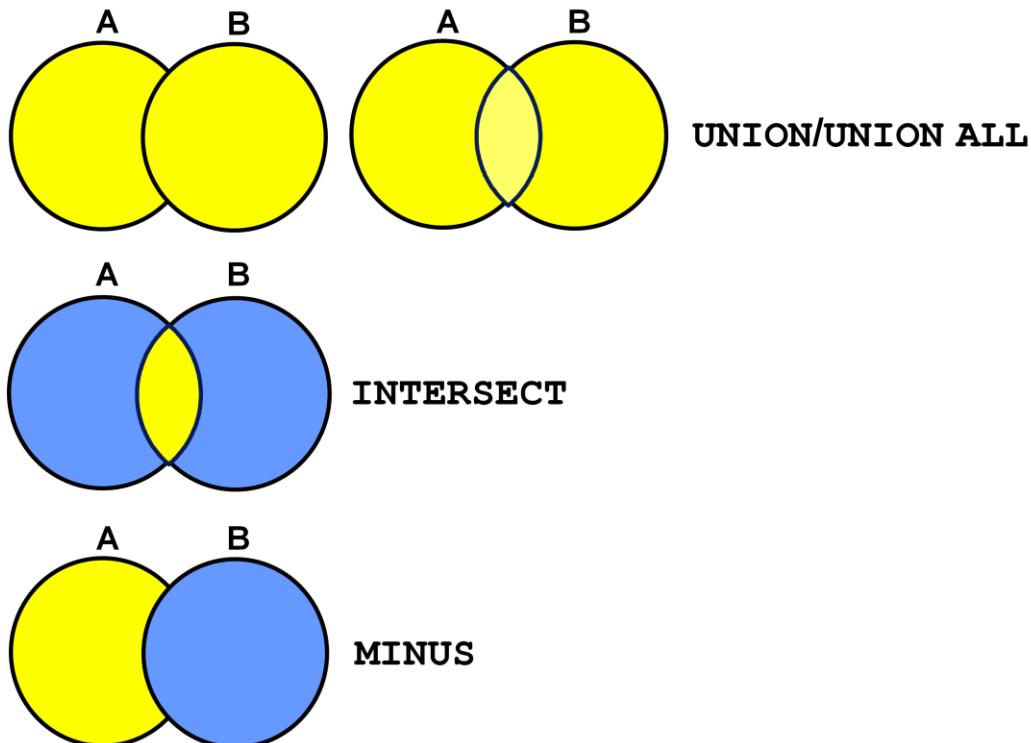
# 8

## Using the Set Operators

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

## Set Operators



ORACLE

8 - 2

Copyright © 2009, Oracle. All rights reserved.

## Set Operators

Set operators combine the results of two or more component queries into one result. Queries containing set operators are called *compound queries*.

Operator	Returns
UNION	Rows from both queries after eliminating duplications
UNION ALL	Rows from both queries, including all duplications
INTERSECT	Rows that are common to both queries
MINUS	Rows in the first query that are not present in the second query

All set operators have equal precedence. If a SQL statement contains multiple set operators, the Oracle server evaluates them from left (top) to right (bottom)—if no parentheses explicitly specify another order. You should use parentheses to specify the order of evaluation explicitly in queries that use the INTERSECT operator with other set operators.

## Set Operator Guidelines

- The expressions in the SELECT lists must match in number.
- The data type of each column in the second query must match the data type of its corresponding column in the first query.
- Parentheses can be used to alter the sequence of execution.
- ORDER BY clause can appear only at the very end of the statement.

ORACLE

8 - 3

Copyright © 2009, Oracle. All rights reserved.

### Set Operator Guidelines

- The expressions in the SELECT lists of the queries must match in number and data type. Queries that use UNION, UNION ALL, INTERSECT, and MINUS operators in their WHERE clause must have the same number and data type of columns in their SELECT list. The data type of the columns in the SELECT list of the queries in the compound query may not be exactly the same. The column in the second query must be in the same data type group (such as numeric or character) as the corresponding column in the first query.
- Set operators can be used in subqueries.
- You should use parentheses to specify the order of evaluation in queries that use the INTERSECT operator with other set operators. This ensures compliance with emerging SQL standards that will give the INTERSECT operator greater precedence than the other set operators.

# Oracle Server and Set Operators

- Duplicate rows are automatically eliminated except in UNION ALL.
- Column names from the first query appear in the result.
- The output is sorted in ascending order by default except in UNION ALL.

ORACLE

8 - 4

Copyright © 2009, Oracle. All rights reserved.

## Oracle Server and Set Operators

When a query uses set operators, the Oracle server eliminates duplicate rows automatically except in the case of the UNION ALL operator. The column names in the output are decided by the column list in the first SELECT statement. By default, the output is sorted in ascending order of the first column of the SELECT clause.

The corresponding expressions in the SELECT lists of the component queries of a compound query must match in number and data type. If component queries select character data, the data type of the return values is determined as follows:

- If both queries select values of CHAR data type, of equal length, the returned values have the CHAR data type of that length. If the queries select values of CHAR with different lengths, the returned value is VARCHAR2 with the length of the larger CHAR value.
- If either or both of the queries select values of VARCHAR2 data type, the returned values have the VARCHAR2 data type.

If component queries select numeric data, the data type of the return values is determined by numeric precedence. If all queries select values of the NUMBER type, the returned values have the NUMBER data type. In queries using set operators, the Oracle server does not perform implicit conversion across data type groups. Therefore, if the corresponding expressions of component queries resolve to both character data and numeric data, the Oracle server returns an error.

## Tables Used in This Lesson

The tables used in this lesson are:

- EMPLOYEES: Provides details regarding all current employees
- JOB\_HISTORY: Records the details of the start date and end date of the former job, and the job identification number and department when an employee switches jobs

ORACLE

8 - 5

Copyright © 2009, Oracle. All rights reserved.

## Tables Used in This Lesson

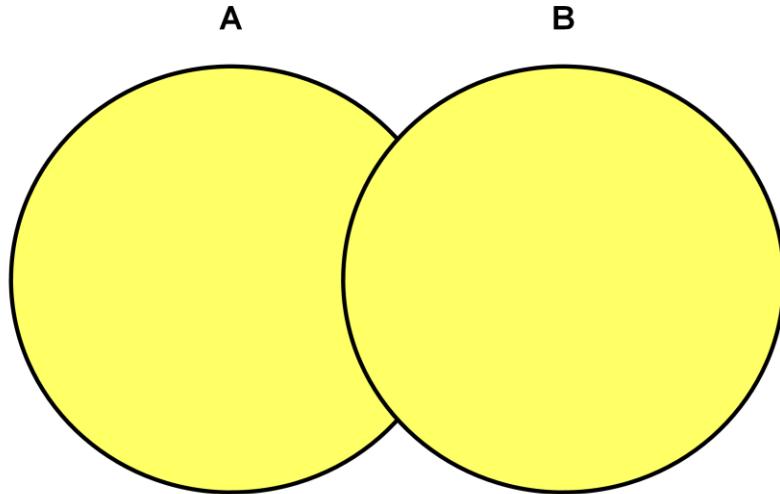
Two tables are used in this lesson: the EMPLOYEES table and the JOB\_HISTORY table.

You are already familiar with the EMPLOYEES table that stores employee details such as a unique identification number, email address, job identification (such as ST\_CLERK, SA REP, and so on), salary, manager, and so on.

Some of the employees have been with the company for a long time and have switched to different jobs. This is monitored using the JOB\_HISTORY table. When an employee switches jobs, the details of the start date and end date of the former job, the job\_id (such as ST\_CLERK, SA REP, and so on), and the department are recorded in the JOB\_HISTORY table.

The structure and data from the EMPLOYEES and JOB\_HISTORY tables are shown on the following pages.

## UNION Operator



**The UNION operator returns rows from both queries after eliminating duplications.**

ORACLE

8 - 9

Copyright © 2009, Oracle. All rights reserved.

## UNION Operator

The UNION operator returns all rows that are selected by either query. Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows.

### Guidelines

- The number of columns being selected must be the same.
- The data types of the columns being selected must be in the same data type group (such as numeric or character).
- The names of the columns need not be identical.
- UNION operates over all of the columns being selected.
- NULL values are not ignored during duplicate checking.
- By default, the output is sorted in ascending order of the columns of the SELECT clause.

## Using the UNION Operator

Display the current and previous job details of all employees.  
Display each employee only once.

```
SELECT employee_id, job_id  
FROM employees  
UNION  
SELECT employee_id, job_id  
FROM job_history;
```

EMPLOYEE_ID	JOB_ID
1	100 AD_PRES
2	101 AC_ACCOUNT
...	
22	200 AC_ACCOUNT
23	200 AD_ASST
...	
27	205 AC_MGR
28	206 AC_ACCOUNT

ORACLE

8 - 10

Copyright © 2009, Oracle. All rights reserved.

## Using the UNION Operator

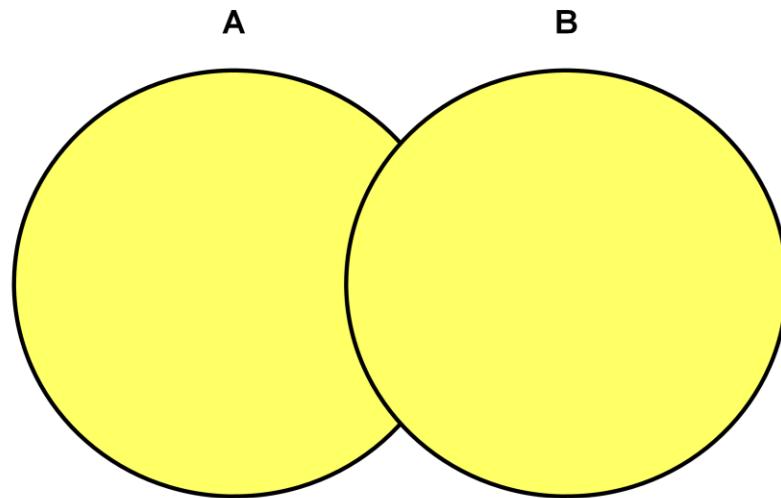
The UNION operator eliminates any duplicate records. If records that occur in both the EMPLOYEES and the JOB\_HISTORY tables are identical, the records are displayed only once. Observe in the output shown in the slide that the record for the employee with the EMPLOYEE\_ID 200 appears twice because the JOB\_ID is different in each row.

Consider the following example:

```
SELECT employee_id, job_id, department_id  
FROM employees  
UNION  
SELECT employee_id, job_id, department_id  
FROM job_history;
```

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
1	100 AD_PRES	90
...		
22	200 AC_ACCOUNT	90
23	200 AD_ASST	10
24	200 AD_ASST	90
...		
29	206 AC_ACCOUNT	110

## UNION ALL Operator



The UNION ALL operator returns rows from both queries, including all duplications.

ORACLE

8 - 11

Copyright © 2009, Oracle. All rights reserved.

### UNION ALL Operator

Use the UNION ALL operator to return all rows from multiple queries.

#### Guidelines

The guidelines for UNION and UNION ALL are the same, with the following two exceptions that pertain to UNION ALL: Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.

## Using the UNION ALL Operator

Display the current and previous departments of all employees.

```
SELECT employee_id, job_id, department_id
FROM   employees
UNION ALL
SELECT employee_id, job_id, department_id
FROM   job_history
ORDER BY employee_id;
```

	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
1	100	AD_PRES	90
...			
17	149	SA_MAN	80
18	174	SA_REP	80
19	176	SA_REP	80
20	176	SA_MAN	80
21	176	SA_REP	80
22	178	SA_REP	(null)
23	200	AD_ASST	10
...			
30	206	AC_ACCOUNT	110

ORACLE

8 - 12

Copyright © 2009, Oracle. All rights reserved.

## Using the UNION ALL Operator

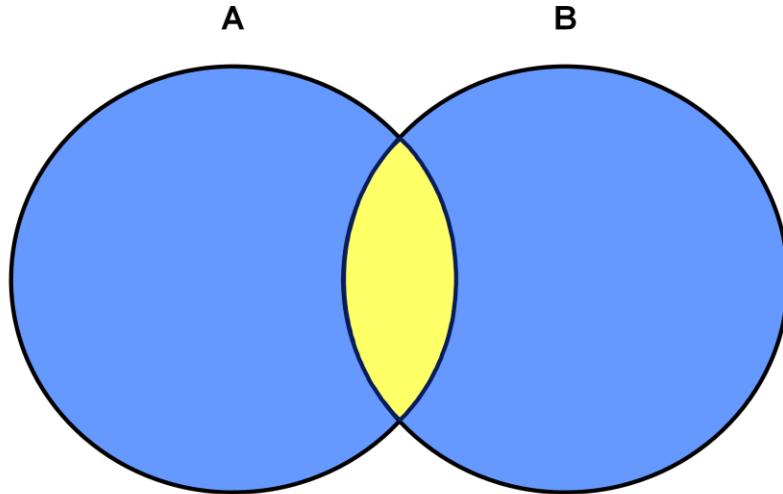
In the example, 30 rows are selected. The combination of the two tables totals to 30 rows. The UNION ALL operator does not eliminate duplicate rows. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates. Consider the query in the slide, now written with the UNION clause:

```
SELECT      employee_id, job_id, department_id
FROM        employees
UNION
SELECT      employee_id, job_id, department_id
FROM        job_history
ORDER BY    employee_id;
```

The preceding query returns 29 rows. This is because it eliminates the following row (because it is a duplicate):

176	SA_REP	80
-----	--------	----

## INTERSECT Operator



The **INTERSECT** operator returns rows that are common to both queries.

ORACLE

8 - 13

Copyright © 2009, Oracle. All rights reserved.

### INTERSECT Operator

Use the **INTERSECT** operator to return all rows that are common to multiple queries.

#### Guidelines

- The number of columns and the data types of the columns being selected by the **SELECT** statements in the queries must be identical in all the **SELECT** statements used in the query. The names of the columns, however, need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- **INTERSECT** does not ignore **NULL** values.

## Using the INTERSECT Operator

Display the employee IDs and job IDs of those employees who currently have a job title that is the same as their previous one (that is, they changed jobs but have now gone back to doing the same job they did previously).

```
SELECT employee_id, job_id
FROM   employees
INTERSECT
SELECT employee_id, job_id
FROM   job_history;
```

	EMPLOYEE_ID	JOB_ID
1		176 SA_REP
2		200 AD_ASST

ORACLE

8 - 14

Copyright © 2009, Oracle. All rights reserved.

## Using the INTERSECT Operator

In the example in this slide, the query returns only those records that have the same values in the selected columns in both tables.

What will be the results if you add the DEPARTMENT\_ID column to the SELECT statement from the EMPLOYEES table and add the DEPARTMENT\_ID column to the SELECT statement from the JOB\_HISTORY table, and run this query? The results may be different because of the introduction of another column whose values may or may not be duplicates.

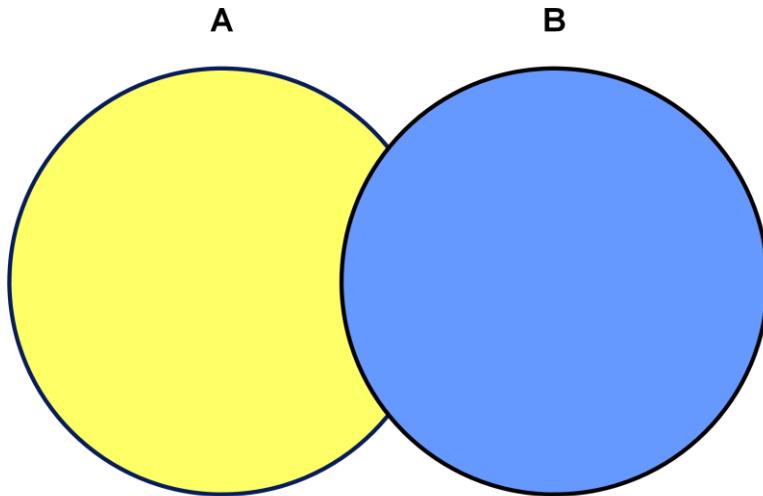
### Example:

```
SELECT employee_id, job_id, department_id
FROM   employees
INTERSECT
SELECT employee_id, job_id, department_id
FROM   job_history;
```

	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
1		176 SA_REP	80

Employee 200 is no longer part of the results because the EMPLOYEES.DEPARTMENT\_ID value is different from the JOB\_HISTORY.DEPARTMENT\_ID value.

## MINUS Operator



**The MINUS operator returns all the distinct rows selected by the first query, but not present in the second query result set.**

ORACLE

8 - 15

Copyright © 2009, Oracle. All rights reserved.

### MINUS Operator

Use the MINUS operator to return all distinct rows selected by the first query, but not present in the second query result set (the first SELECT statement MINUS the second SELECT statement).

**Note:** The number of columns must be the same and the data types of the columns being selected by the SELECT statements in the queries must belong to the same data type group in all the SELECT statements used in the query. The names of the columns, however, need not be identical.

## Using the MINUS Operator

Display the employee IDs of those employees who have not changed their jobs even once.

```
SELECT employee_id  
FROM   employees  
MINUS  
SELECT employee_id  
FROM   job_history;
```

EMPLOYEE_ID
1
2
3
...
13
14
15

ORACLE

## Using the MINUS Operator

In the example in the slide, the employee IDs in the JOB\_HISTORY table are subtracted from those in the EMPLOYEES table. The results set displays the employees remaining after the subtraction; they are represented by rows that exist in the EMPLOYEES table, but do not exist in the JOB\_HISTORY table. These are the records of the employees who have not changed their jobs even once.

## Matching the SELECT Statements

- Using the UNION operator, display the location ID, department name, and the state where it is located.
- You must match the data type (using the TO\_CHAR function or any other conversion functions) when columns do not exist in one or the other table.

```
SELECT location_id, department_name "Department",
       TO_CHAR(NULL) "Warehouse location"
  FROM departments
UNION
SELECT location_id, TO_CHAR(NULL) "Department",
       state_province
  FROM locations;
```

ORACLE

8 - 17

Copyright © 2009, Oracle. All rights reserved.

## Matching the SELECT Statements

Because the expressions in the SELECT lists of the queries must match in number, you can use the dummy columns and the data type conversion functions to comply with this rule. In the slide, the name, Warehouse location, is given as the dummy column heading. The TO\_CHAR function is used in the first query to match the VARCHAR2 data type of the state\_province column that is retrieved by the second query. Similarly, the TO\_CHAR function in the second query is used to match the VARCHAR2 data type of the department\_name column that is retrieved by the first query.

The output of the query is shown:

LOCATION_ID	Department	Warehouse location
1	1400 IT	(null)
2	1400 (null)	Texas
3	1500 Shipping	(null)
4	1500 (null)	California
5	1700 Accounting	(null)
6	1700 Administration	(null)
7	1700 Contracting	(null)
8	1700 Executive	(null)
...		

## Matching the SELECT Statement: Example

Using the UNION operator, display the employee ID, job ID, and salary of all employees.

```
SELECT employee_id, job_id, salary
FROM   employees
UNION
SELECT employee_id, job_id, 0
FROM   job_history;
```

	EMPLOYEE_ID	JOB_ID	SALARY
1	100	AD_PRES	24000
2	101	AC_ACCOUNT	0
3	101	AC_MGR	0
4	101	AD_VP	17000
5	102	AD_VP	17000
...			
29	205	AC_MGR	12000
30	206	AC_ACCOUNT	8300

ORACLE

## Matching the SELECT Statement: Example

The EMPLOYEES and JOB\_HISTORY tables have several columns in common (for example, EMPLOYEE\_ID, JOB\_ID, and DEPARTMENT\_ID). But what if you want the query to display the employee ID, job ID, and salary using the UNION operator, knowing that the salary exists only in the EMPLOYEES table?

The code example in the slide matches the EMPLOYEE\_ID and JOB\_ID columns in the EMPLOYEES and JOB\_HISTORY tables. A literal value of 0 is added to the JOB\_HISTORY SELECT statement to match the numeric SALARY column in the EMPLOYEES SELECT statement.

In the results shown in the slide, each row in the output that corresponds to a record from the JOB\_HISTORY table contains a 0 in the SALARY column.

## Using the ORDER BY Clause in Set Operations

- The ORDER BY clause can appear only once at the end of the compound query.
- Component queries cannot have individual ORDER BY clauses.
- The ORDER BY clause recognizes only the columns of the first SELECT query.
- By default, the first column of the first SELECT query is used to sort the output in an ascending order.

ORACLE

8 - 19

Copyright © 2009, Oracle. All rights reserved.

## Using the ORDER BY Clause in Set Operations

The ORDER BY clause can be used only once in a compound query. If used, the ORDER BY clause must be placed at the end of the query. The ORDER BY clause accepts the column name or an alias. By default, the output is sorted in ascending order in the first column of the first SELECT query.

**Note:** The ORDER BY clause does not recognize the column names of the second SELECT query. To avoid confusion over column names, it is a common practice to ORDER BY column positions.

For example, in the following statement, the output will be shown in ascending order of job\_id.

```
SELECT employee_id, job_id, salary
  FROM employees
UNION
SELECT employee_id, job_id, 0
  FROM job_history
 ORDER BY 2;
```

If you omit ORDER BY, by default, the output will be sorted in ascending order of employee\_id. You cannot use the columns from the second query to sort the output.

# 9

## Manipulating Data

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

# Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A *transaction* consists of a collection of DML statements that form a logical unit of work.

ORACLE

9 - 2

Copyright © 2009, Oracle. All rights reserved.

## Data Manipulation Language

Data manipulation language (DML) is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a *transaction*.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decreasing the savings account, increasing the checking account, and recording the transaction in the transaction journal. The Oracle server must guarantee that all the three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

### Note

- Most of the DML statements in this lesson assume that no constraints on the table are violated. Constraints are discussed later in this course.
- In SQL Developer, click the Run Script icon or press [F5] to run the DML statements. The feedback messages will be shown on the Script Output tabbed page.

# Adding a New Row to a Table

**DEPARTMENTS**

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10 Administration	200	1700	
2	20 Marketing	201	1800	
3	50 Shipping	124	1500	
4	60 IT	103	1400	
5	80 Sales	149	2500	
6	90 Executive	100	1700	
7	110 Accounting	205	1700	
8	190 Contracting	(null)	1700	

**New row**

**Insert new row into the DEPARTMENTS table.**

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	70 Public Relations	100	1700	
2	10 Administration	200	1700	
3	20 Marketing	201	1800	
4	50 Shipping	124	1500	
5	60 IT	103	1400	
6	80 Sales	149	2500	
7	90 Executive	100	1700	
8	110 Accounting	205	1700	
9	190 Contracting	(null)	1700	

**ORACLE**

## Adding a New Row to a Table

The graphic in the slide illustrates the addition of a new department to the DEPARTMENTS table.

## INSERT Statement Syntax

- Add new rows to a table by using the `INSERT` statement:

```
INSERT INTO  table [(column [, column...])]  
VALUES      (value [, value...]);
```

- With this syntax, only one row is inserted at a time.

ORACLE

## INSERT Statement Syntax

You can add new rows to a table by issuing the `INSERT` statement.

In the syntax:

<i>table</i>	Is the name of the table
<i>column</i>	Is the name of the column in the table to populate
<i>value</i>	Is the corresponding value for the column

**Note:** This statement with the `VALUES` clause adds only one row at a time to a table.

## Inserting New Rows

- Insert a new row containing values for each column.
- List values in the default order of the columns in the table.
- Optionally, list the columns in the `INSERT` clause.

```
INSERT INTO departments(department_id,
    department_name, manager_id, location_id)
VALUES (70, 'Public Relations', 100, 1700);
1 rows inserted
```

- Enclose character and date values within single quotation marks.

ORACLE

9 - 5

Copyright © 2009, Oracle. All rights reserved.

## Inserting New Rows

Because you can insert a new row that contains values for each column, the column list is not required in the `INSERT` clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column.

```
DESCRIBE departments
```

Name	Null	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

For clarity, use the column list in the `INSERT` clause.

Enclose character and date values within single quotation marks; however, it is not recommended that you enclose numeric values within single quotation marks.

## Inserting Rows with Null Values

- Implicit method: Omit the column from the column list.

```
INSERT INTO departments (department_id,
                        department_name)
VALUES          (30, 'Purchasing');

1 rows inserted
```

- Explicit method: Specify the NULL keyword in the VALUES clause.

```
INSERT INTO departments
VALUES          (100, 'Finance', NULL, NULL);

1 rows inserted
```

ORACLE

9 - 6

Copyright © 2009, Oracle. All rights reserved.

## Inserting Rows with Null Values

Method	Description
Implicit	Omit the column from the column list.
Explicit	Specify the NULL keyword in the VALUES list; specify the empty string (' ') in the VALUES list for character strings and dates.

Be sure that you can use null values in the targeted column by verifying the Null status with the DESCRIBE command.

The Oracle server automatically enforces all data types, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row.

Common errors that can occur during user input are checked in the following order:

- Mandatory value missing for a NOT NULL column
- Duplicate value violating any unique or primary key constraint
- Any value violating a CHECK constraint
- Referential integrity maintained for foreign key constraint
- Data type mismatches or values too wide to fit in column

# Inserting Special Values

The SYSDATE function records the current date and time.

```
INSERT INTO employees (employee_id,
                      first_name, last_name,
                      email, phone_number,
                      hire_date, job_id, salary,
                      commission_pct, manager_id,
                      department_id)
VALUES
(113,
 'Louis', 'Popp',
 'LPOPP', '515.124.4567',
 SYSDATE, 'AC_ACCOUNT', 6900,
 NULL, 205, 110);

1 rows inserted
```

ORACLE

9 - 7

Copyright © 2009, Oracle. All rights reserved.

## Inserting Special Values

You can use functions to enter special values in your table.

The slide example records information for employee Popp in the EMPLOYEES table. It supplies the current date and time in the HIRE\_DATE column. It uses the SYSDATE function that returns the current date and time of the database server. You may also use the CURRENT\_DATE function to get the current date in the session time zone. You can also use the USER function when inserting rows in a table. The USER function records the current username.

### Confirming Additions to the Table

```
SELECT employee_id, last_name, job_id, hire_date, commission_pct
FROM   employees
WHERE  employee_id = 113;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	COMMISSION_PCT
1	113 Popp	AC_ACCOUNT	10-JUL-09	(null)

## Inserting Specific Date and Time Values

- Add a new employee.

```
INSERT INTO employees
VALUES      (114,
              'Den', 'Raphealy',
              'DRAPHEAL', '515.127.4561',
              TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
              'SA REP', 11000, 0.2, 100, 60);
1 rows inserted
```

- Verify your addition.

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT
1	114	Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-99	SA REP	11000	0.2

ORACLE

## Inserting Specific Date and Time Values

The DD-MON-RR format is generally used to insert a date value. With the RR format, the system provides the correct century automatically.

You may also supply the date value in the DD-MON-YYYY format. This is recommended because it clearly specifies the century and does not depend on the internal RR format logic of specifying the correct century.

If a date must be entered in a format other than the default format (for example, with another century or a specific time), you must use the TO\_DATE function.

The example in the slide records information for employee Raphealy in the EMPLOYEES table. It sets the HIRE\_DATE column to be February 3, 1999.

## Creating a Script

- Use the & substitution in a SQL statement to prompt for values.
- & is a placeholder for the variable value.

```
INSERT INTO departments
    (department_id, department_name, location_id)
VALUES (&department_id, '&department_name',&location);
```

The screenshot shows three separate 'Enter Substitution Variable' dialog boxes. The first dialog has 'DEPARTMENT\_ID:' and '40' entered. The second dialog has 'DEPARTMENT\_NAME:' and 'Human Resources' entered. The third dialog has 'LOCATION:' and '2500' entered. Each dialog has an 'OK' button at the bottom right.

9 - 9

Copyright © 2009, Oracle. All rights reserved.

ORACLE

## Creating a Script

You can save commands with substitution variables to a file and execute the commands in the file. The example in the slide records information for a department in the DEPARTMENTS table.

Run the script file and you are prompted for input for each of the ampersand (&) substitution variables. After entering a value for the substitution variable, click the OK button. The values that you input are then substituted into the statement. This enables you to run the same script file over and over, but supply a different set of values each time you run it.

## 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%';
```

4 rows inserted

- Do not use the `VALUES` clause.
- Match the number of columns in the `INSERT` clause to those in the subquery.
- Inserts all the rows returned by the subquery in the table, `sales_reps`.

ORACLE

9 - 10

Copyright © 2009, 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 the example in the slide, for the `INSERT INTO` statement to work, you must have already created the `sales_reps` table using the `CREATE TABLE` statement. `CREATE TABLE` is discussed in the lesson titled “Using DDL Statements to Create and Manage Tables.”

In place of the `VALUES` clause, you use a subquery.

#### Syntax

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

In the syntax:

`table` Is the name of the table

`column` Is the name of the column in the table to populate

`subquery` Is the subquery that returns rows to 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. Zero or more rows are added depending on the number of rows returned by the subquery. To create a copy of the rows of a table, use `SELECT *` in the subquery:

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

# Changing Data in a Table

## EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	60
104	Bruce	Ernst	6000	103	(null)	60
107	Diana	Lorentz	4200	103	(null)	60
124	Kevin	Mourgos	5800	100	(null)	50

Update rows in the EMPLOYEES table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	80
104	Bruce	Ernst	6000	103	(null)	80
107	Diana	Lorentz	4200	103	(null)	80
124	Kevin	Mourgos	5800	100	(null)	50

ORACLE

## Changing Data in a Table

The slide illustrates changing the department number for employees in department 60 to department 80.

## UPDATE Statement Syntax

- Modify existing values in a table with the UPDATE statement:

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- Update more than one row at a time (if required).

ORACLE

9 - 12

Copyright © 2009, Oracle. All rights reserved.

### UPDATE Statement Syntax

You can modify the existing values in a table by using the UPDATE statement.

In the syntax:

<i>table</i>	Is the name of the table
<i>column</i>	Is the name of the column in the table to populate
<i>value</i>	Is the corresponding value or subquery for the column
<i>condition</i>	Identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

For more information, see the section on “UPDATE” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

**Note:** In general, use the primary key column in the WHERE clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the EMPLOYEES table by name is dangerous, because more than one employee may have the same name.

## Updating Rows in a Table

- Values for a specific row or rows are modified if you specify the WHERE clause:

```
UPDATE employees
SET department_id = 50
WHERE employee_id = 113;
1 rows updated
```

- Values for all the rows in the table are modified if you omit the WHERE clause:

```
UPDATE copy_emp
SET department_id = 110;
22 rows updated
```

- Specify SET column\_name= NULL to update a column value to NULL.

ORACLE

9 - 13

Copyright © 2009, Oracle. All rights reserved.

### Updating Rows in a Table

The UPDATE statement modifies the values of a specific row or rows if the WHERE clause is specified. The example in the slide shows the transfer of employee 113 (Popp) to department 50.

If you omit the WHERE clause, values for all the rows in the table are modified. Examine the updated rows in the COPY\_EMP table.

```
SELECT last_name, department_id
FROM copy_emp;
```

LAST_NAME	DEPARTMENT_ID
Whalen	110
Hartstein	110
Fay	110

...

For example, an employee who was an SA REP has now changed his job to an IT PROG. Therefore, his JOB\_ID needs to be updated and the commission field needs to be set to NULL.

```
UPDATE employees
SET job_id = 'IT_PROG', commission_pct = NULL
WHERE employee_id = 114;
```

**Note:** The COPY\_EMP table has the same data as the EMPLOYEES table.

## Updating Two Columns with a Subquery

Update employee 113's job and salary to match those of employee 205.

```
UPDATE employees
SET    job_id  = (SELECT job_id
                  FROM   employees
                  WHERE  employee_id = 205),
          salary  = (SELECT salary
                  FROM   employees
                  WHERE  employee_id = 205)
WHERE   employee_id = 113;
1 rows updated
```

ORACLE

9 - 14

Copyright © 2009, 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. The syntax is as follows:

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

The example in the slide can also be written as follows:

```
UPDATE employees
SET (job_id, salary) = (SELECT job_id, salary
                         FROM   employees
                         WHERE  employee_id = 205)
WHERE   employee_id = 113;
```

## Updating Rows Based on Another Table

Use the subqueries in the UPDATE statements to update row values in a table based on values from another table:

```
UPDATE copy_emp
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 rows updated
```

ORACLE

9 - 15

Copyright © 2009, Oracle. All rights reserved.

### Updating Rows Based on Another Table

You can use the subqueries in the UPDATE statements to update values in a table. The example in the slide updates the COPY\_EMP 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.

# Removing a Row from a Table

## DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

Delete a row from the DEPARTMENTS table:

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700

ORACLE

## Removing a Row from a Table

The Contracting department has been removed from the DEPARTMENTS table (assuming no constraints on the DEPARTMENTS table are violated), as shown by the graphic in the slide.

## DELETE Statement

You can remove existing rows from a table by using the **DELETE statement**:

```
DELETE [FROM]    table
[WHERE          condition];
```

ORACLE

9 - 17

Copyright © 2009, Oracle. All rights reserved.

### DELETE Statement Syntax

You can remove existing rows from a table by using the **DELETE statement**.

In the syntax:

*table*      Is the name of the table

*condition*    Identifies the rows to be deleted, and is composed of column names, expressions, constants, subqueries, and comparison operators

**Note:** If no rows are deleted, the message “0 rows deleted” is returned (on the Script Output tab in SQL Developer)

For more information, see the section on “**DELETE**” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

## Deleting Rows from a Table

- Specific rows are deleted if you specify the WHERE clause:

```
DELETE FROM departments  
WHERE department_name = 'Finance';  
1 rows deleted
```

- All rows in the table are deleted if you omit the WHERE clause:

```
DELETE FROM copy_emp;  
22 rows deleted
```

ORACLE

9 - 18

Copyright © 2009, Oracle. All rights reserved.

## Deleting Rows from a Table

You can delete specific rows by specifying the WHERE clause in the DELETE statement. The first example in the slide deletes the Accounting department from the DEPARTMENTS table. You can confirm the delete operation by displaying the deleted rows using the SELECT statement.

```
SELECT *  
FROM departments  
WHERE department_name = 'Finance';  
0 rows selected
```

However, if you omit the WHERE clause, all rows in the table are deleted. The second example in the slide deletes all rows from the COPY\_EMP table, because no WHERE clause was specified.

### Example:

Remove rows identified in the WHERE clause.

```
DELETE FROM employees WHERE employee_id = 114;  
1 rows deleted  
  
DELETE FROM departments WHERE department_id IN (30, 40);  
2 rows deleted
```

## Deleting Rows Based on Another Table

Use the subqueries in the `DELETE` statements to remove rows from a table based on values from another table:

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

1 rows deleted
```

ORACLE

9 - 19

Copyright © 2009, Oracle. All rights reserved.

### Deleting Rows Based on Another Table

You can use the subqueries to delete rows from a table based on values from another table. The example in the slide deletes all the employees 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.

## TRUNCATE Statement

- Removes all rows from a table, leaving the table empty and the table structure intact
- Is a data definition language (DDL) statement rather than a DML statement; cannot easily be undone
- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE copy_emp;
```

ORACLE

9 - 20

Copyright © 2009, Oracle. All rights reserved.

### TRUNCATE Statement

A more efficient method of emptying a table is by using the TRUNCATE statement.

You can use the TRUNCATE statement to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lesson.
- Truncating a table does not fire the delete triggers of the table.

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement. Disabling constraints is covered in the lesson titled “Using DDL Statements to Create and Manage Tables.”

# Database Transactions

A database transaction consists of one of the following:

- DML statements that constitute one consistent change to the data
- One DDL statement
- One data control language (DCL) statement

ORACLE

9 - 21

Copyright © 2009, Oracle. All rights reserved.

## Database Transactions

The Oracle server ensures data consistency based on transactions. Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.

Transactions consist of DML statements that constitute one consistent change to the data. For example, a transfer of funds between two accounts should include the debit in one account and the credit to another account of the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

### Transaction Types

Type	Description
Data manipulation language (DML)	Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work
Data definition language (DDL)	Consists of only one DDL statement
Data control language (DCL)	Consists of only one DCL statement

## Database Transactions: Start and End

- Begin when the first DML SQL statement is executed.
- End with one of the following events:
  - A COMMIT or ROLLBACK statement is issued.
  - A DDL or DCL statement executes (automatic commit).
  - The user exits SQL Developer or SQL\*Plus.
  - The system crashes.

ORACLE

9 - 22

Copyright © 2009, Oracle. All rights reserved.

### Database Transaction: Start and End

When does a database transaction start and end?

A transaction begins when the first DML statement is encountered and ends when one of the following occurs:

- A COMMIT or ROLLBACK statement is issued.
- A DDL statement, such as CREATE, is issued.
- A DCL statement is issued.
- The user exits SQL Developer or SQL\*Plus.
- A machine fails or the system crashes.

After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and, therefore, implicitly ends a transaction.

## **Advantages of COMMIT and ROLLBACK Statements**

With COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically-related operations

**ORACLE**

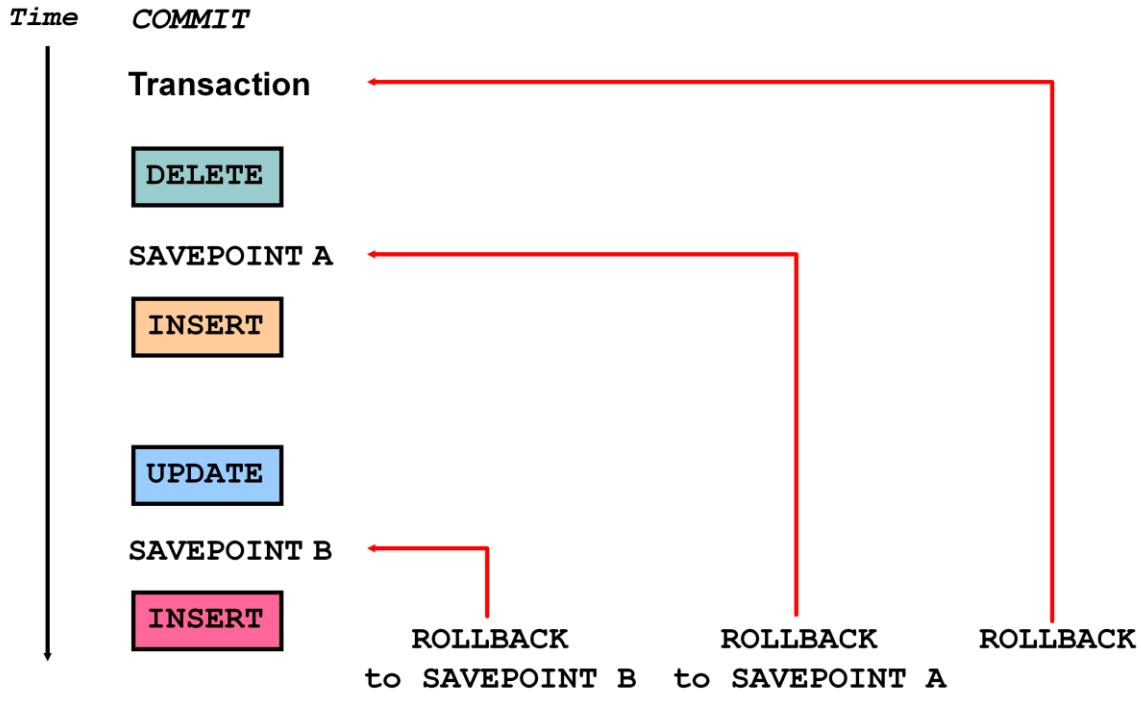
9 - 23

Copyright © 2009, Oracle. All rights reserved.

### **Advantages of COMMIT and ROLLBACK Statements**

With the COMMIT and ROLLBACK statements, you have control over making changes to the data permanent.

# Explicit Transaction Control Statements



9 - 24

Copyright © 2009, Oracle. All rights reserved.

ORACLE

## Explicit Transaction Control Statements

You can control the logic of transactions by using the COMMIT, SAVEPOINT, and ROLLBACK statements.

Statement	Description
COMMIT	COMMIT ends the current transaction by making all pending data changes permanent.
SAVEPOINT <i>name</i>	SAVEPOINT <i>name</i> marks a savepoint within the current transaction.
ROLLBACK	ROLLBACK ends the current transaction by discarding all pending data changes.
ROLLBACK TO SAVEPOINT <i>name</i>	ROLLBACK TO SAVEPOINT rolls back the current transaction to the specified savepoint, thereby discarding any changes and/or savepoints that were created after the savepoint to which you are rolling back. If you omit the TO SAVEPOINT clause, the ROLLBACK statement rolls back the entire transaction. Because savepoints are logical, there is no way to list the savepoints that you have created.

**Note:** You cannot COMMIT to a SAVEPOINT. SAVEPOINT is not ANSI-standard SQL.

## Rolling Back Changes to a Marker

- Create a marker in the current transaction by using the **SAVEPOINT statement**.
- Roll back to that marker by using the **ROLLBACK TO SAVEPOINT statement**.

```
UPDATE...
SAVEPOINT update_done;
SAVEPOINT update_done succeeded.

INSERT...
ROLLBACK TO update_done;
ROLLBACK TO succeeded.
```

ORACLE

9 - 25

Copyright © 2009, Oracle. All rights reserved.

### Rolling Back Changes to a Marker

You can create a marker in the current transaction by using the **SAVEPOINT statement**, which divides the transaction into smaller sections. You can then discard pending changes up to that marker by using the **ROLLBACK TO SAVEPOINT statement**.

Note that if you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

# Implicit Transaction Processing

- An automatic commit occurs in the following circumstances:
  - A DDL statement issued
  - A DCL statement issued
  - Normal exit from SQL Developer or SQL\*Plus, without explicitly issuing COMMIT or ROLLBACK statements
- An automatic rollback occurs when there is an abnormal termination of SQL Developer or SQL\*Plus or a system failure.

ORACLE

9 - 26

Copyright © 2009, Oracle. All rights reserved.

## Implicit Transaction Processing

Status	Circumstances
Automatic commit	DDL statement or DCL statement issued SQL Developer or SQL*Plus exited normally, without explicitly issuing COMMIT or ROLLBACK commands
Automatic rollback	Abnormal termination of SQL Developer or SQL*Plus or system failure

**Note:** In SQL\*Plus, the AUTOCOMMIT command can be toggled ON or OFF. If set to ON, each individual DML statement is committed as soon as it is executed. You cannot roll back the changes. If set to OFF, the COMMIT statement can still be issued explicitly. Also, the COMMIT statement is issued when a DDL statement is issued or when you exit SQL\*Plus. The SET AUTOCOMMIT ON/OFF command is skipped in SQL Developer. DML is committed on a normal exit from SQL Developer only if you have the Autocommit preference enabled. To enable Autocommit, perform the following:

- In the Tools menu, select Preferences. In the Preferences dialog box, expand Database and select Worksheet Parameters.
- In the right pane, select the “Autocommit in SQL Worksheet” option. Click OK.

## **State of the Data Before COMMIT or ROLLBACK**

- The previous state of the data can be recovered.
- The current user can review the results of the DML operations by using the `SELECT` statement.
- Other users *cannot* view the results of the DML statements issued by the current user.
- The affected rows are *locked*; other users cannot change the data in the affected rows.

**ORACLE**

9 - 28

Copyright © 2009, Oracle. All rights reserved.

### **State of the Data Before COMMIT or ROLLBACK**

Every data change made during the transaction is temporary until the transaction is committed.

The state of the data before COMMIT or ROLLBACK statements are issued can be described as follows:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current user can review the results of the data manipulation operations by querying the tables.
- Other users cannot view the results of the data manipulation operations made by the current user. The Oracle server institutes read consistency to ensure that each user sees data as it existed at the last commit.
- The affected rows are locked; other users cannot change the data in the affected rows.

## State of the Data After COMMIT

- Data changes are saved in the database.
- The previous state of the data is overwritten.
- All users can view the results.
- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.

ORACLE

9 - 29

Copyright © 2009, Oracle. All rights reserved.

## State of the Data After COMMIT

Make all pending changes permanent by using the COMMIT statement. Here is what happens after a COMMIT statement:

- Data changes are written to the database.
- The previous state of the data is no longer available with normal SQL queries.
- All users can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other users to perform new data changes.
- All savepoints are erased.

# Committing Data

- Make the changes:

```
DELETE FROM employees  
WHERE employee_id = 99999;  
1 rows deleted  
  
INSERT INTO departments  
VALUES (290, 'Corporate Tax', NULL, 1700);  
1 rows inserted
```

- Commit the changes:

```
COMMIT;  
COMMIT succeeded.
```

ORACLE

9 - 30

Copyright © 2009, Oracle. All rights reserved.

## Committing Data

In the example in the slide, a row is deleted from the EMPLOYEES table and a new row is inserted into the DEPARTMENTS table. The changes are saved by issuing the COMMIT statement.

### Example:

Remove departments 290 and 300 in the DEPARTMENTS table and update a row in the EMPLOYEES table. Save the data change.

```
DELETE FROM departments  
WHERE department_id IN (290, 300);  
  
UPDATE employees  
SET department_id = 80  
WHERE employee_id = 206;  
  
COMMIT;
```

## State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement:

- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;  
ROLLBACK ;
```

ORACLE

9 - 31

Copyright © 2009, Oracle. All rights reserved.

## State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement, which results in the following:

- Data changes are undone.
- The previous state of the data is restored.
- Locks on the affected rows are released.

## State of the Data After ROLLBACK: Example

```
DELETE FROM test;
25,000 rows deleted.

ROLLBACK;
Rollback complete.

DELETE FROM test WHERE id = 100;
1 row deleted.

SELECT * FROM test WHERE id = 100;
No rows selected.

COMMIT;
Commit complete.
```

ORACLE

9 - 32

Copyright © 2009, Oracle. All rights reserved.

## State of the Data After ROLLBACK: Example

While attempting to remove a record from the TEST table, you may accidentally empty the table. However, you can correct the mistake, reissue a proper statement, and make the data change permanent.

## Statement-Level Rollback

- If a single DML statement fails during execution, only that statement is rolled back.
- The Oracle server implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a COMMIT or ROLLBACK statement.

ORACLE

9 - 33

Copyright © 2009, Oracle. All rights reserved.

### Statement-Level Rollback

A part of a transaction can be discarded through an implicit rollback if a statement execution error is detected. If a single DML statement fails during execution of a transaction, its effect is undone by a statement-level rollback, but the changes made by the previous DML statements in the transaction are not discarded. They can be committed or rolled back explicitly by the user.

The Oracle server issues an implicit commit before and after any DDL statement. So, even if your DDL statement does not execute successfully, you cannot roll back the previous statement because the server issued a commit.

Terminate your transactions explicitly by executing a COMMIT or ROLLBACK statement.

# Read Consistency

- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with the changes made by another user.
- Read consistency ensures that, on the same data:
  - Readers do not wait for writers
  - Writers do not wait for readers
  - Writers wait for writers

ORACLE

9 - 34

Copyright © 2009, Oracle. All rights reserved.

## Read Consistency

Database users access the database in two ways:

- Read operations (SELECT statement)
- Write operations (INSERT, UPDATE, DELETE statements)

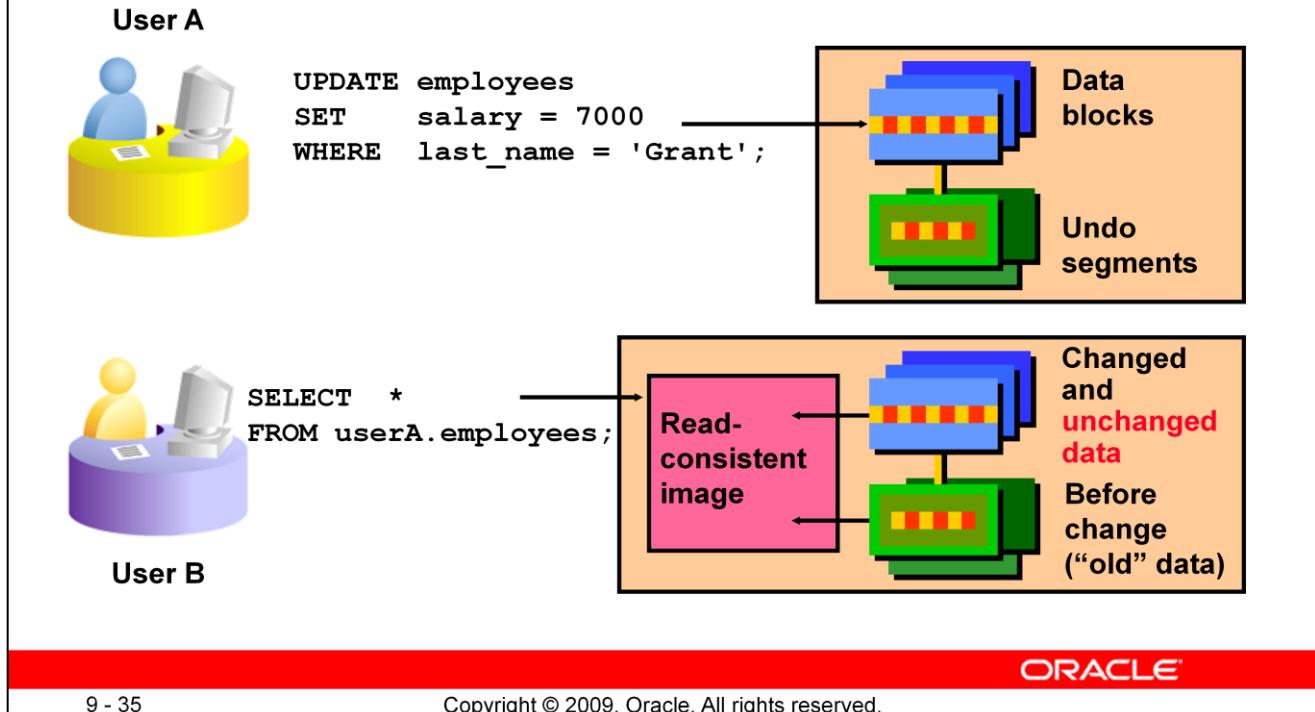
You need read consistency so that the following occur:

- The database reader and writer are ensured a consistent view of the data.
- Readers do not view data that is in the process of being changed.
- Writers are ensured that the changes to the database are done in a consistent manner.
- Changes made by one writer do not disrupt or conflict with the changes being made by another writer.

The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.

**Note:** The same user can log in to different sessions. Each session maintains read consistency in the manner described above, even if they are the same users.

# Implementing Read Consistency



## Implementing Read Consistency

Read consistency is an automatic implementation. It keeps a partial copy of the database in the undo segments. The read-consistent image is constructed from the committed data in the table and the old data that is being changed and is not yet committed from the undo segment.

When an insert, update, or delete operation is made on the database, the Oracle server takes a copy of the data before it is changed and writes it to an *undo segment*.

All readers, except the one who issued the change, see the database as it existed before the changes started; they view the undo segment's "snapshot" of the data.

Before the changes are committed to the database, only the user who is modifying the data sees the database with the alterations. Everyone else sees the snapshot in the undo segment. This guarantees that readers of the data read consistent data that is not currently undergoing change.

When a DML statement is committed, the change made to the database becomes visible to anyone issuing a `SELECT` statement *after* the commit is done. The space occupied by the *old* data in the undo segment file is freed for reuse.

If the transaction is rolled back, the changes are undone:

- The original, older version of the data in the undo segment is written back to the table.
- All users see the database as it existed before the transaction began.

# 10

## Using DDL Statements to Create and Manage Tables

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

# Database Objects

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative name to an object

ORACLE

10 - 2

Copyright © 2009, Oracle. All rights reserved.

## Database Objects

The Oracle Database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

- **Table:** Stores data
- **View:** Subset of data from one or more tables
- **Sequence:** Generates numeric values
- **Index:** Improves the performance of some queries
- **Synonym:** Gives alternative name to an object

## Oracle Table Structures

- Tables can be created at any time, even when users are using the database.
- You do not need to specify the size of a table. The size is ultimately defined by the amount of space allocated to the database as a whole. It is important, however, to estimate how much space a table will use over time.
- Table structure can be modified online.

**Note:** More database objects are available, but are not covered in this course.

# Naming Rules

Table names and column names must:

- Begin with a letter
- Be 1–30 characters long
- Contain only A–Z, a–z, 0–9, \_, \$, and #
- Not duplicate the name of another object owned by the same user
- Not be an Oracle server–reserved word

ORACLE

10 - 3

Copyright © 2009, Oracle. All rights reserved.

## Naming Rules

You name database tables and columns according to the standard rules for naming any Oracle database object:

- Table names and column names must begin with a letter and be 1–30 characters long.
- Names must contain only the characters A–Z, a–z, 0–9, \_ (underscore), \$, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle server user.
- Names must not be an Oracle server–reserved word.
  - You may also use quoted identifiers to represent the name of an object. A quoted identifier begins and ends with double quotation marks (""). If you name a schema object using a quoted identifier, you must use the double quotation marks whenever you refer to that object. Quoted identifiers can be reserved words, although this is not recommended.

## Naming Guidelines

Use descriptive names for tables and other database objects.

**Note:** Names are not case-sensitive. For example, EMPLOYEES is treated to be the same name as eMPLOYEES or eMPLoYES. However, quoted identifiers are case-sensitive.

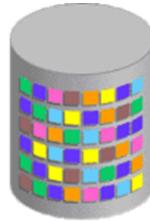
For more information, see the section on *Schema Object Names and Qualifiers* in the *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

## CREATE TABLE Statement

- You must have:
  - The CREATE TABLE privilege
  - A storage area

```
CREATE TABLE [schema.]table  
  (column datatype [DEFAULT expr] [, . . .]) ;
```

- You specify:
  - The table name
  - The column name, column data type, and column size



ORACLE

10 - 4

Copyright © 2009, Oracle. All rights reserved.

### CREATE TABLE Statement

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements that are a subset of the SQL statements used to create, modify, or remove Oracle Database structures. These statements have an immediate effect on the database and they also record information in the data dictionary.

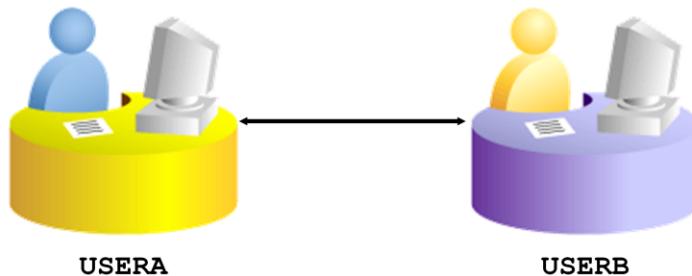
To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator (DBA) uses data control language (DCL) statements to grant privileges to users.

In the syntax:

<i>schema</i>	Is the same as the owner's name
<i>table</i>	Is the name of the table
DEFAULT <i>expr</i>	Specifies a default value if a value is omitted in the INSERT statement
<i>column</i>	Is the name of the column
<i>datatype</i>	Is the column's data type and length

## Referencing Another User's Tables

- Tables belonging to other users are not in the user's schema.
- You should use the owner's name as a prefix to those tables.



```
SELECT *  
FROM userB.employees;
```

```
SELECT *  
FROM userA.employees;
```

ORACLE

10 - 5

Copyright © 2009, Oracle. All rights reserved.

## Referencing Another User's Tables

A schema is a collection of logical structures of data or *schema objects*. A schema is owned by a database user and has the same name as that user. Each user owns a single schema.

Schema objects can be created and manipulated with SQL and include tables, views, synonyms, sequences, stored procedures, indexes, clusters, and database links.

If a table does not belong to the user, the owner's name must be prefixed to the table. For example, if there are schemas named USERA and USERB, and both have an EMPLOYEES table, then if USERA wants to access the EMPLOYEES table that belongs to USERB, USERA must prefix the table name with the schema name:

```
SELECT *  
FROM userb.employees;
```

If USERB wants to access the EMPLOYEES table that is owned by USERA, USERB must prefix the table name with the schema name:

```
SELECT *  
FROM usera.employees;
```

## DEFAULT Option

- Specify a default value for a column during an insert.

```
... hire_date DATE DEFAULT SYSDATE, ...
```

- Literal values, expressions, or SQL functions are legal values.
- Another column's name or a pseudocolumn are illegal values.
- The default data type must match the column data type.

```
CREATE TABLE hire_dates
  (id          NUMBER(8),
   hire_date DATE DEFAULT SYSDATE);
```

```
CREATE TABLE succeeded.
```

ORACLE

10 - 6

Copyright © 2009, Oracle. All rights reserved.

## DEFAULT Option

When you define a table, you can specify that a column should be given a default value by using the DEFAULT option. This option prevents null values from entering the columns when a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function (such as SYSDATE or USER), but the value cannot be the name of another column or a pseudocolumn (such as NEXTVAL or CURRVAL). The default expression must match the data type of the column.

Consider the following examples:

```
INSERT INTO hire_dates values(45, NULL);
```

The above statement will insert the null value rather than the default value.

```
INSERT INTO hire_dates(id) values(35);
```

The above statement will insert SYSDATE for the HIRE\_DATE column.

**Note:** In SQL Developer, click the Run Script icon or press [F5] to run the DDL statements. The feedback messages will be shown on the Script Output tabbed page.

# Creating Tables

- Create the table:

```
CREATE TABLE dept
  (deptno      NUMBER(2) ,
   dname       VARCHAR2(14) ,
   loc         VARCHAR2(13) ,
   create_date DATE DEFAULT SYSDATE) ;
CREATE TABLE succeeded.
```

- Confirm table creation:

```
DESCRIBE dept
```

Name	Null	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)
CREATE_DATE		DATE
4 rows selected		

ORACLE

10 - 7

Copyright © 2009, Oracle. All rights reserved.

## Creating Tables

The example in the slide creates the DEPT table with four columns: DEPTNO, DNAME, LOC, and CREATE\_DATE. The CREATE\_DATE column has a default value. If a value is not provided for an INSERT statement, the system date is automatically inserted.

To confirm that the table was created, run the DESCRIBE command.

Because creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

**Note:** You can view the list of tables you own by querying the data dictionary. For example:

```
select table_name from user_tables
```

Using data dictionary views, you can also find information about other database objects such as views, indexes, and so on. You will learn about data dictionaries in detail in the *Oracle Database 11g: SQL Fundamentals II* course.

# Data Types

Data Type	Description
VARCHAR2 ( <i>size</i> )	Variable-length character data
CHAR ( <i>size</i> )	Fixed-length character data
NUMBER ( <i>p, s</i> )	Variable-length numeric data
DATE	Date and time values
LONG	Variable-length character data (up to 2 GB)
CLOB	Character data (up to 4 GB)
RAW and LONG RAW	Raw binary data
BLOB	Binary data (up to 4 GB)
BFILE	Binary data stored in an external file (up to 4 GB)
ROWID	A base-64 number system representing the unique address of a row in its table

ORACLE

10 - 8

Copyright © 2009, Oracle. All rights reserved.

## Data Types

When you identify a column for a table, you need to provide a data type for the column. There are several data types available:

Data Type	Description
VARCHAR2 ( <i>size</i> )	Variable-length character data (A maximum <i>size</i> must be specified: minimum <i>size</i> is 1; maximum <i>size</i> is 4,000.)
CHAR [ ( <i>size</i> ) ]	Fixed-length character data of length <i>size</i> bytes (Default and minimum <i>size</i> is 1; maximum <i>size</i> is 2,000.)
NUMBER [ ( <i>p, s</i> ) ]	Number having precision <i>p</i> and scale <i>s</i> (Precision is the total number of decimal digits and scale is the number of digits to the right of the decimal point; precision can range from 1 to 38, and scale can range from -84 to 127.)
DATE	Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.
LONG	Variable-length character data (up to 2 GB)
CLOB	Character data (up to 4 GB)

# Datetime Data Types

You can use several datetime data types:

Data Type	Description
TIMESTAMP	Date with fractional seconds
INTERVAL YEAR TO MONTH	Stored as an interval of years and months
INTERVAL DAY TO SECOND	Stored as an interval of days, hours, minutes, and seconds



ORACLE

10 - 9

Copyright © 2009, Oracle. All rights reserved.

## Datetime Data Types

Data Type	Description
TIMESTAMP	Enables storage of time as a date with fractional seconds. It stores the year, month, day, hour, minute, and the second value of the DATE data type as well as the fractional seconds value There are several variations of this data type such as WITH TIMEZONE, WITH LOCALTIMEZONE.
INTERVAL YEAR TO MONTH	Enables storage of time as an interval of years and months. Used to represent the difference between two datetime values in which the only significant portions are the year and month
INTERVAL DAY TO SECOND	Enables storage of time as an interval of days, hours, minutes, and seconds. Used to represent the precise difference between two datetime values

**Note:** These datetime data types are available with Oracle9*i* and later releases. The datetime data types are discussed in detail in the lesson titled “Managing Data in Different Time Zones” in the *Oracle Database 11g: SQL Fundamentals II* course.

Also, for more information about the datetime data types, see the sections on “TIMESTAMP Datatype,” “INTERVAL YEAR TO MONTH Datatype,” and “INTERVAL DAY TO SECOND Datatype” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

# Including Constraints

- Constraints enforce rules at the table level.
- Constraints prevent the deletion of a table if there are dependencies.
- The following constraint types are valid:
  - NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK



ORACLE

10 - 10

Copyright © 2009, Oracle. All rights reserved.

## Constraints

The Oracle server uses constraints to prevent invalid data entry into tables.

You can use constraints to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables.
- Provide rules for Oracle tools, such as Oracle Developer.

## Data Integrity Constraints

Constraint	Description
NOT NULL	Specifies that the column cannot contain a null value
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table
PRIMARY KEY	Uniquely identifies each row of the table
FOREIGN KEY	Establishes and enforces a referential integrity between the column and a column of the referenced table such that values in one table match values in another table.
CHECK	Specifies a condition that must be true

# Constraint Guidelines

- You can name a constraint, or the Oracle server generates a name by using the `SYS_Cn` format.
- Create a constraint at either of the following times:
  - At the same time as the creation of the table
  - After the creation of the table
- Define a constraint at the column or table level.
- View a constraint in the data dictionary.

ORACLE

10 - 11

Copyright © 2009, Oracle. All rights reserved.

## Constraint Guidelines

All constraints are stored in the data dictionary. Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object-naming rules, except that the name cannot be the same as another object owned by the same user. If you do not name your constraint, the Oracle server generates a name with the format `SYS_Cn`, where  $n$  is an integer so that the constraint name is unique.

Constraints can be defined at the time of table creation or after the creation of the table. You can define a constraint at the column or table level. Functionally, a table-level constraint is the same as a column-level constraint.

For more information, see the section on “Constraints” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

# Defining Constraints

- Syntax:

```
CREATE TABLE [schema.]table  
  (column datatype [DEFAULT expr]  
   [column_constraint],  
   ...  
   [table_constraint] [, . . .]) ;
```

- Column-level constraint syntax:

```
column [CONSTRAINT constraint_name] constraint_type,
```

- Table-level constraint syntax:

```
column, . . .  
[CONSTRAINT constraint_name] constraint_type  
(column, . . .),
```

ORACLE

10 - 12

Copyright © 2009, Oracle. All rights reserved.

## Defining Constraints

The slide gives the syntax for defining constraints when creating a table. You can create constraints at either the column level or table level. Constraints defined at the column level are included when the column is defined. Table-level constraints are defined at the end of the table definition and must refer to the column or columns on which the constraint pertains in a set of parentheses. It is mainly the syntax that differentiates the two; otherwise, functionally, a column-level constraint is the same as a table-level constraint.

NOT NULL constraints must be defined at the column level.

Constraints that apply to more than one column must be defined at the table level.

In the syntax:

schema Is the same as the owner's name

table Is the name of the table

DEFAULT expr Specifies a default value to be used if a value is omitted in the  
INSERT statement

column Is the name of the column

datatype Is the column's data type and length

column\_constraint Is an integrity constraint as part of the column definition

table\_constraint Is an integrity constraint as part of the table definition

# Defining Constraints

- Example of a column-level constraint:

```
CREATE TABLE employees(
    employee_id  NUMBER(6)
        CONSTRAINT emp_emp_id_pk PRIMARY KEY,
    first_name    VARCHAR2(20),
    ...);
```

1

- Example of a table-level constraint:

```
CREATE TABLE employees(
    employee_id  NUMBER(6),
    first_name    VARCHAR2(20),
    ...
    job_id        VARCHAR2(10) NOT NULL,
    CONSTRAINT emp_emp_id_pk
        PRIMARY KEY (EMPLOYEE_ID));
```

2

ORACLE

10 - 13

Copyright © 2009, Oracle. All rights reserved.

## Defining Constraints (continued)

Constraints are usually created at the same time as the table. Constraints can be added to a table after its creation and also be temporarily disabled.

Both examples in the slide create a primary key constraint on the EMPLOYEE\_ID column of the EMPLOYEES table.

1. The first example uses the column-level syntax to define the constraint.
2. The second example uses the table-level syntax to define the constraint.

More details about the primary key constraint are provided later in this lesson.

## NOT NULL Constraint

Ensures that null values are not permitted for the column:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID	EMAIL	PHONE_NUMBER	HIRE_DATE
100	Steven	King	24000	(null)	90	SKING	515.123.4567	17-JUN-87
101	Neena	Kochhar	17000	(null)	90	NKOCHHAR	515.123.4568	21-SEP-89
102	Lex	De Haan	17000	(null)	90	LDEHAAN	515.123.4569	13-JAN-93
103	Alexander	Hunold	9000	(null)	60	AHUNOLD	590.423.4567	03-JAN-90
104	Bruce	Ernst	6000	(null)	60	BERNST	590.423.4568	21-MAY-91
107	Diana	Lorentz	4200	(null)	60	DLORENTZ	590.423.5567	07-FEB-99
124	Kevin	Mourgos	5800	(null)	50	KMOURGOS	650.123.5234	16-NOV-99
141	Trenna	Rajs	3500	(null)	50	TRAJS	650.121.8009	17-OCT-95
142	Curtis	Davies	3100	(null)	50	CDAVIES	650.121.2994	29-JAN-97
143	Randall	Matos	2600	(null)	50	RMATOS	650.121.2874	15-MAR-98
144	Peter	Vargas	2500	(null)	50	PVARGAS	650.121.2004	09-JUL-98
149	Eleni	Zlotkey	10500	0.2	80	EZLOTKEY	011.44.1344.429018	29-JAN-00
174	Ellen	Abel	11000	0.3	80	EABEL	011.44.1644.429267	11-MAY-96
176	Jonathon	Taylor	8600	0.2	80	JTAYLOR	011.44.1644.429265	24-MAR-98
178	Kimberely	Grant	7000	0.15	(null)	KGRANT	011.44.1644.429263	24-MAY-99
200	Jennifer	Whalen	4400	(null)	10	JWHALEN	515.123.4444	17-SEP-87
201	Michael	Hartstein	13000	(null)	20	MHARTSTE	515.123.5555	17-FEB-96
202	Pat	Fay	6000	(null)	20	PFAY	603.123.6666	17-AUG-97
205	Shelley	Higgins	12000	(null)	110	SHIGGINS	515.123.8080	07-JUN-94
206	William	Gietz	8300	(null)	110	WGIEWTZ	515.123.8181	07-JUN-94

NOT NULL constraint  
(Primary Key enforces  
NOT NULL constraint.)

NOT NULL  
constraint

Absence of NOT NULL constraint  
(Any row can contain a null  
value for this column.)

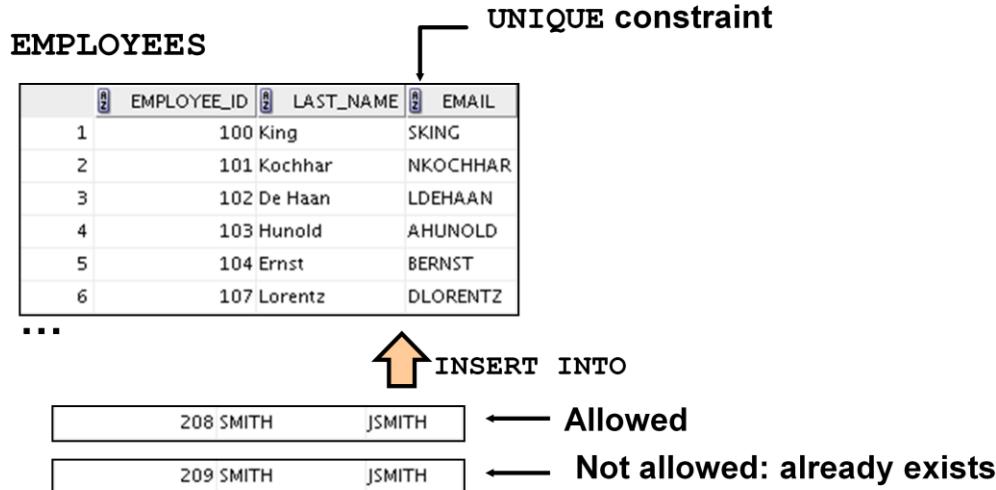
ORACLE

## NOT NULL Constraint

The NOT NULL constraint ensures that the column contains no null values. Columns without the NOT NULL constraint can contain null values by default. NOT NULL constraints must be defined at the column level. In the EMPLOYEES table, the EMPLOYEE\_ID column inherits a NOT NULL constraint as it is defined as a primary key. Otherwise, the LAST\_NAME, EMAIL, HIRE\_DATE, and JOB\_ID columns have the NOT NULL constraint enforced on them.

**Note:** Primary key constraint is discussed in detail later in this lesson.

# UNIQUE Constraint



ORACLE

10 - 15

Copyright © 2009, Oracle. All rights reserved.

## UNIQUE Constraint

A **UNIQUE** key integrity constraint requires that every value in a column or a set of columns (key) be unique—that is, no two rows of a table can have duplicate values in a specified column or a set of columns. The column (or set of columns) included in the definition of the **UNIQUE** key constraint is called the *unique key*. If the **UNIQUE** constraint comprises more than one column, that group of columns is called a *composite unique key*.

**UNIQUE** constraints enable the input of nulls unless you also define **NOT NULL** constraints for the same columns. In fact, any number of rows can include nulls for columns without the **NOT NULL** constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite **UNIQUE** key) always satisfies a **UNIQUE** constraint.

**Note:** Because of the search mechanism for the **UNIQUE** constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite **UNIQUE** key constraint.

## UNIQUE Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
    ...
    CONSTRAINT emp_email_uk UNIQUE(email));
```

ORACLE

10 - 16

Copyright © 2009, Oracle. All rights reserved.

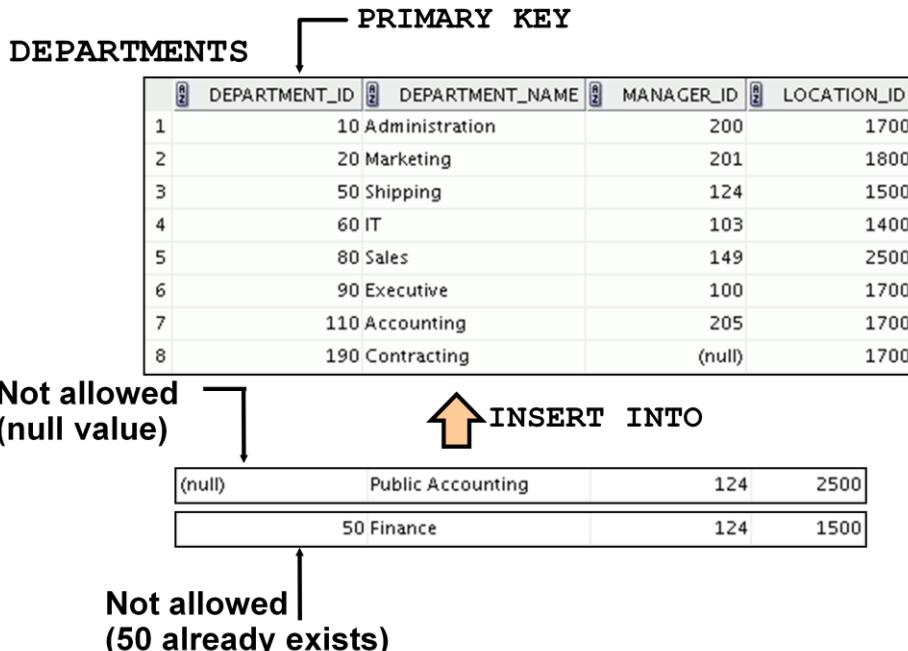
### UNIQUE Constraint (continued)

UNIQUE constraints can be defined at the column level or table level. You define the constraint at the table level when you want to create a composite unique key. A composite key is defined when there is not a single attribute that can uniquely identify a row. In that case, you can have a unique key that is composed of two or more columns, the combined value of which is always unique and can identify rows.

The example in the slide applies the UNIQUE constraint to the EMAIL column of the EMPLOYEES table. The name of the constraint is EMP\_EMAIL\_UK.

**Note:** The Oracle server enforces the UNIQUE constraint by implicitly creating a unique index on the unique key column or columns.

## PRIMARY KEY Constraint



ORACLE

10 - 17

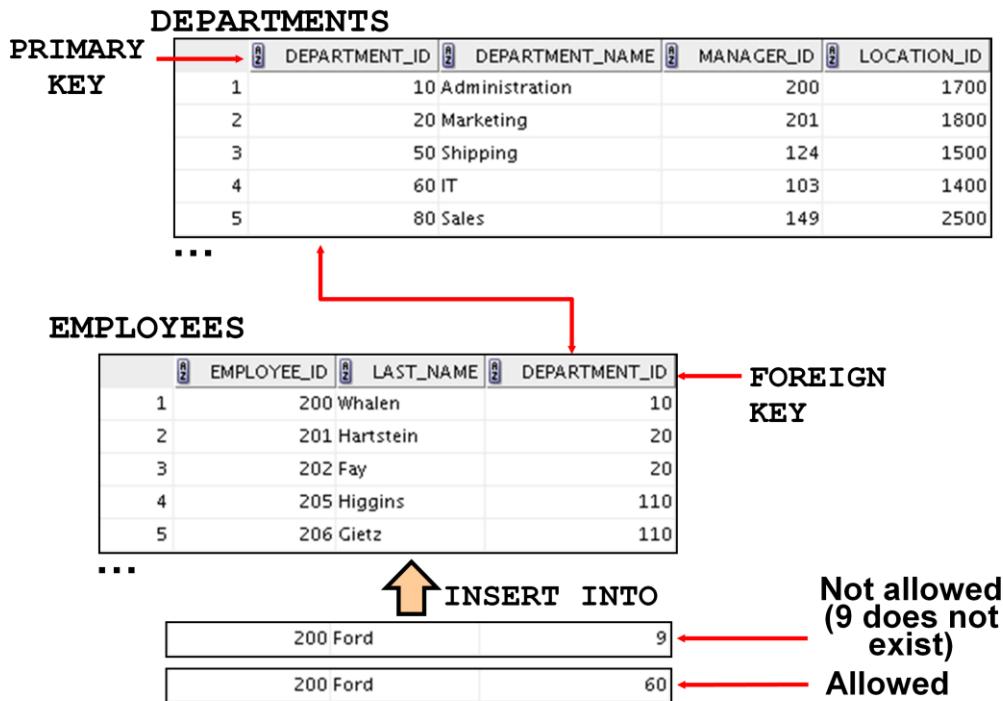
Copyright © 2009, Oracle. All rights reserved.

## PRIMARY KEY Constraint

A PRIMARY KEY constraint creates a primary key for the table. Only one primary key can be created for each table. The PRIMARY KEY constraint is a column or a set of columns that uniquely identifies each row in a table. This constraint enforces the uniqueness of the column or column combination and ensures that no column that is part of the primary key can contain a null value.

**Note:** Because uniqueness is part of the primary key constraint definition, the Oracle server enforces the uniqueness by implicitly creating a unique index on the primary key column or columns.

## FOREIGN KEY Constraint



ORACLE

### FOREIGN KEY Constraint

The FOREIGN KEY (or referential integrity) constraint designates a column or a combination of columns as a foreign key and establishes a relationship with a primary key or a unique key in the same table or a different table.

In the example in the slide, DEPARTMENT\_ID has been defined as the foreign key in the EMPLOYEES table (dependent or child table); it references the DEPARTMENT\_ID column of the DEPARTMENTS table (the referenced or parent table).

### Guidelines

- A foreign key value must match an existing value in the parent table or be NULL.
- Foreign keys are based on data values and are purely logical, rather than physical, pointers.

## FOREIGN KEY Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
    ...
    department_id    NUMBER(4),
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id)
        REFERENCES departments(department_id),
    CONSTRAINT emp_email_uk UNIQUE(email));
```

ORACLE

10 - 19

Copyright © 2009, Oracle. All rights reserved.

### FOREIGN KEY Constraint (continued)

FOREIGN KEY constraints can be defined at the column or table constraint level. A composite foreign key must be created by using the table-level definition.

The example in the slide defines a FOREIGN KEY constraint on the DEPARTMENT\_ID column of the EMPLOYEES table, using table-level syntax. The name of the constraint is EMP\_DEPT\_FK.

The foreign key can also be defined at the column level, provided that the constraint is based on a single column. The syntax differs in that the keywords FOREIGN KEY do not appear. For example:

```
CREATE TABLE employees
(
    ...
    department_id NUMBER(4) CONSTRAINT emp_deptid_fk
        REFERENCES departments(department_id),
    ...
)
```

## FOREIGN KEY Constraint: Keywords

- FOREIGN KEY: Defines the column in the child table at the table-constraint level
- REFERENCES: Identifies the table and column in the parent table
- ON DELETE CASCADE: Deletes the dependent rows in the child table when a row in the parent table is deleted
- ON DELETE SET NULL: Converts dependent foreign key values to null

ORACLE

10 - 20

Copyright © 2009, Oracle. All rights reserved.

## FOREIGN KEY Constraint: Keywords

The foreign key is defined in the child table and the table containing the referenced column is the parent table. The foreign key is defined using a combination of the following keywords:

- FOREIGN KEY is used to define the column in the child table at the table-constraint level.
- REFERENCES identifies the table and the column in the parent table.
- ON DELETE CASCADE indicates that when a row in the parent table is deleted, the dependent rows in the child table are also deleted.
- ON DELETE SET NULL indicates that when a row in the parent table is deleted, the foreign key values are set to null.

The default behavior is called the *restrict rule*, which disallows the update or deletion of referenced data.

Without the ON DELETE CASCADE or the ON DELETE SET NULL options, the row in the parent table cannot be deleted if it is referenced in the child table.

## CHECK Constraint

- Defines a condition that each row must satisfy
- The following expressions are not allowed:
  - References to CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
  - Calls to SYSDATE, UID, USER, and USERENV functions
  - Queries that refer to other values in other rows

```
..., salary NUMBER(2)
CONSTRAINT emp_salary_min
    CHECK (salary > 0),...
```

ORACLE

10 - 21

Copyright © 2009, Oracle. All rights reserved.

### CHECK Constraint

The CHECK constraint defines a condition that each row must satisfy. The condition can use the same constructs as the query conditions, with the following exceptions:

- References to the CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
- Calls to SYSDATE, UID, USER, and USERENV functions
- Queries that refer to other values in other rows

A single column can have multiple CHECK constraints that refer to the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

CHECK constraints can be defined at the column level or table level.

```
CREATE TABLE employees
(
    ...
    salary NUMBER(8,2) CONSTRAINT emp_salary_min
        CHECK (salary > 0),
    ...
)
```

## CREATE TABLE: Example

```
CREATE TABLE employees
( employee_id      NUMBER(6)
  CONSTRAINT emp_employee_id PRIMARY KEY
, first_name       VARCHAR2(20)
, last_name        VARCHAR2(25)
  CONSTRAINT emp_last_name_nn NOT NULL
, email            VARCHAR2(25)
  CONSTRAINT emp_email_nn    NOT NULL
  CONSTRAINT emp_email_uk    UNIQUE
, phone_number     VARCHAR2(20)
, hire_date        DATE
  CONSTRAINT emp_hire_date_nn NOT NULL
, job_id           VARCHAR2(10)
  CONSTRAINT emp_job_nn    NOT NULL
, salary            NUMBER(8,2)
  CONSTRAINT emp_salary_ck   CHECK (salary>0)
, commission_pct   NUMBER(2,2)
, manager_id       NUMBER(6)
  CONSTRAINT emp_manager_fk REFERENCES
          employees (employee_id)
, department_id    NUMBER(4)
  CONSTRAINT emp_dept_fk    REFERENCES
          departments (department_id));
```

ORACLE

10 - 22

Copyright © 2009, Oracle. All rights reserved.

## CREATE TABLE: Example

The example in the slide shows the statement that is used to create the EMPLOYEES table in the HR schema.

## Violating Constraints

```
UPDATE employees  
SET department_id = 55  
WHERE department_id = 110;
```

```
Error starting at line 1 in command:  
UPDATE employees  
SET department_id = 55  
WHERE department_id = 110  
Error report:  
SQL Error: ORA-02291: integrity constraint (ORA1.EMP_DEPT_FK) violated - parent key not found  
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"  
*Cause: A foreign key value has no matching primary key value.
```

Department 55 does not exist.

ORACLE

10 - 23

Copyright © 2009, Oracle. All rights reserved.

## Violating Constraints

When you have constraints in place on columns, an error is returned if you try to violate the constraint rule. For example, if you try to update a record with a value that is tied to an integrity constraint, an error is returned.

In the example in the slide, department 55 does not exist in the parent table, DEPARTMENTS, and so you receive the “parent key not found” violation ORA-02291.

# Violating Constraints

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

```
DELETE FROM departments  
WHERE department_id = 60;
```

```
Error starting at line 1 in command:  
DELETE FROM departments  
WHERE department_id = 60  
Error report:  
SQL Error: ORA-02292: integrity constraint (ORA1.JHIST_DEPT_FK) violated - child record found  
02292. 00000 - "integrity constraint (%s.%s) violated - child record found"  
*Cause:    attempted to delete a parent key value that had a foreign  
           dependency.  
*Action:   delete dependencies first then parent or disable constraint.
```

ORACLE

10 - 24

Copyright © 2009, Oracle. All rights reserved.

## Violating Constraints (continued)

If you attempt to delete a record with a value that is tied to an integrity constraint, an error is returned. The example in the slide tries to delete department 60 from the DEPARTMENTS table, but it results in an error because that department number is used as a foreign key in the EMPLOYEES table. If the parent record that you attempt to delete has child records, you receive the “child record found” violation ORA-02292.

The following statement works because there are no employees in department 70:

```
DELETE FROM departments  
WHERE department_id = 70;
```

1 rows deleted

# Creating a Table Using a Subquery

- Create a table and insert rows by combining the CREATE TABLE statement and the AS *subquery* option.

```
CREATE TABLE table
    [ (column, column...) ]
AS subquery;
```

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.

ORACLE

10 - 25

Copyright © 2009, Oracle. All rights reserved.

## Creating a Table Using a Subquery

A second method for creating a table is to apply the AS *subquery* clause, which both creates the table and inserts rows returned from the subquery.

In the syntax:

<i>table</i>	Is the name of the table
<i>column</i>	Is the name of the column, default value, and integrity constraint
<i>subquery</i>	Is the SELECT statement that defines the set of rows to be inserted into the new table

## Guidelines

- The table is created with the specified column names, and the rows retrieved by the SELECT statement are inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery SELECT list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery.

# Creating a Table Using a Subquery

```
CREATE TABLE dept80
  AS
    SELECT employee_id, last_name,
           salary*12 ANNSAL,
           hire_date
      FROM employees
     WHERE department_id = 80;
CREATE TABLE succeeded.
```

```
DESCRIBE dept80
```

Name	Null	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

ORACLE

10 - 26

Copyright © 2009, Oracle. All rights reserved.

## Creating a Table Using a Subquery (continued)

The example in the slide creates a table named DEPT80, which contains details of all the employees working in department 80. Notice that the data for the DEPT80 table comes from the EMPLOYEES table.

You can verify the existence of a database table and check the column definitions by using the DESCRIBE command.

However, be sure to provide a column alias when selecting an expression. The expression SALARY\*12 is given the alias ANNSAL. Without the alias, the following error is generated:

```
Error starting at line 1 in command:
CREATE TABLE dept80
  AS  SELECT employee_id, last_name,
        salary*12 ,
        hire_date FROM employees WHERE department_id = 80
Error at Command Line:3 Column:18
Error report:
SQL Error: ORA-00998: must name this expression with a column alias
00998. 00000 -  "must name this expression with a column alias"
*Cause:
*Action:
```

## **ALTER TABLE Statement**

Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column definition
- Define a default value for the new column
- Drop a column
- Rename a column
- Change table to read-only status

**ORACLE**

10 - 27

Copyright © 2009, Oracle. All rights reserved.

### **ALTER TABLE Statement**

After you create a table, you may need to change the table structure for any of the following reasons:

- You omitted a column.
- Your column definition or its name needs to be changed.
- You need to remove columns.
- You want to put the table into the read-only mode

You can do this by using the ALTER TABLE statement.

# Read-Only Tables

You can use the ALTER TABLE syntax to:

- Put a table into read-only mode, which prevents DDL or DML changes during table maintenance
- Put the table back into read/write mode

```
ALTER TABLE employees READ ONLY;  
  
-- perform table maintenance and then  
-- return table back to read/write mode  
  
ALTER TABLE employees READ WRITE;
```

ORACLE

10 - 28

Copyright © 2009, Oracle. All rights reserved.

## Read-Only Tables

With Oracle Database 11g, you can specify READ ONLY to place a table in the read-only mode. When the table is in the READ-ONLY mode, you cannot issue any DML statements that affect the table or any SELECT . . . FOR UPDATE statements. You can issue DDL statements as long as they do not modify any data in the table. Operations on indexes associated with the table are allowed when the table is in the READ ONLY mode.

Specify READ/WRITE to return a read-only table to the read/write mode.

**Note:** You can drop a table that is in the READ ONLY mode. The DROP command is executed only in the data dictionary, so access to the table contents is not required. The space used by the table will not be reclaimed until the tablespace is made read/write again, and then the required changes can be made to the block segment headers, and so on. For information about the ALTER TABLE statement, see the course titled *Oracle Database 10g SQL Fundamentals II*.

# Dropping a Table

- Moves a table to the recycle bin
- Removes the table and all its data entirely if the PURGE clause is specified
- Invalidates dependent objects and removes object privileges on the table

```
DROP TABLE dept80;  
DROP TABLE dept80 succeeded.
```

ORACLE

10 - 29

Copyright © 2009, Oracle. All rights reserved.

## Dropping a Table

The `DROP TABLE` statement moves a table to the recycle bin or removes the table and all its data from the database entirely. Unless you specify the PURGE clause, the `DROP TABLE` statement does not result in space being released back to the tablespace for use by other objects, and the space continues to count towards the user's space quota. Dropping a table invalidates the dependent objects and removes object privileges on the table.

When you drop a table, the database loses all the data in the table and all the indexes associated with it.

### Syntax

```
DROP TABLE table [PURGE]
```

In the syntax, *table* is the name of the table.

### Guidelines

- All the data is deleted from the table.
- Any views and synonyms remain, but are invalid.
- Any pending transactions are committed.
- Only the creator of the table or a user with the `DROP ANY TABLE` privilege can remove a table.

**Note:** Use the `FLASHBACK TABLE` statement to restore a dropped table from the recycle bin. This is discussed in detail in the course titled *Oracle Database 11g: SQL Fundamentals II*.

# 11

## Creating Other Schema Objects

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

# Database Objects

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of data retrieval queries
Synonym	Gives alternative names to objects

ORACLE

11 - 2

Copyright © 2009, Oracle. All rights reserved.

## Database Objects

There are several other objects in a database in addition to tables.

With views, you can present and hide data from the tables.

Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of data retrieval queries, you should consider creating an index. You can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.

# What Is a View?

**EMPLOYEES** table

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
104	Bruce	Ernst	BERNSTEIN	590.423.4568	20-JAN-94	IT_PROG	6000
105	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300

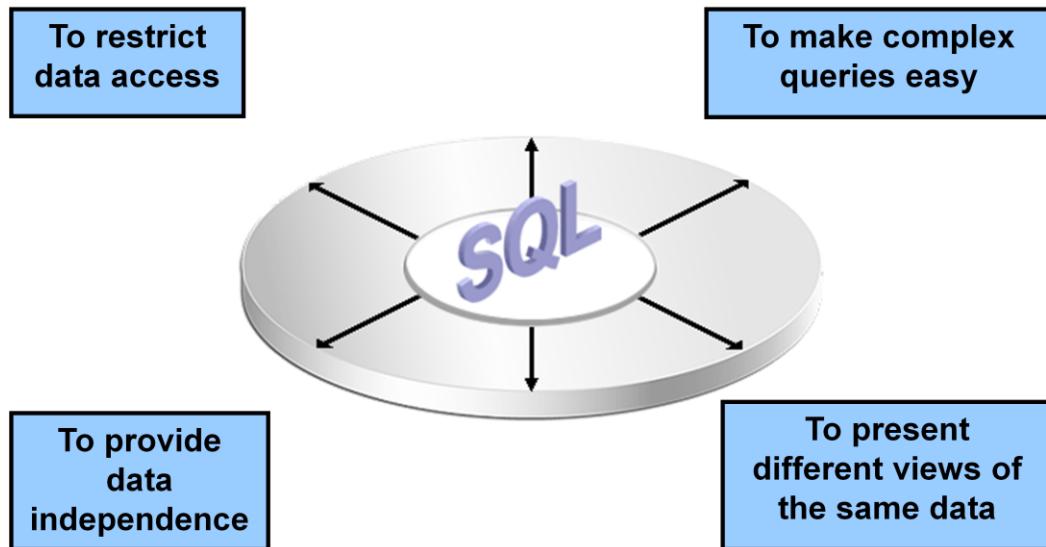
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
100	Steven	King	24000
101	Neena	Kochhar	17000
102	Lex	De Haan	17000
103	Alexander	Hunold	9000
104	Bruce	Ernst	6000

ORACLE

## What Is a View?

You can present logical subsets or combinations of data by creating views of tables. A view is a logical table based on a table or another view. A view contains no data of its own, but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called *base tables*. The view is stored as a SELECT statement in the data dictionary.

# Advantages of Views



ORACLE

11 - 4

Copyright © 2009, Oracle. All rights reserved.

## Advantages of Views

- Views restrict access to the data because it displays selected columns from the table.
- Views can be used to make simple queries to retrieve the results of complicated queries. For example, views can be used to query information from multiple tables without the user knowing how to write a join statement.
- Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- Views provide groups of users access to data according to their particular criteria.

For more information, see the section on “CREATE VIEW” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

# Simple Views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

ORACLE

11 - 5

Copyright © 2009, Oracle. All rights reserved.

## Simple Views and Complex Views

There are two classifications for views: simple and complex. The basic difference is related to the DML (INSERT, UPDATE, and DELETE) operations.

- A simple view is one that:
  - Derives data from only one table
  - Contains no functions or groups of data
  - Can perform DML operations through the view
- A complex view is one that:
  - Derives data from many tables
  - Contains functions or groups of data
  - Does not always allow DML operations through the view

# Creating a View

- You embed a subquery in the CREATE VIEW statement:

```
CREATE [OR REPLACE] [FORCE|NFORCE] VIEW view
  [(alias[, alias]...)]
  AS subquery
  [WITH CHECK OPTION [CONSTRAINT constraint]]
  [WITH READ ONLY [CONSTRAINT constraint]];
```

- The subquery can contain complex SELECT syntax.

ORACLE

11 - 6

Copyright © 2009, Oracle. All rights reserved.

## Creating a View

You can create a view by embedding a subquery in the CREATE VIEW statement.

In the syntax:

OR REPLACE	Re-creates the view if it already exists
FORCE	Creates the view regardless of whether or not the base tables exist
NOFORCE	Creates the view only if the base tables exist (This is the default.)
<i>view</i>	Is the name of the view
<i>alias</i>	Specifies names for the expressions selected by the view's query (The number of aliases must match the number of expressions selected by the view.)
<i>subquery</i>	Is a complete SELECT statement (You can use aliases for the columns in the SELECT list.)
WITH CHECK OPTION	Specifies that only those rows that are accessible to the view can be inserted or updated
<i>constraint</i>	Is the name assigned to the CHECK OPTION constraint
WITH READ ONLY	Ensures that no DML operations can be performed on this view

**Note:** In SQL Developer, click the Run Script icon or press [F5] to run the data definition language (DDL) statements. The feedback messages will be shown on the Script Output tabbed page.

# Creating a View

- Create the EMPVU80 view, which contains details of the employees in department 80:

```
CREATE VIEW empvu80
AS SELECT employee_id, last_name, salary
   FROM employees
 WHERE department_id = 80;
```

CREATE VIEW succeeded.

- Describe the structure of the view by using the SQL\*Plus DESCRIBE command:

```
DESCRIBE empvu80
```

ORACLE

11 - 7

Copyright © 2009, Oracle. All rights reserved.

## Creating a View (continued)

The example in the slide creates a view that contains the employee number, last name, and salary for each employee in department 80.

You can display the structure of the view by using the DESCRIBE command.

Name	Null	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
SALARY		NUMBER(8,2)

## Guidelines

- The subquery that defines a view can contain complex SELECT syntax, including joins, groups, and subqueries.
- If you do not specify a constraint name for the view created with the WITH CHECK OPTION, the system assigns a default name in the SYS\_Cn format.
- You can use the OR REPLACE option to change the definition of the view without dropping and re-creating it, or regranting the object privileges previously granted on it.

## Creating a View

- Create a view by using column aliases in the subquery:

```
CREATE VIEW salvu50
AS SELECT employee_id ID_NUMBER, last_name NAME,
           salary*12 ANN_SALARY
      FROM employees
     WHERE department_id = 50;
CREATE VIEW succeeded.
```

- Select the columns from this view by the given alias names.

ORACLE

11 - 8

Copyright © 2009, Oracle. All rights reserved.

### Creating a View (continued)

You can control the column names by including column aliases in the subquery.

The example in the slide creates a view containing the employee number (EMPLOYEE\_ID) with the alias ID\_NUMBER, name (LAST\_NAME) with the alias NAME, and annual salary (SALARY) with the alias ANN\_SALARY for every employee in department 50.

Alternatively, you can use an alias after the CREATE statement and before the SELECT subquery. The number of aliases listed must match the number of expressions selected in the subquery.

```
CREATE OR REPLACE VIEW salvu50 (ID_NUMBER, NAME, ANN_SALARY)
AS SELECT employee_id, last_name, salary*12
      FROM employees
     WHERE department_id = 50;
```

```
CREATE VIEW succeeded.
```

## Retrieving Data from a View

```
SELECT *  
FROM salvu50;
```

	ID_NUMBER	NAME	ANN_SALARY
1	124 Mourgos		69600
2	141 Rajs		42000
3	142 Davies		37200
4	143 Matos		31200
5	144 Vargas		30000

ORACLE

## Retrieving Data from a View

You can retrieve data from a view as you would from any table. You can display either the contents of the entire view or just specific rows and columns.

## Modifying a View

- Modify the EMPVU80 view by using a CREATE OR REPLACE VIEW clause. Add an alias for each column name:

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT employee_id, first_name || ' '
      || last_name, salary, department_id
    FROM employees
   WHERE department_id = 80;
CREATE OR REPLACE VIEW succeeded.
```

- Column aliases in the CREATE OR REPLACE VIEW clause are listed in the same order as the columns in the subquery.

ORACLE

11 - 10

Copyright © 2009, Oracle. All rights reserved.

### Modifying a View

With the OR REPLACE option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranting object privileges.

**Note:** When assigning column aliases in the CREATE OR REPLACE VIEW clause, remember that the aliases are listed in the same order as the columns in the subquery.

## Creating a Complex View

Create a complex view that contains group functions to display values from two tables:

```
CREATE OR REPLACE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT      d.department_name, MIN(e.salary) ,
                  MAX(e.salary), AVG(e.salary)
   FROM      employees e JOIN departments d
   ON          (e.department_id = d.department_id)
 GROUP BY    d.department_name;
CREATE OR REPLACE VIEW succeeded.
```

ORACLE

11 - 11

Copyright © 2009, Oracle. All rights reserved.

### Creating a Complex View

The example in the slide creates a complex view of department names, minimum salaries, maximum salaries, and the average salaries by department. Note that alternative names have been specified for the view. This is a requirement if any column of the view is derived from a function or an expression. You can view the structure of the view by using the DESCRIBE command. Display the contents of the view by issuing a SELECT statement.

```
SELECT *
  FROM      dept_sum_vu;
```

#	NAME	MINSAL	MAXSAL	AVGSAL
1	Administration	4400	4400	4400
2	Accounting	8300	12000	10150
3	IT	4200	9000	6400
4	Executive	17000	24000	19333.33333333...
5	Shipping	2500	5800	3500
6	Sales	8600	11000	10033.33333333...
7	Marketing	6000	13000	9500

## Rules for Performing DML Operations on a View

- You can usually perform DML operations on simple views. 
- You cannot remove a row if the view contains the following:
  - Group functions
  - A GROUP BY clause
  - The DISTINCT keyword
  - The pseudocolumn ROWNUM keyword

ORACLE

11 - 12

Copyright © 2009, Oracle. All rights reserved.

## Rules for Performing DML Operations on a View

- You can perform DML operations on data through a view if those operations follow certain rules.
- You can remove a row from a view unless it contains any of the following:
  - Group functions
  - A GROUP BY clause
  - The DISTINCT keyword
  - The pseudocolumn ROWNUM keyword

## Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

- Group functions
- A GROUP BY clause
- The DISTINCT keyword
- The pseudocolumn ROWNUM keyword
- Columns defined by expressions



### Rules for Performing DML Operations on a View (continued)

You can modify data through a view unless it contains any of the conditions mentioned in the previous slide or columns defined by expressions (for example, SALARY \* 12).

## Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

- Group functions
- A GROUP BY clause
- The DISTINCT keyword
- The pseudocolumn ROWNUM keyword
- Columns defined by expressions
- NOT NULL columns in the base tables that are not selected by the view



## Rules for Performing DML Operations on a View (continued)

You can add data through a view unless it contains any of the items listed in the slide. You cannot add data to a view if the view contains NOT NULL columns without default values in the base table. All the required values must be present in the view. Remember that you are adding values directly to the underlying table *through* the view.

For more information, see the section on “CREATE VIEW” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

## Using the WITH CHECK OPTION Clause

- You can ensure that DML operations performed on the view stay in the domain of the view by using the WITH CHECK OPTION clause:

```
CREATE OR REPLACE VIEW empvu20
AS SELECT      *
   FROM employees
  WHERE department_id = 20
    WITH CHECK OPTION CONSTRAINT empvu20_ck ;
CREATE OR REPLACE VIEW succeeded.
```

- Any attempt to INSERT a row with a department\_id other than 20, or to UPDATE the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint.

ORACLE

11 - 15

Copyright © 2009, Oracle. All rights reserved.

## Using the WITH CHECK OPTION Clause

It is possible to perform referential integrity checks through views. You can also enforce constraints at the database level. The view can be used to protect data integrity, but the use is very limited.

The WITH CHECK OPTION clause specifies that INSERTS and UPDATES performed through the view cannot create rows that the view cannot select. Therefore, it enables integrity constraints and data validation checks to be enforced on data being inserted or updated. If there is an attempt to perform DML operations on rows that the view has not selected, an error is displayed, along with the constraint name if that has been specified.

```
UPDATE empvu20
  SET department_id = 10
 WHERE employee_id = 201;
```

causes:

```
Error report:
SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
01402. 00000 - "view WITH CHECK OPTION where-clause violation"
```

**Note:** No rows are updated because, if the department number were to change to 10, the view would no longer be able to see that employee. With the WITH CHECK OPTION clause, therefore, the view can see only the employees in department 20 and does not allow the department number for those employees to be changed through the view.

## Denying DML Operations

- You can ensure that no DML operations occur by adding the WITH READ ONLY option to your view definition.
- Any attempt to perform a DML operation on any row in the view results in an Oracle server error.



ORACLE

11 - 16

Copyright © 2009, Oracle. All rights reserved.

### Denying DML Operations

You can ensure that no DML operations occur on your view by creating it with the WITH READ ONLY option. The example in the next slide modifies the EMPVU10 view to prevent any DML operations on the view.

# Denying DML Operations

```
CREATE OR REPLACE VIEW empvu10
  (employee_number, employee_name, job_title)
AS SELECT      employee_id, last_name, job_id
   FROM        employees
  WHERE        department_id = 10
    WITH READ ONLY ;
CREATE OR REPLACE VIEW succeeded.
```

ORACLE

11 - 17

Copyright © 2009, Oracle. All rights reserved.

## Denying DML Operations (continued)

Any attempt to remove a row from a view with a read-only constraint results in an error:

```
DELETE FROM empvu10
WHERE employee_number = 200;
```

Similarly, any attempt to insert a row or modify a row using the view with a read-only constraint results in the same error.

Error report:

```
SQL Error: ORA-42399: cannot perform a DML operation on a read-only view
```

## Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

```
DROP VIEW empvu80;
```

```
DROP VIEW empvu80 succeeded.
```

ORACLE

11 - 18

Copyright © 2009, Oracle. All rights reserved.

### Removing a View

You use the `DROP VIEW` statement to remove a view. The statement removes the view definition from the database. However, dropping views has no effect on the tables on which the view was based. Alternatively, views or other applications based on the deleted views become invalid. Only the creator or a user with the `DROP ANY VIEW` privilege can remove a view.

In the syntax, `view` is the name of the view.

# Sequences

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

11 - 19

Copyright © 2009, Oracle. All rights reserved.

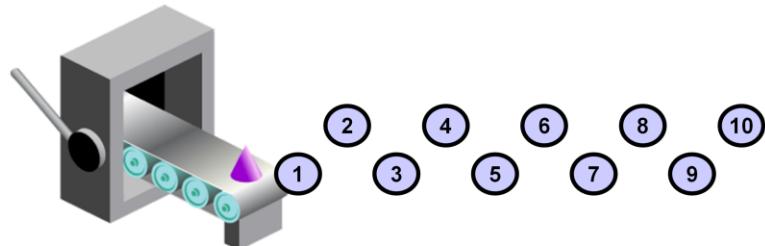
## Sequences

A sequence is a database object that creates integer values. You can create sequences and then use them to generate numbers.

# Sequences

A sequence:

- Can automatically generate unique numbers
- Is a shareable object
- Can be used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory



ORACLE

11 - 20

Copyright © 2009, Oracle. All rights reserved.

## Sequences (continued)

A sequence is a user-created database object that can be shared by multiple users to generate integers. You can define a sequence to generate unique values or to recycle and use the same numbers again. A typical usage for sequences is to create a primary key value, which must be unique for each row. A sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object because it can reduce the amount of application code needed to write a sequence-generating routine.

Sequence numbers are stored and generated independent of tables. Therefore, the same sequence can be used for multiple tables.

## CREATE SEQUENCE Statement: Syntax

Define a sequence to generate sequential numbers automatically:

```
CREATE SEQUENCE sequence
    [INCREMENT BY n]
    [START WITH n]
    [{MAXVALUE n | NOMAXVALUE}]
    [{MINVALUE n | NOMINVALUE}]
    [{CYCLE | NOCYCLE}]
    [{CACHE n | NOCACHE}];
```

ORACLE

11 - 21

Copyright © 2009, Oracle. All rights reserved.

## CREATE SEQUENCE Statement: Syntax

Automatically generate sequential numbers by using the CREATE SEQUENCE statement.

In the syntax:

<i>sequence</i>	Is the name of the sequence generator
INCREMENT BY <i>n</i>	Specifies the interval between sequence numbers, where <i>n</i> is an integer (If this clause is omitted, the sequence increments by 1.)
START WITH <i>n</i>	Specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
MAXVALUE <i>n</i>	Specifies the maximum value the sequence can generate
NOMAXVALUE	Specifies a maximum value of $10^{27}$ for an ascending sequence and $-1$ for a descending sequence (This is the default option.)
MINVALUE <i>n</i>	Specifies the minimum sequence value
NOMINVALUE	Specifies a minimum value of 1 for an ascending sequence and $-(10^{26})$ for a descending sequence (This is the default option.)

# Creating a Sequence

- Create a sequence named DEPT\_DEPTID\_SEQ to be used for the primary key of the DEPARTMENTS table.
- Do not use the CYCLE option.

```
CREATE SEQUENCE dept_deptid_seq
    INCREMENT BY 10
    START WITH 120
    MAXVALUE 9999
    NOCACHE
    NOCYCLE;
```

```
CREATE SEQUENCE succeeded.
```

ORACLE

11 - 22

Copyright © 2009, Oracle. All rights reserved.

## Creating a Sequence (continued)

CYCLE   NOCYCLE	Specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default option.)
CACHE n   NOCACHE	Specifies how many values the Oracle server preallocates and keeps in memory (By default, the Oracle server caches 20 values.)

The example in the slide creates a sequence named DEPT\_DEPTID\_SEQ to be used for the DEPARTMENT\_ID column of the DEPARTMENTS table. The sequence starts at 120, does not allow caching, and does not cycle.

Do not use the CYCLE option if the sequence is used to generate primary key values, unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

For more information, see the section on “CREATE SEQUENCE” in the *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

**Note:** The sequence is not tied to a table. Generally, you should name the sequence after its intended use. However, the sequence can be used anywhere, regardless of its name.

## **NEXTVAL and CURRVAL Pseudocolumns**

- NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.
- CURRVAL obtains the current sequence value.
- NEXTVAL must be issued for that sequence before CURRVAL contains a value.

**ORACLE**

11 - 23

Copyright © 2009, Oracle. All rights reserved.

### **NEXTVAL and CURRVAL Pseudocolumns**

After you create your sequence, it generates sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference *sequence*.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.

The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. However, NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When you reference *sequence*.CURRVAL, the last value returned to that user's process is displayed.

# Using a Sequence

- Insert a new department named “Support” in location ID 2500:

```
INSERT INTO departments(department_id,
                        department_name, location_id)
VALUES      (dept_deptid_seq.NEXTVAL,
              'Support', 2500);
```

1 rows inserted

- View the current value for the DEPT\_DEPTID\_SEQ sequence:

```
SELECT    dept_deptid_seq.CURRVAL
FROM      dual;
```

ORACLE

11 - 24

Copyright © 2009, Oracle. All rights reserved.

## Using a Sequence

The example in the slide inserts a new department in the DEPARTMENTS table. It uses the DEPT\_DEPTID\_SEQ sequence to generate a new department number as follows.

You can view the current value of the sequence using the *sequence\_name*.CURRVAL, as shown in the second example in the slide. The output of the query is shown below:

AZ	CURRVAL
1	120

Suppose that you now want to hire employees to staff the new department. The INSERT statement to be executed for all new employees can include the following code:

```
INSERT INTO employees (employee_id, department_id, ...)
VALUES (employees_seq.NEXTVAL, dept_deptid_seq.CURRVAL, ...);
```

**Note:** The preceding example assumes that a sequence called EMPLOYEE\_SEQ has already been created to generate new employee numbers.

## Caching Sequence Values

- Caching sequence values in memory gives faster access to those values.
- Gaps in sequence values can occur when:
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table

ORACLE

11 - 25

Copyright © 2009, Oracle. All rights reserved.

### Caching Sequence Values

You can cache sequences in memory to provide faster access to those sequence values. The cache is populated the first time you refer to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence value is used, the next request for the sequence pulls another cache of sequences into memory.

### Gaps in the Sequence

Although sequence generators issue sequential numbers without gaps, this action occurs independent of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost.

Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in memory, those values are lost if the system crashes.

Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. However, if you do so, each table can contain gaps in the sequential numbers.

# Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option:

```
ALTER SEQUENCE dept_deptid_seq
    INCREMENT BY 20
    MAXVALUE 999999
    NOCACHE
    NOCYCLE;
```

```
ALTER SEQUENCE dept_deptid_seq succeeded.
```

ORACLE

11 - 26

Copyright © 2009, Oracle. All rights reserved.

## Modifying a Sequence

If you reach the MAXVALUE limit for your sequence, no additional values from the sequence are allocated and you will receive an error indicating that the sequence exceeds the MAXVALUE. To continue to use the sequence, you can modify it by using the ALTER SEQUENCE statement.

### Syntax

```
ALTER SEQUENCE sequence
    [INCREMENT BY n]
    [ {MAXVALUE n | NOMAXVALUE} ]
    [ {MINVALUE n | NOMINVALUE} ]
    [ {CYCLE | NOCYCLE} ]
    [ {CACHE n | NOCACHE} ];
```

In the syntax, *sequence* is the name of the sequence generator.

For more information, see the section on “ALTER SEQUENCE” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

## Guidelines for Modifying a Sequence

- You must be the owner or have the ALTER privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.
- To remove a sequence, use the DROP statement:

```
DROP SEQUENCE dept_deptid_seq;
```

```
DROP SEQUENCE dept_deptid_seq succeeded.
```

ORACLE

11 - 27

Copyright © 2009, Oracle. All rights reserved.

## Guidelines for Modifying a Sequence

- You must be the owner or have the ALTER privilege for the sequence to modify it. You must be the owner or have the DROP ANY SEQUENCE privilege to remove it.
- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- The START WITH option cannot be changed using ALTER SEQUENCE. The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed. For example, a new MAXVALUE that is less than the current sequence number cannot be imposed.

```
ALTER SEQUENCE dept_deptid_seq
    INCREMENT BY 20
    MAXVALUE 90
    NOCACHE
    NOCYCLE;
```

- The error:

Error report:

```
SQL Error: ORA-04009: MAXVALUE cannot be made to be less than the current value
04009. 00000 -  "MAXVALUE cannot be made to be less than the current value"
*Cause:    the current value exceeds the given MAXVALUE
*Action:   make sure that the new MAXVALUE is larger than the current value
```

# Indexes

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

11 - 28

Copyright © 2009, Oracle. All rights reserved.

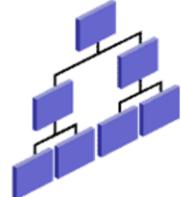
## Indexes

Indexes are database objects that you can create to improve the performance of some queries. Indexes can also be created automatically by the server when you create a primary key or a unique constraint.

# Indexes

An index:

- Is a schema object
- Can be used by the Oracle server to speed up the retrieval of rows by using a pointer
- Can reduce disk input/output (I/O) by using a rapid path access method to locate data quickly
- Is independent of the table that it indexes
- Is used and maintained automatically by the Oracle server



ORACLE

11 - 29

Copyright © 2009, Oracle. All rights reserved.

## Indexes (continued)

An Oracle server index is a schema object that can speed up the retrieval of rows by using a pointer. Indexes can be created explicitly or automatically. If you do not have an index on the column, a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the disk I/O by using an indexed path to locate data quickly. An index is used and maintained automatically by the Oracle server. After an index is created, no direct activity is required by the user.

Indexes are logically and physically independent of the table that they index. This means that they can be created or dropped at any time, and have no effect on the base tables or other indexes.

**Note:** When you drop a table, the corresponding indexes are also dropped.

For more information, see the section on “Schema Objects: Indexes” in *Oracle Database Concepts 11g, Release 1 (11.1)*.

## How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a PRIMARY KEY or UNIQUE constraint in a table definition.



- Manually: Users can create nonunique indexes on columns to speed up access to the rows.



ORACLE

11 - 30

Copyright © 2009, Oracle. All rights reserved.

### How Are Indexes Created?

You can create two types of indexes.

- **Unique index:** The Oracle server automatically creates this index when you define a column in a table to have a PRIMARY KEY or a UNIQUE constraint. The name of the index is the name that is given to the constraint.
- **Nonunique index:** This is an index that a user can create. For example, you can create the FOREIGN KEY column index for a join in a query to improve the speed of retrieval.

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

## Creating an Index

- Create an index on one or more columns:

```
CREATE [UNIQUE] [BITMAP] INDEX index
ON table (column[, column]...);
```

- Improve the speed of query access to the LAST\_NAME column in the EMPLOYEES table:

```
CREATE INDEX emp_last_name_idx
ON employees(last_name);
CREATE INDEX succeeded.
```

ORACLE

11 - 31

Copyright © 2009, Oracle. All rights reserved.

## Creating an Index

Create an index on one or more columns by issuing the CREATE INDEX statement.

In the syntax:

- *index* Is the name of the index
- *table* Is the name of the table
- *column* Is the name of the column in the table to be indexed

Specify UNIQUE to indicate that the value of the column (or columns) upon which the index is based must be unique. Specify BITMAP to indicate that the index is to be created with a bitmap for each distinct key, rather than indexing each row separately. Bitmap indexes store the rowids associated with a key value as a bitmap.

For more information, see the section on “CREATE INDEX” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

# Index Creation Guidelines

## Create an index when:

- A column contains a wide range of values
- A column contains a large number of null values
- One or more columns are frequently used together in a WHERE clause or a join condition
- The table is large and most queries are expected to retrieve less than 2% to 4% of the rows in the table

## Do not create an index when:

- The columns are not often used as a condition in the query
- The table is small or most queries are expected to retrieve more than 2% to 4% of the rows in the table
- The table is updated frequently
- The indexed columns are referenced as part of an expression

ORACLE

11 - 32

Copyright © 2009, Oracle. All rights reserved.

## Index Creation Guidelines

### More Is Not Always Better

Having more indexes on a table does not produce faster queries. Each DML operation that is committed on a table with indexes means that the indexes must be updated. The more indexes that you have associated with a table, the more effort the Oracle server must make to update all the indexes after a DML operation.

### When to Create an Index

Therefore, you should create indexes only if:

- The column contains a wide range of values
- The column contains a large number of null values
- One or more columns are frequently used together in a WHERE clause or join condition
- The table is large and most queries are expected to retrieve less than 2% to 4% of the rows

Remember that if you want to enforce uniqueness, you should define a unique constraint in the table definition. A unique index is then created automatically.

# Removing an Index

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

```
DROP INDEX index;
```

- Remove the `emp_last_name_idx` index from the data dictionary:

```
DROP INDEX emp_last_name_idx;  
DROP INDEX emp_last_name_idx succeeded.
```

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

ORACLE

11 - 33

Copyright © 2009, 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.

# Synonyms

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

11 - 34

Copyright © 2009, Oracle. All rights reserved.

## Synonyms

Synonyms are database objects that enable you to call a table by another name. You can create synonyms to give an alternative name to a table.

# Creating a Synonym for an Object

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- Create an easier reference to a table that is owned by another user
- Shorten lengthy object names

```
CREATE [PUBLIC] SYNONYM synonym
FOR object;
```

ORACLE

11 - 35

Copyright © 2009, Oracle. All rights reserved.

## Creating a Synonym for an Object

To refer to a table that is owned by another user, you need to prefix the table name with the name of the user who created it, followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

In the syntax:

PUBLIC	Creates a synonym that is accessible to all users
<i>synonym</i>	Is the name of the synonym to be created
<i>object</i>	Identifies the object for which the synonym is created

### Guidelines

- The object cannot be contained in a package.
- A private synonym name must be distinct from all other objects that are owned by the same user.

For more information, see the section on “CREATE SYNONYM” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

# Creating and Removing Synonyms

- Create a shortened name for the DEPT\_SUM\_VU view:

```
CREATE SYNONYM d_sum  
FOR dept_sum_vu;  
CREATE SYNONYM succeeded.
```

- Drop a synonym:

```
DROP SYNONYM d_sum;  
DROP SYNONYM d_sum succeeded.
```

ORACLE

11 - 36

Copyright © 2009, Oracle. All rights reserved.

## Creating and Removing Synonyms

### Creating a Synonym

The slide example creates a synonym for the DEPT\_SUM\_VU view for quicker reference.

The database administrator can create a public synonym that is accessible to all users. The following example creates a public synonym named DEPT for Alice's DEPARTMENTS table:

```
CREATE PUBLIC SYNONYM dept  
CREATE SYNONYM succeeded.
```

### Removing a Synonym

To remove a synonym, use the DROP SYNONYM statement. Only the database administrator can drop a public synonym.

```
DROP PUBLIC SYNONYM dept;
```

For more information, see the section on “DROP SYNONYM” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

---

**Oracle Database 11g: SQL  
Fundamentals II**

Electronic Presentation

.....

D49996GC20  
Edition 2.0  
October 2009

**ORACLE®**



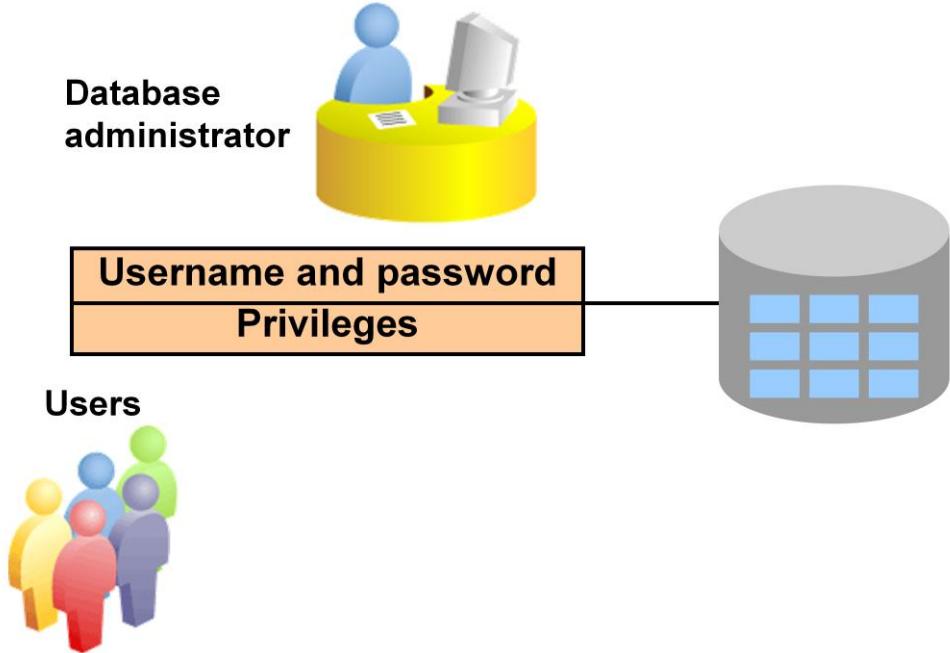
# 1

## Controlling User Access

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

# Controlling User Access



## 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.

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 perform on the objects.

# Privileges

- Database security:
  - System security
  - Data security
- System privileges: Performing a particular action within the database
- Object privileges: Manipulating the content of the database objects
- Schemas: Collection of objects such as tables, views, and sequences

ORACLE®

1 - 3

Copyright © 2009, Oracle. All rights reserved.

## Privileges

A privilege is 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.

A system privilege is the right to perform a particular action, or to perform an action on any schema objects of a particular type. An object privilege provides the user the ability to perform a particular action on a specific schema object.

For more information, see the *Oracle Database 2 Day DBA 11g Release 2 (11.2)* 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 - 4

Copyright © 2009, Oracle. All rights reserved.

## System Privileges

More than 100 distinct system privileges are available for users and roles. Typically, system privileges are provided by the database administrator (DBA).

### 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 demo
IDENTIFIED BY demo;
```

ORACLE

## Creating Users

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 the *Oracle Database 11g SQL Reference*.

**Note:** Starting with Oracle Database 11g, passwords are case-sensitive.

# 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 - 6

Copyright © 2009, 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  
(which designates that every user is granted the privilege)

# Granting System Privileges

The DBA can grant specific system privileges to a user.

```
GRANT  create session, create table,  
       create sequence, create view  
TO     demo;
```

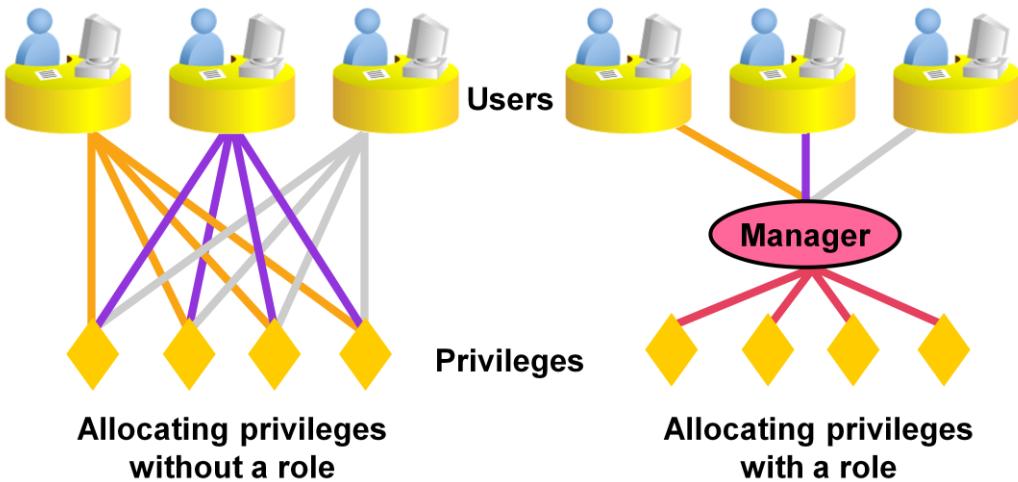


## Granting System Privileges

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 in the slide, the `demo` user has been assigned the privileges to create sessions, tables, sequences, and views.

# What Is a Role?



ORACLE

1 - 8

Copyright © 2009, 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. A role is not a schema object, therefore any user can add privileges to a role.

# Creating and Granting Privileges to a Role

- Create a role:

```
CREATE ROLE manager;
```

- Grant privileges to a role:

```
GRANT create table, create view  
TO manager;
```

- Grant a role to users:

```
GRANT manager TO alice;
```

ORACLE

## Creating a Role

The example in the slide creates a `manager` role and then enables the manager to create tables and views. It then grants user `alice` the role of a manager. Now `alice` can create tables and views.

If users have multiple roles granted to them, they receive all the privileges associated with all 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 demo  
IDENTIFIED BY employ;
```

ORACLE

1 - 10

Copyright © 2009, 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.

The slide example shows that the `demo` user changes the password by using the ALTER USER statement.

### Syntax

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

In the syntax:

<code>user</code>	Is the name of the user
<code>password</code>	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 11g 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 SQL Developer.

# Object Privileges

Object privilege	Table	View	Sequence
ALTER	✓		✓
DELETE	✓	✓	
INDEX	✓		
INSERT	✓	✓	
REFERENCES	✓		
SELECT	✓	✓	✓
UPDATE	✓	✓	

ORACLE

## 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 in 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.

**Note:** With the REFERENCES privilege, you can ensure that other users can create FOREIGN KEY constraints that reference your table.

# 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 - 12

Copyright © 2009, 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, 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   demo;
```

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

```
GRANT update (department_name, location_id)  
ON   departments  
TO   demo, manager;
```

ORACLE

1 - 13

Copyright © 2009, 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 in the slide grants the demo user the privilege to query your EMPLOYEES table. The second example grants UPDATE privileges on specific columns in the DEPARTMENTS table to demo and to the manager role.

For example, if your schema is oraxx, and the demo user now wants to use a SELECT statement to obtain data from your EMPLOYEES table, the syntax he or she must use is:

```
SELECT * FROM oraxx.employees;
```

Alternatively, the demo user can create a synonym for the table and issue a SELECT statement from the synonym:

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

# Passing On Your Privileges

- Give a user authority to pass along privileges:

```
GRANT select, insert  
ON departments  
TO demo  
WITH GRANT OPTION;
```

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

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

ORACLE

1 - 14

Copyright © 2009, Oracle. All rights reserved.

## Passing On Your Privileges

### **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 in the slide gives the demo user access to your DEPARTMENTS table with the privileges to query the table and add rows to the table. The example also shows that user1 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 Granted Privileges

Data Dictionary View	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_SYS_PRIVS	System privileges granted to the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects
USER_TAB_PRIVS_REC'D	Object privileges granted to the user
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects
USER_COL_PRIVS_REC'D	Object privileges granted to the user on specific columns

ORACLE

1 - 15

Copyright © 2009, 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,” 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

The data dictionary is organized in tables and views and contains information about the database. You can access the data dictionary to view the privileges that you have. The table in the slide describes various data dictionary views.

You learn more about data dictionary views in the lesson titled “Managing Objects with 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 - 16

Copyright © 2009, 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 the *Oracle Database 11g SQL Reference*.

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

# Revoking Object Privileges

Revoke the SELECT and INSERT privileges given to the demo user on the DEPARTMENTS table.

```
REVOKE select, insert  
ON departments  
FROM demo;
```

ORACLE

1 - 17

Copyright © 2009, Oracle. All rights reserved.

## Revoking Object Privileges (continued)

The example in the slide revokes SELECT and INSERT privileges given to the demo user 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, the revoking cascades to all the 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, the privileges granted to users C and D are also revoked.

# 2

## Managing Schema Objects

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

## **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 - 2

Copyright © 2009, Oracle. All rights reserved.

### **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.

## 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 [, column] ...);
```

ORACLE

2 - 3

Copyright © 2009, Oracle. All rights reserved.

### 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 column
<i>datatype</i>	Is the data type and length of the column
DEFAULT <i>expr</i>	Specifies the default value for a column

# Adding a Column

- You use the ADD clause to add columns:

```
ALTER TABLE dept80
ADD          (job_id VARCHAR2(9)) ;
```

```
ALTER TABLE dept80 succeeded.
```

- The new column becomes the last column:

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
1	145	Russell	14000	01-OCT-96	(null)
2	146	Partners	13500	05-JAN-97	(null)
3	147	Errazuriz	12000	10-MAR-97	(null)
4	148	Cambrault	11000	15-OCT-99	(null)
5	149	Zlotkey	10500	29-JAN-00	(null)

ORACLE

## 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 in 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, the new column is initially null or takes the default value for all the rows. You can add a mandatory NOT NULL column to a table that contains data in the other columns only if you specify a default value. You can add a NOT NULL column to an empty table without the default value.

# Modifying a Column

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

```
ALTER TABLE dept80
MODIFY      (last_name VARCHAR2(30));
```

```
ALTER TABLE dept80 succeeded.
```

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

ORACLE

2 - 5

Copyright © 2009, 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 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 that you no longer need from the table:

```
ALTER TABLE dept80
DROP COLUMN job_id;
```

```
ALTER TABLE dept80 succeeded.
```

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
1	145	Russell	14000	01-OCT-96
2	146	Partners	13500	05-JAN-97
3	147	Errazuriz	12000	10-MAR-97
4	148	Cambrault	11000	15-OCT-99
5	149	Zlotkey	10500	29-JAN-00

ORACLE

## 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 DROP COLUMN` 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 there are fewer users on the system 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.

For more information about index-organized tables and partitioned table, refer to *Oracle Database Concepts* and *Oracle Database Administrator's Guide*.

## 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> [ , <column_name>]) ;
OR
ALTER TABLE <table_name>
SET UNUSED COLUMN <column_name> [,<column_name>];
```

```
ALTER TABLE <table_name>
DROP UNUSED COLUMNS;
```

ORACLE

2 - 7

Copyright © 2009, Oracle. All rights reserved.

### 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. The 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 for dropping a column.

# 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 - 9

Copyright © 2009, 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 generates 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;
```

```
ALTER TABLE emp2 succeeded.
```

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

```
ALTER TABLE succeeded.
```

ORACLE

2 - 10

Copyright © 2009, Oracle. All rights reserved.

### Adding a Constraint (continued)

The first example in 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 in 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 Clause

- Use the ON DELETE CASCADE clause to delete child rows when a parent key is deleted:

```
ALTER TABLE emp2 ADD CONSTRAINT emp_dt_fk  
FOREIGN KEY (Department_id)  
REFERENCES departments(department_id) ON DELETE CASCADE;
```

ALTER TABLE Emp2 succeeded.

- Use the ON DELETE SET NULL clause to set the child rows value to null when a parent key is deleted:

```
ALTER TABLE emp2 ADD CONSTRAINT emp_dt_fk  
FOREIGN KEY (Department_id)  
REFERENCES departments(department_id) ON DELETE SET NULL;
```

ALTER TABLE Emp2 succeeded.

ORACLE

2 - 11

Copyright © 2009, Oracle. All rights reserved.

### ON DELETE

By using the ON DELETE clause you can determine how Oracle Database handles referential integrity if you remove a referenced primary or unique key value.

#### 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 the 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.

#### ON DELETE SET NULL

When data in the parent key is deleted, the ON DELETE SET NULL action causes all the rows in the child table that depend on the deleted parent key value to be converted to null.

If you omit this clause, Oracle does not allow you to delete referenced key values in the parent table that have dependent rows in the child table.

# Dropping a Constraint

- Remove the manager constraint from the EMP2 table:

```
ALTER TABLE emp2
DROP CONSTRAINT emp_mgr_fk;
```

```
ALTER TABLE Emp2 succeeded.
```

- 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;
```

```
ALTER TABLE dept2 succeeded.
```

ORACLE

## 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;
```

```
ALTER TABLE Emp2 succeeded.
```

ORACLE

2 - 13

Copyright © 2009, 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;
```

```
ALTER TABLE Emp2 succeeded.
```

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

ORACLE

2 - 14

Copyright © 2009, 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 a PRIMARY KEY constraint, a UNIQUE or PRIMARY KEY index is created automatically. If an index already exists, it can be used by these keys.
- You can use the ENABLE clause in both the CREATE TABLE statement and the ALTER TABLE statement.

# 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 - 16

Copyright © 2009, Oracle. All rights reserved.

## Cascading Constraints

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

```
CREATE TABLE test1 (
    col1_pk NUMBER PRIMARY KEY,
    col2_fk NUMBER,
    col1 NUMBER,
    col2 NUMBER,
    CONSTRAINT fk_constraint FOREIGN KEY (col2_fk) REFERENCES
        test1,
    CONSTRAINT ck1 CHECK (col1_pk > 0 and col1 > 0),
    CONSTRAINT ck2 CHECK (col2_fk > 0));
```

An error is returned for the following statements:

```
ALTER TABLE test1 DROP (col1_pk); —col1_pk is a parent key.
ALTER TABLE test1 DROP (col1); —col1 is referenced by the multicolumn
                                constraint, ck1.
```

# Cascading Constraints

Example:

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

```
ALTER TABLE Emp2 succeeded.
```

```
ALTER TABLE test1
DROP (col1_pk, col2_fk, col1) CASCADE CONSTRAINTS;
```

```
ALTER TABLE test1 succeeded.
```

ORACLE

2 - 17

Copyright © 2009, Oracle. All rights reserved.

## Cascading Constraints (continued)

Submitting the following statement drops the EMPLOYEE\_ID column, 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, CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to the COL1\_PK column, it is valid to submit the following statement without the CASCADE CONSTRAINTS clause for the TEST1 table created on the previous page:

```
ALTER TABLE test1 DROP (col1_pk, col2_fk, col1);
```

# Renaming Table Columns and Constraints

Use the RENAME COLUMN clause of the ALTER TABLE statement to rename table columns.

```
ALTER TABLE marketing RENAME COLUMN team_id  
TO id;
```

```
ALTER TABLE marketing succeeded.
```

a

Use the RENAME CONSTRAINT clause of the ALTER TABLE statement to rename any existing constraint for a table.

```
ALTER TABLE marketing RENAME CONSTRAINT mktg_pk  
TO new_mktg_pk;
```

```
ALTER TABLE marketing succeeded.
```

b

ORACLE

2 - 18

Copyright © 2009, Oracle. All rights reserved.

## Renaming Table Columns and Constraints

When you rename a table column, the new name must not conflict with the name of any existing column in the table. You cannot use any other clauses in conjunction with the RENAME COLUMN clause.

The slide examples use the marketing table with the PRIMARY KEY mktg\_pk defined on the id column.

```
CREATE TABLE marketing (team_id NUMBER(10),  
                      target VARCHAR2(50),  
                      CONSTRAINT mktg_pk PRIMARY KEY(team_id));
```

```
CREATE TABLE succeeded.
```

Example **a** shows that the id column of the marketing table is renamed mktg\_id. Example **b** shows that mktg\_pk is renamed new\_mktg\_pk.

When you rename any existing constraint for a table, the new name must not conflict with any of your existing constraint names. You can use the RENAME CONSTRAINT clause to rename system-generated constraint names.

# Overview of Indexes

Indexes are created:

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

ORACLE

2 - 19

Copyright © 2009, 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 11g 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 the 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));
```

```
CREATE TABLE succeeded.
```

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

INDEX_NAME	TABLE_NAME
EMP_ID_IDX	NEW_EMP

ORACLE

2 - 20

Copyright © 2009, Oracle. All rights reserved.

## CREATE INDEX with the CREATE TABLE Statement

In the example in 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 constraint.

You can query the USER\_INDEXES data dictionary view for information about your indexes.

**Note:** You learn more about USER\_INDEXES in the lesson titled “Managing Objects with Data Dictionary Views.”

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));
```

```
CREATE TABLE succeeded.
```

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

## 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)) ;
```

CREATE INDEX succeeded.

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

ORACLE

2 - 22

Copyright © 2009, Oracle. All rights reserved.

## Function-Based Indexes

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

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

This 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.

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;
```

```
DROP INDEX upper_dept_name_idx succeeded.
```

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

ORACLE

## 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, then indexes, constraints, and triggers are automatically dropped, but views and sequences remain.

## **DROP TABLE ... PURGE**

```
DROP TABLE dept80 PURGE ;
```

```
DROP TABLE dept80 succeeded.
```

**ORACLE**

### **DROP TABLE ... PURGE**

Oracle Database provides a 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, include the `PURGE` clause as shown in the statement in 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`, 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.

## FLASHBACK TABLE Statement

- Enables you to recover tables to a specified point in time with a single statement
- Restores table data along with associated indexes and constraints
- Enables you to revert the table and its contents to a certain point in time or system change number (SCN)



ORACLE

2 - 25

Copyright © 2009, Oracle. All rights reserved.

### FLASHBACK TABLE Statement

Oracle Flashback Table enables you to recover tables to a specified point in time with a single statement. You can restore table data along with associated indexes and constraints while the database is online, undoing changes to only the specified tables.

The Flashback Table feature is similar to a self-service repair tool. For example, if a user accidentally deletes important rows from a table and then wants to recover the deleted rows, you can use the FLASHBACK TABLE statement to restore the table to the time before the deletion and see the missing rows in the table.

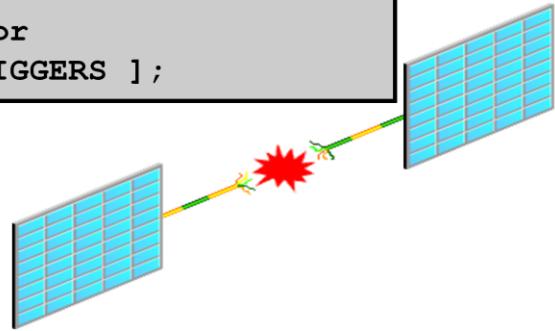
When using the FLASHBACK TABLE statement, you can revert the table and its contents to a certain time or to an SCN.

**Note:** The SCN is an integer value associated with each change to the database. It is a unique incremental number in the database. Every time you commit a transaction, a new SCN is recorded.

## FLASHBACK TABLE Statement

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

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



ORACLE

2 - 26

Copyright © 2009, Oracle. All rights reserved.

### FLASHBACK TABLE Statement (continued)

#### Self-Service Repair Facility

Oracle Database provides a SQL data definition language (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 time stamp. By default, database triggers are disabled during the flashback operation 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 Guide 11g Release 2 (11.2)*.

# Using the FLASHBACK TABLE Statement

```
DROP TABLE emp2;
```

```
DROP TABLE emp2 succeeded.
```

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

ORIGINAL_NAME	OPERATION	DROPTIME
EMP2	DROP	2009-05-20:18:00:39

...

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

```
FLASHBACK TABLE succeeded.
```

ORACLE

## Using the FLASHBACK TABLE Statement

### Syntax and Examples

The example restores the EMP2 table to a state before 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 until 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 or her objects in the recycle bin by using the following statement:

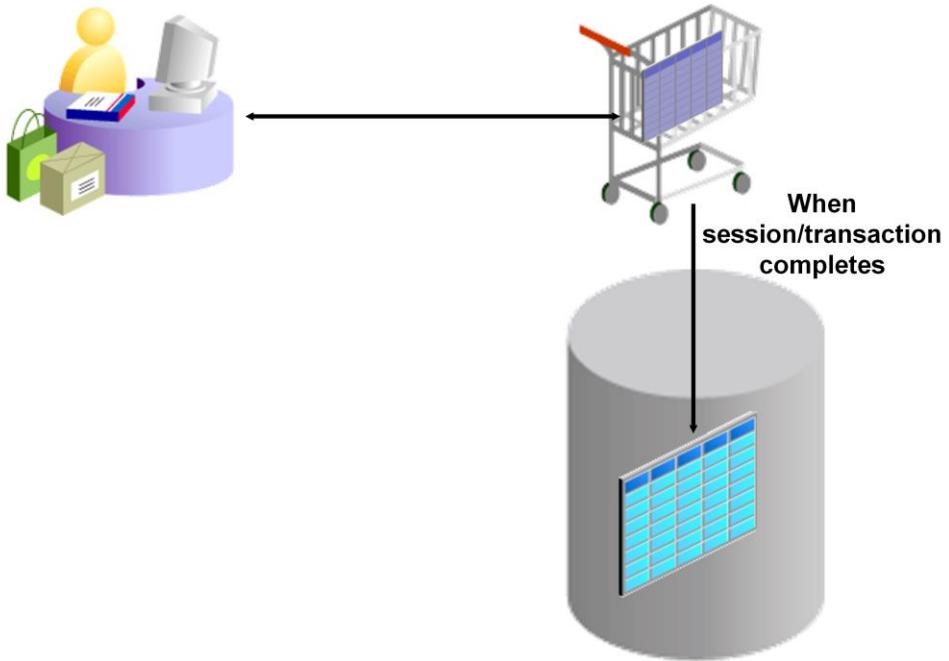
```
SELECT * FROM RECYCLEBIN;
```

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;
```

# Temporary Tables



ORACLE

## Temporary Tables

A temporary table is a table that holds data that exists only for the duration of a transaction or session. Data in a temporary table is private to the session, which means that each session can only see and modify its own data.

Temporary tables are useful in applications where a result set must be buffered. For example a shopping cart in an online application can be a temporary table. Each item is represented by a row in the temporary table. While you are shopping in an online store, you can keep on adding or removing items from your cart. During the session, this cart data is private. After you finalize your shopping and make the payments, the application moves the row for the chosen cart to a permanent table. At the end of the session, the data in the temporary data is automatically dropped.

Because temporary tables are statically defined, you can create indexes for them. Indexes created on temporary tables are also temporary. The data in the index has the same session or transaction scope as the data in the temporary table. You can also create a view or trigger on a temporary table.

# Creating a Temporary Table

```
CREATE GLOBAL TEMPORARY TABLE cart  
ON COMMIT DELETE ROWS;
```

1

```
CREATE GLOBAL TEMPORARY TABLE today_sales  
ON COMMIT PRESERVE ROWS AS  
SELECT * FROM orders  
WHERE order_date = SYSDATE;
```

2

ORACLE

2 - 29

Copyright © 2009, Oracle. All rights reserved.

## Creating a Temporary Table

To create a temporary table you can use the following command:

```
CREATE GLOBAL TEMPORARY TABLE tablename  
ON COMMIT [PRESERVE | DELETE] ROWS
```

By associating one of the following settings with the ON COMMIT clause, you can decide whether the data in the temporary table is transaction-specific (default) or session specific.

1. **DELETE ROWS:** As shown in example 1 in the slide, the DELETE ROWS setting creates a temporary table that is transaction specific. A session becomes bound to the temporary table with a transaction's first insert into the table. The binding goes away at the end of the transaction. The database truncates the table (delete all rows) after each commit.
2. **PRESERVE ROWS:** As shown in example 2 in the slide, the PRESERVE ROWS setting creates a temporary table that is session specific. Each sales representative session can store its own sales data for the day in the table. When a salesperson performs first insert on the today\_sales table, his or her session gets bound to the today\_sales table. This binding goes away at the end of the session or by issuing a TRUNCATE of the table in the session. The database truncates the table when you terminate the session.

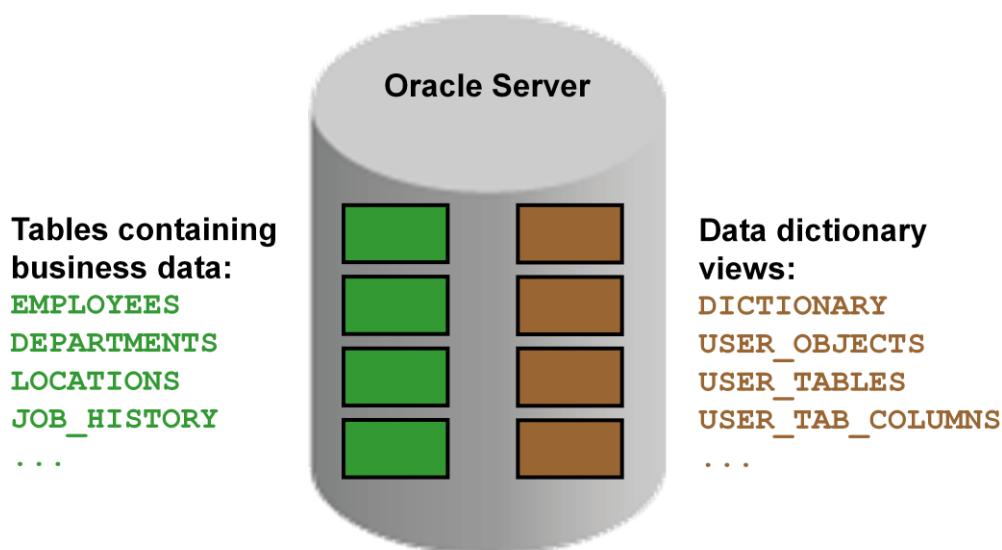


## **Managing Objects with Data Dictionary Views**

**ORACLE®**

Copyright © 2009, Oracle. All rights reserved.

# Data Dictionary



## Data Dictionary

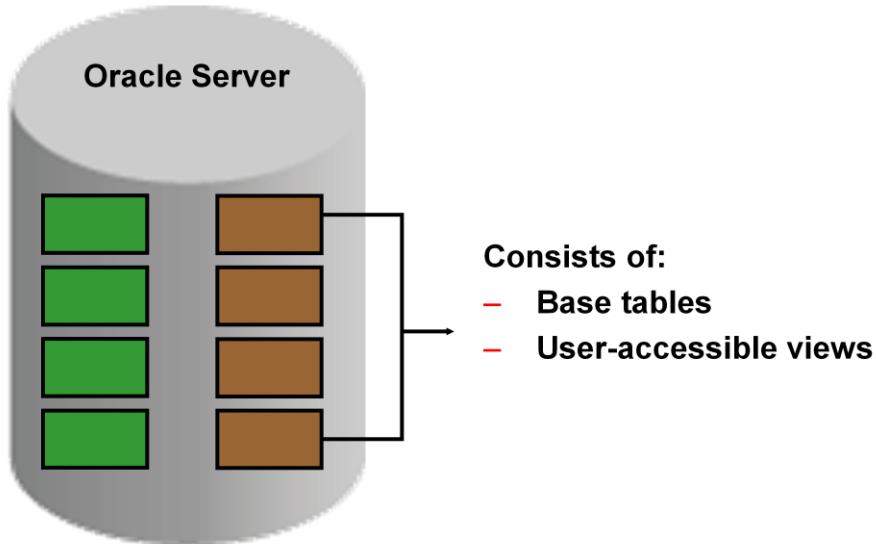
User tables are tables created by the user and contain business data, such as EMPLOYEES. There is another collection of tables and views in the Oracle database known as the *data dictionary*. This collection is created and maintained by the Oracle Server and contains information about the database. The data dictionary is structured in tables and views, just like other database data. Not only is the data dictionary central to every Oracle database, but it is also an important tool for all users, from end users to application designers and database administrators.

You use SQL statements to access the data dictionary. Because the data dictionary is read-only, you can issue only queries against its tables and views.

You can query the dictionary views that are based on the dictionary tables to find information such as:

- Definitions of all schema objects in the database (tables, views, indexes, synonyms, sequences, procedures, functions, packages, triggers, and so on)
- Default values for columns
- Integrity constraint information
- Names of Oracle users
- Privileges and roles that each user has been granted
- Other general database information

# Data Dictionary Structure



ORACLE®

## Data Dictionary Structure

Underlying base tables store information about the associated database. Only the Oracle Server should write to and read from these tables. You rarely access them directly.

There are several views that summarize and display the information stored in the base tables of the data dictionary. These views decode the base table data into useful information (such as user or table names) using joins and WHERE clauses to simplify the information. Most users are given access to the views rather than the base tables.

The Oracle user SYS owns all base tables and user-accessible views of the data dictionary. No Oracle user should *ever* alter (UPDATE, DELETE, or INSERT) any rows or schema objects contained in the SYS schema because such activity can compromise data integrity.

# Data Dictionary Structure

View naming convention:

View Prefix	Purpose
USER	User's view (what is in your schema; what you own)
ALL	Expanded user's view (what you can access)
DBA	Database administrator's view (what is in everyone's schemas)
V\$	Performance-related data

ORACLE

3 - 4

Copyright © 2009, Oracle. All rights reserved.

## Data Dictionary Structure (continued)

The data dictionary consists of sets of views. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes. For example, there is a view named `USER_OBJECTS`, another named `ALL_OBJECTS`, and a third named `DBA_OBJECTS`.

These three views contain similar information about objects in the database, except that the scope is different. `USER_OBJECTS` contains information about objects that you own or created. `ALL_OBJECTS` contains information about all objects to which you have access.

`DBA_OBJECTS` contains information about all objects that are owned by all users. For views that are prefixed with `ALL` or `DBA`, there is usually an additional column in the view named `OWNER` to identify who owns the object.

There is also a set of views that is prefixed with `v$`. These views are dynamic in nature and hold information about performance. Dynamic performance tables are not true tables, and they should not be accessed by most users. However, database administrators can query and create views on the tables and grant access to those views to other users. This course does not go into details about these views.

# How to Use the Dictionary Views

Start with DICTONARY. It contains the names and descriptions of the dictionary tables and views.

## DESCRIBE DICTIONARY

Name	Null	Type
TABLE_NAME		VARCHAR2(30)
COMMENTS		VARCHAR2(4000)

2 rows selected

```
SELECT *
FROM   dictionary
WHERE  table_name = 'USER_OBJECTS';
```

TABLE_NAME	COMMENTS
USER_OBJECTS	Objects owned by the user

ORACLE

## How to Use the Dictionary Views

To familiarize yourself with the dictionary views, you can use the dictionary view named DICTONARY. It contains the name and short description of each dictionary view to which you have access.

You can write queries to search for information about a particular view name, or you can search the COMMENTS column for a word or phrase. In the example shown, the DICTONARY view is described. It has two columns. The SELECT statement retrieves information about the dictionary view named USER\_OBJECTS. The USER\_OBJECTS view contains information about all the objects that you own.

You can write queries to search the COMMENTS column for a word or phrase. For example, the following query returns the names of all views that you are permitted to access in which the COMMENTS column contains the word *columns*:

```
SELECT table_name
FROM   dictionary
WHERE  LOWER(comments) LIKE '%columns%';
```

**Note:** The names in the data dictionary are in uppercase.

## **USER\_OBJECTS and ALL\_OBJECTS Views**

### **USER\_OBJECTS:**

- **Query USER\_OBJECTS to see all the objects that you own.**
- **Using USER\_OBJECTS, you can obtain a listing of all object names and types in your schema, plus the following information:**
  - Date created
  - Date of last modification
  - Status (valid or invalid)

### **ALL\_OBJECTS:**

- **Query ALL\_OBJECTS to see all the objects to which you have access.**

**ORACLE**

## **USER\_OBJECTS and ALL\_OBJECTS Views**

You can query the USER\_OBJECTS view to see the names and types of all the objects in your schema. There are several columns in this view:

- **OBJECT\_NAME:** Name of the object
- **OBJECT\_ID:** Dictionary object number of the object
- **OBJECT\_TYPE:** Type of object (such as TABLE, VIEW, INDEX, SEQUENCE)
- **CREATED:** Time stamp for the creation of the object
- **LAST\_DDL\_TIME:** Time stamp for the last modification of the object resulting from a data definition language (DDL) command
- **STATUS:** Status of the object (VALID, INVALID, or N/A)
- **GENERATED:** Was the name of this object system generated? (Y | N)

**Note:** This is not a complete listing of the columns. For a complete listing, see “USER\_OBJECTS” in the *Oracle Database Reference*.

You can also query the ALL\_OBJECTS view to see a listing of all objects to which you have access.

## USER\_OBJECTS View

```
SELECT object_name, object_type, created, status
FROM   user_objects
ORDER BY object_type;
```

OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
1 LOC_COUNTRY_IX	INDEX	19-MAY-09	VALID

...

53 EMPLOYEES2	TABLE	22-MAY-09	VALID
54 SECURE_EMPLOYEES	TRIGGER	19-MAY-09	VALID
55 UPDATE_JOB_HISTORY	TRIGGER	19-MAY-09	VALID
56 EMP_DETAILS_VIEW	VIEW	19-MAY-09	VALID

...

ORACLE

## USER\_OBJECTS View

The example shows the names, types, dates of creation, and status of all objects that are owned by this user.

The OBJECT\_TYPE column holds the values of either TABLE, VIEW, SEQUENCE, INDEX, PROCEDURE, FUNCTION, PACKAGE, or TRIGGER.

The STATUS column holds a value of VALID, INVALID, or N/A. Although tables are always valid, the views, procedures, functions, packages, and triggers may be invalid.

## The CAT View

For a simplified query and output, you can query the CAT view. This view contains only two columns: TABLE\_NAME and TABLE\_TYPE. It provides the names of all your INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, or UNDEFINED objects.

**Note:** CAT is a synonym for USER\_CATALOG—a view that lists tables, views, synonyms and sequences owned by the user.

# Table Information

USER\_TABLES:

```
DESCRIBE user_tables
```

Name	Null	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLESPACE_NAME		VARCHAR2(30)
CLUSTER_NAME		VARCHAR2(30)
IOT_NAME		VARCHAR2(30)

...

```
SELECT table_name  
FROM user_tables;
```

TABLE_NAME
1 REGIONS
2 LOCATIONS
3 DEPARTMENTS
4 JOBS
5 EMPLOYEES
6 JOB_HISTORY

...

ORACLE

## Table Information

You can use the USER\_TABLES view to obtain the names of all your tables. The USER\_TABLES view contains information about your tables. In addition to providing the table name, it contains detailed information about the storage.

The TABS view is a synonym of the USER\_TABLES view. You can query it to see a listing of tables that you own:

```
SELECT table_name  
FROM tabs;
```

**Note:** For a complete listing of the columns in the USER\_TABLES view, see “USER\_TABLES” in the *Oracle Database Reference*.

You can also query the ALL\_TABLES view to see a listing of all tables to which you have access.

# Column Information

USER\_TAB\_COLUMNS:

```
DESCRIBE user_tab_columns
```

Name	Null	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME	NOT NULL	VARCHAR2(30)
DATA_TYPE		VARCHAR2(106)
DATA_TYPE_MOD		VARCHAR2(3)
DATA_TYPE_OWNER		VARCHAR2(30)
DATA_LENGTH	NOT NULL	NUMBER
DATA_PRECISION		NUMBER
DATA_SCALE		NUMBER
NULLABLE		VARCHAR2(1)

...

ORACLE

## Column Information

You can query the USER\_TAB\_COLUMNS view to find detailed information about the columns in your tables. Although the USER\_TABLES view provides information about your table names and storage, detailed column information is found in the USER\_TAB\_COLUMNS view.

This view contains information such as:

- Column names
- Column data types
- Length of data types
- Precision and scale for NUMBER columns
- Whether nulls are allowed (Is there a NOT NULL constraint on the column?)
- Default value

**Note:** For a complete listing and description of the columns in the USER\_TAB\_COLUMNS view, see “USER\_TAB\_COLUMNS” in the *Oracle Database Reference*.

# Column Information

```
SELECT column_name, data_type, data_length,
       data_precision, data_scale, nullable
  FROM user_tab_columns
 WHERE table_name = 'EMPLOYEES';
```

	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION
1	EMPLOYEE_ID	NUMBER	22	6
2	FIRST_NAME	VARCHAR2	20	(null)
3	LAST_NAME	VARCHAR2	25	(null)
4	EMAIL	VARCHAR2	25	(null)
5	PHONE_NUMBER	VARCHAR2	20	(null)
6	HIRE_DATE	DATE	7	(null)
7	JOB_ID	VARCHAR2	10	(null)
8	SALARY	NUMBER	22	8
9	COMMISSION_PCT	NUMBER	22	2
10	MANAGER_ID	NUMBER	22	6
11	DEPARTMENT_ID	NUMBER	22	4

ORACLE

## Column Information (continued)

By querying the USER\_TAB\_COLUMNS table, you can find details about your columns such as the names, data types, data type lengths, null constraints, and default value for a column.

The example shown displays the columns, data types, data lengths, and null constraints for the EMPLOYEES table. Note that this information is similar to the output from the DESCRIBE command.

To view information about columns set as unused, you use the USER\_UNUSED\_COL\_TABS dictionary view.

**Note:** Names of the objects in Data Dictionary are in uppercase.

## Constraint Information

- `USER_CONSTRAINTS` describes the constraint definitions on your tables.
- `USER_CONS_COLUMNS` describes columns that are owned by you and that are specified in constraints.

```
DESCRIBE user_constraints
```

Name	Null	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
CONSTRAINT_TYPE		VARCHAR2(1)
TABLE_NAME	NOT NULL	VARCHAR2(30)
SEARCH_CONDITION		LONG()
R_OWNER		VARCHAR2(30)
R_CONSTRAINT_NAME		VARCHAR2(30)
DELETE_RULE		VARCHAR2(9)
STATUS		VARCHAR2(8)

...

ORACLE

## Constraint Information

You can find out the names of your constraints, the type of constraint, the table name to which the constraint applies, the condition for check constraints, foreign key constraint information, deletion rule for foreign key constraints, the status, and many other types of information about your constraints.

**Note:** For a complete listing and description of the columns in the `USER_CONSTRAINTS` view, see “`USER_CONSTRAINTS`” in the *Oracle Database Reference*.

## USER\_CONSTRAINTS: Example

```
SELECT constraint_name, constraint_type,
       search_condition, r_constraint_name,
       delete_rule, status
  FROM user_constraints
 WHERE table_name = 'EMPLOYEES';
```

#	CONSTRAINT_NAME	C...	SEARCH_CONDITION	R_CONSTR...	DELET...	STATUS
1	EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL	(null)	(null)	ENABLED
2	EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL	(null)	(null)	ENABLED
3	EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL	(null)	(null)	ENABLED
4	EMP_JOB_NN	C	"JOB_ID" IS NOT NULL	(null)	(null)	ENABLED
5	EMP_SALARY_MIN	C	salary > 0	(null)	(null)	ENABLED
6	EMP_EMAIL_UK	U	(null)	(null)	(null)	ENABLED
7	EMP_EMP_ID_PK	P	(null)	(null)	(null)	ENABLED
8	EMP_DEPT_FK	R	(null)	DEPT_ID_PK	NO ACTION	ENABLED
9	EMP_JOB_FK	R	(null)	JOB_ID_PK	NO ACTION	ENABLED
10	EMP_MANAGER_FK	R	(null)	EMP_EMP_ID_PK	NO ACTION	ENABLED

ORACLE

## USER\_CONSTRAINTS: Example

In the example shown, the USER\_CONSTRAINTS view is queried to find the names, types, check conditions, name of the unique constraint that the foreign key references, deletion rule for a foreign key, and status for constraints on the EMPLOYEES table.

The CONSTRAINT\_TYPE can be:

- C (check constraint on a table , or NOT NULL)
- P (primary key)
- U (unique key)
- R (referential integrity)
- V (with check option, on a view)
- O (with read-only, on a view)

The DELETE\_RULE can be:

- **CASCADE:** If the parent record is deleted, the child records are deleted too.
- **SET NULL:** If the parent record is deleted, change the respective child record to null.
- **NO ACTION:** A parent record can be deleted only if no child records exist.

The STATUS can be:

- **ENABLED:** Constraint is active.
- **DISABLED:** Constraint is made not active.

## Querying USER\_CONS\_COLUMNS

```
DESCRIBE user_cons_columns
```

Name	Null	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
POSITION		NUMBER

```
SELECT constraint_name, column_name
FROM   user_cons_columns
WHERE  table_name = 'EMPLOYEES' ;
```

CONSTRAINT_NAME	COLUMN_NAME
1 EMP_LAST_NAME_NN	LAST_NAME
2 EMP_EMAIL_NN	EMAIL
3 EMP_HIRE_DATE_NN	HIRE_DATE
4 EMP_JOB_NN	JOB_ID
5 EMP_SALARY_MIN	SALARY
6 EMP_EMAIL_UK	EMAIL

...

ORACLE

## Querying USER\_CONS\_COLUMNS

To find the names of the columns to which a constraint applies, query the USER\_CONS\_COLUMNS dictionary view. This view tells you the name of the owner of a constraint, the name of the constraint, the table that the constraint is on, the names of the columns with the constraint, and the original position of column or attribute in the definition of the object.

**Note:** A constraint may apply to more than one column.

You can also write a join between USER\_CONSTRAINTS and USER\_CONS\_COLUMNS to create customized output from both tables.

# View Information

1

```
DESCRIBE user_views
```

Name	Null	Type
VIEW_NAME	NOT NULL	VARCHAR2(30)
TEXT_LENGTH		NUMBER
TEXT		LONG()

2

```
SELECT view_name FROM user_views;
```

```
VIEW_NAME  
1 EMP_DETAILS_VIEW
```

3

```
SELECT text FROM user_views  
WHERE view_name = 'EMP_DETAILS_VIEW' ;
```

```
TEXT  
1 SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.location_id, l.co  
...  
AND c.region_id = r.region_id AND j.job_id = e.job_id WITH READ ONLY
```

ORACLE

3 - 14

Copyright © 2009, Oracle. All rights reserved.

## View Information

After your view is created, you can query the data dictionary view called USER\_VIEWS to see the name of the view and the view definition. The text of the SELECT statement that constitutes your view is stored in a LONG column. The LENGTH column is the number of characters in the SELECT statement. By default, when you select from a LONG column, only the first 80 characters of the column's value are displayed. To see more than 80 characters in SQL\*Plus, use the SET LONG command:

```
SET LONG 1000
```

In the examples in the slide:

1. The USER\_VIEWS columns are displayed. Note that this is a partial listing.
2. The names of your views are retrieved
3. The SELECT statement for the EMP\_DETAILS\_VIEW is displayed from the dictionary

## Data Access Using Views

When you access data by using a view, the Oracle Server performs the following operations:

- It retrieves the view definition from the data dictionary table USER\_VIEWS.
- It checks access privileges for the view base table.

# Sequence Information

```
DESCRIBE user_sequences
```

Name	Null	Type
SEQUENCE_NAME	NOT NULL	VARCHAR2(30)
MIN_VALUE		NUMBER
MAX_VALUE		NUMBER
INCREMENT_BY	NOT NULL	NUMBER
CYCLE_FLAG		VARCHAR2(1)
ORDER_FLAG		VARCHAR2(1)
CACHE_SIZE	NOT NULL	NUMBER
LAST_NUMBER	NOT NULL	NUMBER

ORACLE

## Sequence Information

The USER\_SEQUENCES view describes all sequences that you own. When you create the sequence, you specify criteria that are stored in the USER\_SEQUENCES view. The columns in this view are:

- **SEQUENCE\_NAME:** Name of the sequence
- **MIN\_VALUE:** Minimum value of the sequence
- **MAX\_VALUE:** Maximum value of the sequence
- **INCREMENT\_BY:** Value by which the sequence is incremented
- **CYCLE\_FLAG:** Does sequence wrap around on reaching the limit?
- **ORDER\_FLAG:** Are sequence numbers generated in order?
- **CACHE\_SIZE:** Number of sequence numbers to cache
- **LAST\_NUMBER:** Last sequence number written to disk. If a sequence uses caching, the number written to disk is the last number placed in the sequence cache. This number is likely to be greater than the last sequence number that was used.

# Confirming Sequences

- Verify your sequence values in the `USER_SEQUENCES` data dictionary table.

```
SELECT    sequence_name, min_value, max_value,
          increment_by, last_number
FROM      user_sequences;
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPARTMENTS_SEQ	1	9990	10	280
EMPLOYEES_SEQ	1	99999999999999...	1	207
LOCATIONS_SEQ	1	9900	100	3300

- The `LAST_NUMBER` column displays the next available sequence number if `NOCACHE` is specified.

ORACLE

## Confirming Sequences

After creating your sequence, it is documented in the data dictionary. Because a sequence is a database object, you can identify it in the `USER_OBJECTS` data dictionary table.

You can also confirm the settings of the sequence by selecting from the `USER_SEQUENCES` data dictionary view.

### Viewing the Next Available Sequence Value Without Incrementing It

If the sequence was created with `NOCACHE`, it is possible to view the next available sequence value without incrementing it by querying the `USER_SEQUENCES` table.

## Index Information

- `USER_INDEXES` provides information about your indexes.
- `USER_IND_COLUMNS` describes columns comprising your indexes and columns of indexes on your tables.

```
DESCRIBE user_indexes
```

Name	Null	Type
INDEX_NAME	NOT NULL	VARCHAR2(30)
INDEX_TYPE		VARCHAR2(27)
TABLE_OWNER	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLE_TYPE		VARCHAR2(11)
UNIQUENESS		VARCHAR2(9)

...

ORACLE

## Index Information

You query the `USER_INDEXES` view to find out the names of your indexes, the table name on which the index is created, and whether the index is unique.

**Note:** For a complete listing and description of the columns in the `USER_INDEXES` view, see “`USER_INDEXES`” in the *Oracle Database Reference 11g Release 2 (11.1)*.

## USER\_INDEXES: Examples

a

```
SELECT index_name, table_name, uniqueness  
FROM   user_indexes  
WHERE  table_name = 'EMPLOYEES';
```

	INDEX_NAME	TABLE_NAME	UNIQUENESS
1	EMP_EMAIL_UK	EMPLOYEES	UNIQUE
2	EMP_EMP_ID_PK	EMPLOYEES	UNIQUE
3	EMP_DEPARTMENT_IK	EMPLOYEES	NONUNIQUE
4	EMP_JOB_IK	EMPLOYEES	NONUNIQUE
5	EMP_MANAGER_IK	EMPLOYEES	NONUNIQUE
6	EMP_NAME_IK	EMPLOYEES	NONUNIQUE

b

```
SELECT index_name, table_name  
FROM   user_indexes  
WHERE  table_name = 'emp_lib';
```

	INDEX_NAME	TABLE_NAME
1	SYS_C0011777	EMP_LIB

ORACLE

## USER\_INDEXES: Example

In the slide example **a**, the USER\_INDEXES view is queried to find the name of the index, name of the table on which the index is created, and whether the index is unique.

In the slide example **b**, observe that the Oracle Server gives a generic name to the index that is created for the PRIMARY KEY column. The EMP\_LIB table is created by using the following code:

```
CREATE TABLE EMP_LIB  
(book_id NUMBER(6) PRIMARY KEY ,  
 title VARCHAR2(25) ,  
 category VARCHAR2(20));
```

CREATE TABLE succeeded.

## Querying USER\_IND\_COLUMNS

```
DESCRIBE user_ind_columns
```

Name	Null	Type
INDEX_NAME		VARCHAR2(30)
TABLE_NAME		VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
COLUMN_POSITION		NUMBER
COLUMN_LENGTH		NUMBER
CHAR_LENGTH		NUMBER
DESCEND		VARCHAR2(4)

```
SELECT index_name, column_name, table_name  
FROM   user_ind_columns  
WHERE  index_name = 'lname_idx';
```

INDEX_NAME	COLUMN_NAME	TABLE_NAME
1 LNAME_IDX	LAST_NAME	EMP_TEST

ORACLE

## Querying USER\_IND\_COLUMNS

The USER\_IND\_COLUMNS dictionary view provides information such as the name of the index, name of the indexed table, name of a column within the index, and the column's position within the index.

For the slide example, the emp\_test table and LNAME\_IDX index are created by using the following code:

```
CREATE TABLE emp_test AS SELECT * FROM employees;  
CREATE INDEX LNAME_IDX ON emp_test(Last_Name);
```

# Synonym Information

```
DESCRIBE user_synonyms
```

Name	Null	Type
SYNONYM_NAME	NOT NULL	VARCHAR2(30)
TABLE_OWNER		VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
DB_LINK		VARCHAR2(128)

```
SELECT *
FROM    user_synonyms;
```

SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK
TEAM2	ORA22	DEPARTMENTS	(null)

ORACLE

## Synonym Information

The USER\_SYNONYMS dictionary view describes private synonyms (synonyms that you own). You can query this view to find your synonyms. You can query ALL\_SYNONYMS to find out the name of all the synonyms that are available to you and the objects on which these synonyms apply.

The columns in this view are:

- **SYNONYM\_NAME:** Name of the synonym
- **TABLE\_OWNER:** Owner of the object that is referenced by the synonym
- **TABLE\_NAME:** Name of the table or view that is referenced by the synonym
- **DB\_LINK:** Name of the database link reference (if any)

## Adding Comments to a Table

- You can add comments to a table or column by using the COMMENT statement:

```
COMMENT ON TABLE employees  
IS 'Employee Information';
```

```
COMMENT ON COLUMN employees.first_name  
IS 'First name of the employee';
```

- Comments can be viewed through the data dictionary views:
  - ALL\_COL\_COMMENTS
  - USER\_COL\_COMMENTS
  - ALL\_TAB\_COMMENTS
  - USER\_TAB\_COMMENTS

ORACLE

3 - 21

Copyright © 2009, Oracle. All rights reserved.

### Adding Comments to a Table

You can add a comment of up to 4,000 bytes about a column, table, view, or snapshot by using the COMMENT statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the COMMENTS column:

- ALL\_COL\_COMMENTS
- USER\_COL\_COMMENTS
- ALL\_TAB\_COMMENTS
- USER\_TAB\_COMMENTS

### Syntax

```
COMMENT ON {TABLE table | COLUMN table.column}  
IS 'text';
```

You can drop a comment from the database by setting it to empty string (''):

```
COMMENT ON TABLE employees IS '';
```

# Manipulating Large Data Sets

# 4

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

## Using Subqueries to Manipulate Data

You can use subqueries in data manipulation language (DML) statements to:

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

ORACLE

4 - 2

Copyright © 2009, 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.

**Note:** You learned how to update and delete rows based on another table in the course titled *Oracle Database 11g: SQL Fundamentals I*.

# Retrieving Data by Using a Subquery as Source

```
SELECT department_name, city
FROM   departments
NATURAL JOIN (SELECT l.location_id, l.city, l.country_id
               FROM loc l
               JOIN countries c
                 ON(l.country_id = c.country_id)
               JOIN regions USING(region_id)
              WHERE region_name = 'Europe');
```

DEPARTMENT_NAME	CITY
Human Resources	London
Sales	Oxford
Public Relations	Munich

ORACLE

4 - 3

Copyright © 2009, Oracle. All rights reserved.

## Retrieving Data by 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. As with a database view, the SELECT statement in the subquery can be as simple or as complex as you like.

When a database view is created, the associated SELECT statement is stored in the data dictionary. In situations where you do not have the necessary privileges to create database views, or when you would like to test the suitability of a SELECT statement to become a view, you can use an inline view.

With inline views, you can have all the code needed to support the query in one place. This means that you can avoid the complexity of creating a separate database view. The example in the slide shows how to use an inline view to display the department name and the city in Europe. The subquery in the FROM clause fetches the location ID, city name, and the country by joining three different tables. The output of the inner query is considered as a table for the outer query. The inner query is similar to that of a database view but does not have any physical name.

For the example in the slide, the loc table is created by running the following statement:

```
CREATE TABLE loc AS SELECT * FROM locations;
```

## Inserting by Using a Subquery as a Target

```
INSERT INTO (SELECT l.location_id, l.city, l.country_id
             FROM   locations l
             JOIN   countries c
             ON(l.country_id = c.country_id)
             JOIN   regions USING(region_id)
             WHERE  region_name = 'Europe')
VALUES (3300, 'Cardiff', 'UK');
```

1 rows inserted

ORACLE

4 - 5

Copyright © 2009, Oracle. All rights reserved.

## Inserting by 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 location ID or leave out a value for a mandatory NOT NULL column.

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

The example in the slide uses a subquery in the place of LOC to create a record for a new European city.

**Note:** You can also perform the INSERT operation on the EUROPEAN\_CITIES view by using the following code:

```
INSERT INTO european_cities
VALUES (3300, 'Cardiff', 'UK');
```

# Inserting by Using a Subquery as a Target

Verify the results.

```
SELECT location_id, city, country_id  
FROM loc
```

	LOCATION_ID	CITY	COUNTRY_ID
20	2900	Geneva	CH
21	3000	Bern	CH
22	3100	Utrecht	NL
23	3200	Mexico City	MX
24	3300	Cardiff	UK

ORACLE

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

The example in the slide shows that the insert via the inline view created a new record in the base table LOC.

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

```
SELECT l.location_id, l.city, l.country_id  
FROM loc l  
JOIN countries c  
ON(l.country_id = c.country_id)  
JOIN regions USING(region_id)  
WHERE region_name = 'Europe'
```

	LOCATION_ID	CITY	COUNTRY_ID
6	2700	Munich	DE
7	2900	Geneva	CH
8	3000	Bern	CH
9	3100	Utrecht	NL
10	3300	Cardiff	UK

# Overview of the Explicit Default Feature

- Use the `DEFAULT` keyword as a column value where the default column value is desired.
- 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

4 - 7

Copyright © 2009, 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 having to hard code 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; therefore, 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

4 - 8

Copyright © 2009, 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 in 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 *SQL Fundamentals I*.

# 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 inserted

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

ORACLE

4 - 9

Copyright © 2009, 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:

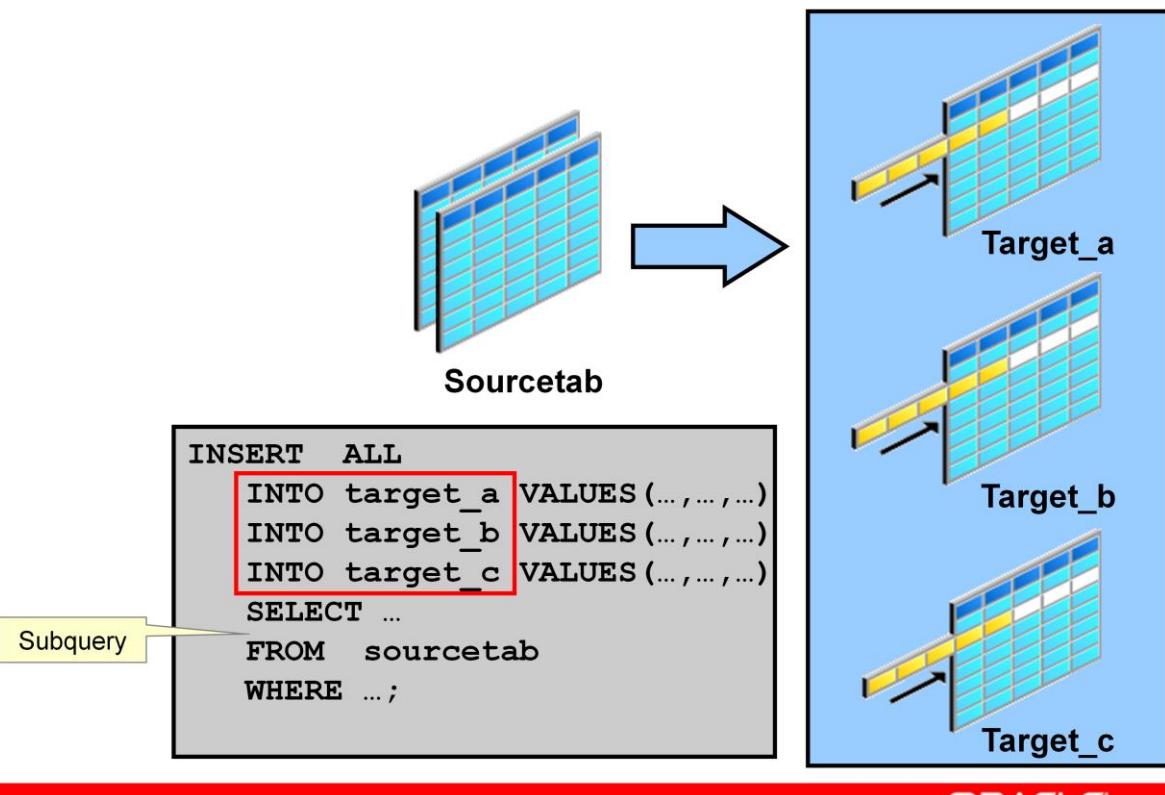
<code>table</code>	Is the table name
<code>column</code>	Is the name of the column in the table to populate
<code>subquery</code>	Is the subquery that returns rows into the table

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;
```

**Note:** You use the `LOG ERRORS` clause in your DML statement to enable the DML operation to complete regardless of errors. Oracle writes the details of the error message to an error-logging table that you have created. For more information, see *Oracle Database 11g SQL Reference*.

# Overview of Multitable INSERT Statements



4 - 10

Copyright © 2009, Oracle. All rights reserved.

ORACLE

## 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 are useful 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

- Use the `INSERT...SELECT` statement to insert rows into multiple tables as part of a single DML statement.
- Multitable `INSERT` statements are 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 perform multiple inserts by using the `IF...THEN` syntax

ORACLE

4 - 11

Copyright © 2009, 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. Without multitable `INSERT`, 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 INSERT ALL**
- **Pivoting INSERT**
- **Conditional INSERT FIRST**

ORACLE

4 - 12

Copyright © 2009, Oracle. All rights reserved.

### Types of Multitable INSERT Statements

You use different clauses to indicate the type of INSERT to be executed. The types of multitable INSERT statements are:

- **Unconditional INSERT:** For each row returned by the subquery, a row is inserted into each of the target tables.
- **Conditional INSERT ALL:** For each row returned by the subquery, a row is inserted into each target table if the specified condition is met.
- **Pivoting INSERT:** This is a special case of the unconditional INSERT ALL.
- **Conditional INSERT FIRST:** For each row returned by the subquery, a row is inserted into the very first target table in which the condition is met.

# Multitable INSERT Statements

- Syntax for multitable INSERT:

```
INSERT [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

4 - 13

Copyright © 2009, 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.

## 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 by 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 inserted

ORACLE

4 - 15

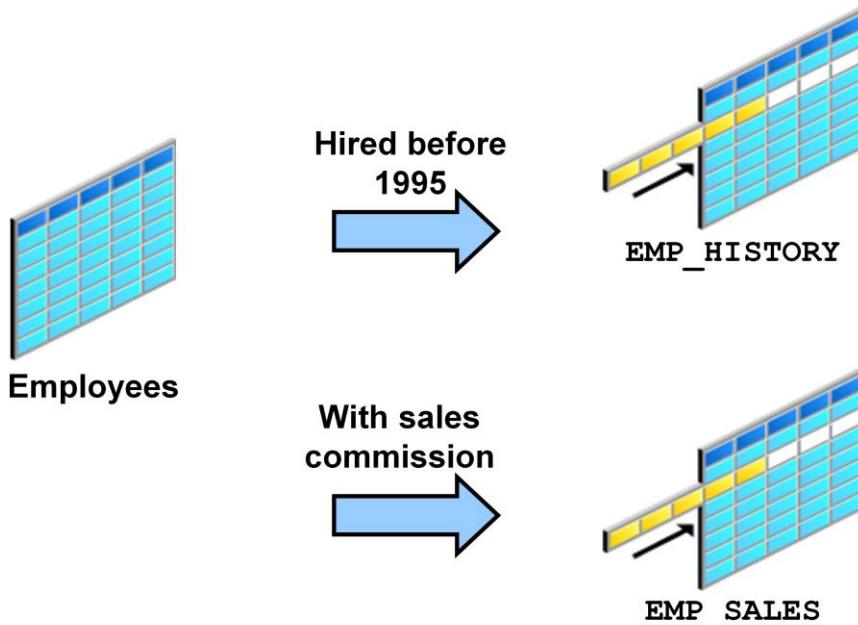
Copyright © 2009, Oracle. All rights reserved.

### 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.

## Conditional INSERT ALL: Example



ORACLE

### Conditional INSERT ALL: Example

For all employees in the employees tables, if the employee was hired before 1995, insert that employee record into the employee history. If the employee earns a sales commission, insert the record information into the `EMP_SALES` table. The SQL statement is shown on the next page.

## Conditional INSERT ALL

```
INSERT ALL
  WHEN HIREDATE < '01-JAN-95' THEN
    INTO emp_history VALUES (EMPID, HIREDATE, SAL)
  WHEN COMM IS NOT NULL THEN
    INTO emp_sales VALUES (EMPID, COMM, SAL)
  SELECT employee_id EMPID, hire_date HIREDATE,
         salary SAL, commission_pct COMM
  FROM employees
```

48 rows inserted

ORACLE

4 - 18

Copyright © 2009, Oracle. All rights reserved.

### Conditional INSERT ALL

The example in the slide is similar to the example in the previous slide because it inserts rows into both the EMP\_HISTORY and the EMP\_SALES tables. The SELECT statement retrieves details such as employee ID, hire date, salary, and commission percentage for all employees from the EMPLOYEES table. Details such as employee ID, hire date, and salary are inserted into the EMP\_HISTORY table. Details such as employee ID, commission percentage, and salary are inserted into the EMP\_SALES table.

This INSERT statement is referred to as a conditional `INSERT ALL` 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 hire date was prior to 1995 are inserted in the EMP\_HISTORY table. Similarly, only those rows where the value of commission percentage is not null are inserted in the EMP\_SALES table.

```
SELECT count(*) FROM emp_history;
```

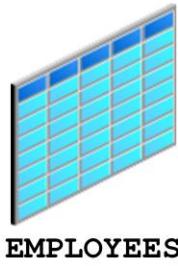
COUNT(*)	
1	13

```
SELECT count(*) FROM emp_sales;
```

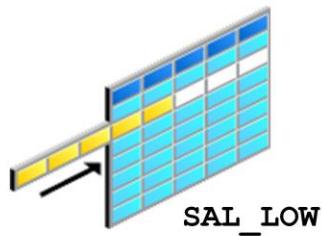
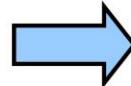
COUNT(*)	
1	35

## Conditional INSERT FIRST: Example

**Scenario:** If an employee salary is 2,000, the record is inserted into the SAL\_LOW table only.



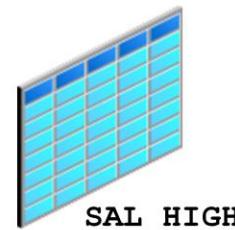
**Salary < 5,000**



**5000 <= Salary  
<= 10,000**



**Otherwise**



**ORACLE**

### Conditional INSERT FIRST: Example

For all employees in the EMPLOYEES table, insert the employee information into the first target table that meets the condition. In the example, if an employee has a salary of 2,000, the record is inserted into the SAL\_LOW table only. The SQL statement is shown on the next page.

## Conditional INSERT FIRST

```
INSERT FIRST  
WHEN salary < 5000 THEN  
    INTO sal_low VALUES (employee_id, last_name, salary)  
WHEN salary between 5000 and 10000 THEN  
    INTO sal_mid VALUES (employee_id, last_name, salary)  
ELSE  
    INTO sal_high VALUES (employee_id, last_name, salary)  
SELECT employee_id, last_name, salary  
FROM employees
```

107 rows inserted

ORACLE

4 - 21

Copyright © 2009, Oracle. All rights reserved.

### Conditional INSERT FIRST

The SELECT statement retrieves details such as employee ID, last name, and salary for every employee in the EMPLOYEES table. For each employee record, it is inserted into the very first target table that meets the condition.

This `INSERT` statement is referred to as a conditional `INSERT FIRST`. The `WHEN salary < 5000` condition is evaluated first. If this first `WHEN` clause evaluates to true, the Oracle server executes the corresponding `INTO` clause and inserts the record into the `SAL_LOW` table. It skips subsequent `WHEN` clauses for this row.

If the row does not satisfy the first `WHEN` condition (`WHEN salary < 5000`), the next condition (`WHEN salary between 5000 and 10000`) is evaluated. If this condition evaluates to true, the record is inserted into the `SAL_MID` table, and the last condition is skipped.

If neither the first condition (`WHEN salary < 5000`) nor the second condition (`WHEN salary between 5000 and 10000`) is evaluated to true, the Oracle server executes the corresponding `INTO` clause for the `ELSE` clause.

## Pivoting INSERT

Convert the set of sales records from the nonrelational database table to relational format.



Emp_ID	Week_ID	MON	TUES	WED	THUR	FRI
176	6	2000	3000	4000	5000	6000

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

ORACLE

4 - 23

Copyright © 2009, 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.

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

To solve this problem, 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 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 inserted
```

ORACLE

4 - 24

Copyright © 2009, Oracle. All rights reserved.

### Pivoting INSERT (continued)

In the example in 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)

## MERGE Statement

- Provides the ability to conditionally update, insert, or delete 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

4 - 26

Copyright © 2009, Oracle. All rights reserved.

### MERGE Statement

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 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.

## MERGE Statement Syntax

You can conditionally insert, update, or delete 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 = col1_val,
     col2 = col2_val
 WHEN NOT MATCHED THEN
   INSERT (column_list)
   VALUES (column_values);
```

ORACLE

4 - 27

Copyright © 2009, Oracle. All rights reserved.

### Merging Rows

You can update existing rows, and insert new rows conditionally by using the MERGE statement. Using the MERGE statement, you can delete obsolete rows at the same time as you update rows in a table. To do this, you include a DELETE clause with its own WHERE clause in the syntax of 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 on which the MERGE operation either updates or inserts
WHEN MATCHED   join	Instructs the server how to respond to the results of the condition
WHEN NOT MATCHED	

**Note:** For more information, see *Oracle Database 11g SQL Reference*.

## Merging Rows: Example

Insert or update rows in the COPY\_EMP3 table to match the EMPLOYEES table.

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM 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,
...
DELETE WHERE (e.COMMISSION_PCT IS NOT NULL)
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

4 - 28

Copyright © 2009, Oracle. All rights reserved.

## Merging Rows: Example

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM 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*2,
c.commission_pct = e.commission_pct,
c.manager_id = e.manager_id,
c.department_id = e.department_id
DELETE WHERE (e.COMMISSION_PCT IS NOT NULL)
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: Example

```
TRUNCATE TABLE copy_emp3;
SELECT * FROM copy_emp3;
0 rows selected
```

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM 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,
...
DELETE WHERE (e.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, ...)
```

```
SELECT * FROM copy_emp3;
107 rows selected.
```

ORACLE

4 - 30

Copyright © 2009, Oracle. All rights reserved.

### Merging Rows: Example (continued)

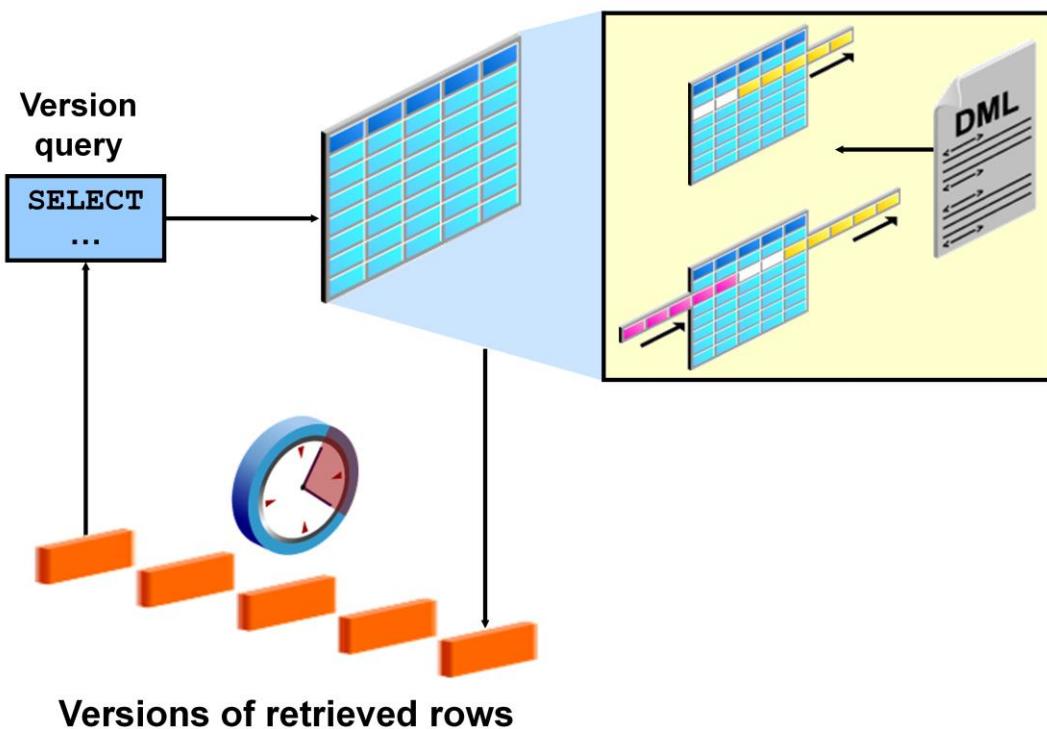
The examples in the slide show that the COPY\_EMP3 table is empty. The `c.employee_id = e.employee_id` condition is evaluated. The condition returns false—there are no matches. The logic falls into the WHEN NOT MATCHED clause, and the MERGE command inserts the rows of the EMPLOYEES table into the COPY\_EMP3 table. This means that the COPY\_EMP3 table now has exactly the same data as in the EMPLOYEES table.

```
SELECT employee_id, salary, commission_pct from copy_emp3;
```

EMPLOYEE_ID	SALARY	COMMISSION_PCT
1	144	(null)
2	143	(null)
3	202	(null)
4	141	(null)
5	174	0.3
...		

15	149	10500	0.2
16	206	8300	(null)
17	176	8600	0.2
18	124	5800	(null)
19	205	12000	(null)
20	178	7000	0.15

# Tracking Changes in Data



4 - 31

Copyright © 2009, Oracle. All rights reserved.

ORACLE

## 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 a system change number (SCN) or the time stamp range within 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 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 autotuned parameter. A query that includes a VERSIONS clause is referred to as a version query. The results of a version query behaves as though 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 an 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

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

2

```
COMMIT;
```

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

3

1

	SALARY
1	4200

3

	SALARY
1	5460
2	4200

ORACLE

4 - 32

Copyright © 2009, Oracle. All rights reserved.

## Example of the Flashback Version Query

In the example in 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, 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.

## VERSIONS BETWEEN Clause

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

	START_DATE	END_DATE	SALARY
1	18-JUN-09 05.07.10.000000000 PM (null)		5460
2	(null)	18-JUN-09 05.07.10.000000000 PM	4200

ORACLE

## VERSIONS BETWEEN Clause

You can use the VERSIONS BETWEEN clause to retrieve all 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 less than the lower bound time or the SCN of the BETWEEN clause, 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 bounds.

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

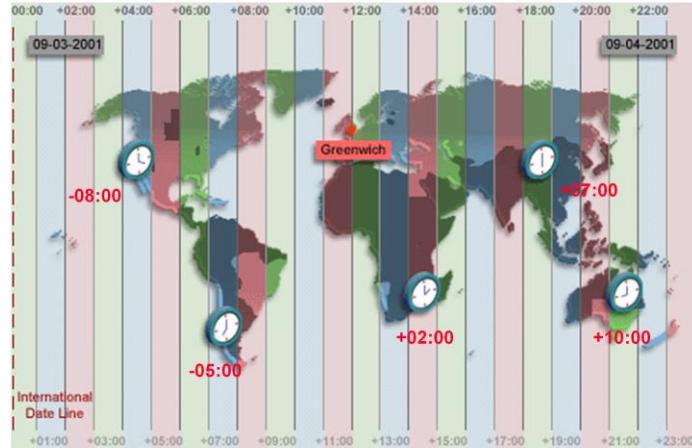
# **Managing Data in Different Time Zones**



**ORACLE®**

Copyright © 2009, Oracle. All rights reserved.

# Time Zones



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

ORACLE

5 - 2

Copyright © 2009, 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 (GMT). GMT is now known as Coordinated Universal Time (UTC). UTC 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 UTC 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**

## **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 user's 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 user session
  - Has a data type of DATE
- CURRENT\_TIMESTAMP:
  - Returns the current date and time from the user session
  - Has a data type of TIMESTAMP WITH TIME ZONE
- LOCALTIMESTAMP:
  - Returns the current date and time from the user session
  - Has a data type of TIMESTAMP

**ORACLE**

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

The CURRENT\_DATE and CURRENT\_TIMESTAMP functions return the current date and current time stamp, 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 through 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 national language support (NLS)-sensitive—that is, the results will be in the current NLS calendar and datetime formats.

**Note:** The SYSDATE function returns the current date and time as a DATE data type. You learned how to use the SYSDATE function in the course titled *Oracle Database 11g: SQL Fundamentals I*.

## Comparing Date and Time in a Session's Time Zone

The TIME\_ZONE parameter is set to -5:00 and then SELECT statements for each date and time are executed to compare differences.

```
ALTER SESSION  
SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';  
ALTER SESSION SET TIME_ZONE = '-5:00';
```

```
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL; 1
```

```
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL; 2
```

```
SELECT SESSIONTIMEZONE, LOCALTIMESTAMP FROM DUAL; 3
```

ORACLE

5 - 5

Copyright © 2009, Oracle. All rights reserved.

### Comparing Date and Time in a Session's Time Zone

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).

The example in the slide illustrates that the session is altered to set the TIME\_ZONE parameter to -5:00. Then the SELECT statement for CURRENT\_DATE, CURRENT\_TIMESTAMP, and LOCALTIMESTAMP is executed to observe the differences in format.

**Note:** 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.

# Comparing Date and Time in a Session's Time Zone

Results of queries:

ALTER SESSION succeeded.	
SESSIONTIMEZONE	CURRENT_DATE
1 -05:00	23-JUN-2009 01:34:52
SESSIONTIMEZONE	CURRENT_TIMESTAMP
1 -05:00	23-JUN-09 01.35.26.239882000 AM -05:00
SESSIONTIMEZONE	LOCALTIMESTAMP
1 -05:00	23-JUN-09 01.36.21.811798000 AM

1

2

3

ORACLE

## Comparing Date and Time in a Session's Time Zone (continued)

In this case, the CURRENT\_DATE function returns the current date in the session's time zone, the CURRENT\_TIMESTAMP function returns the current date and time in the session's time zone as a value of the data type TIMESTAMP WITH TIME ZONE, and the LOCALTIMESTAMP function returns the current date and time in the session's time zone.

## DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone:

```
SELECT DBTIMEZONE FROM DUAL;
```

DBTIMEZONE
1 +00:00

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

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

SESSIONTIMEZONE
1 -05:00

ORACLE

## 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 in 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 in 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 Types

Data Type	Fields
TIMESTAMP	Year, Month, Day, Hour, Minute, Second with fractional seconds
TIMESTAMP WITH TIME ZONE	Same as the TIMESTAMP data type; also includes: TIMEZONE_HOUR, and TIMEZONE_MINUTE or TIMEZONE_REGION
TIMESTAMP WITH LOCAL TIME ZONE	Same as the TIMESTAMP data type; also includes a time zone offset in its value

ORACLE

5 - 8

Copyright © 2009, Oracle. All rights reserved.

### TIMESTAMP Data Types

The TIMESTAMP data type is an extension of the DATE data type.

**`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 through 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**

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

**ORACLE**

## **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 employees;
```

HIRE_DATE
1 21-JUN-99
2 13-JAN-00
3 17-SEP-87
4 17-FEB-96
5 17-AUG-97
6 07-JUN-94
7 07-JUN-94
8 07-JUN-94
9 07-JUN-94

B

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

HIRE_DATE
1 21-JUN-99 12.00.00.000000000 AM
2 13-JAN-00 12.00.00.000000000 AM
3 17-SEP-87 12.00.00.000000000 AM
4 17-FEB-96 12.00.00.000000000 AM
5 17-AUG-97 12.00.00.000000000 AM
6 07-JUN-94 12.00.00.000000000 AM
7 07-JUN-94 12.00.00.000000000 AM
8 07-JUN-94 12.00.00.000000000 AM

...

ORACLE

5 - 10

Copyright © 2009, Oracle. All rights reserved.

## TIMESTAMP Data Type: Example

In the slide, example A shows the data from the `hire_date` column of the `EMPLOYEES` table when the data type of the column is `DATE`. In example B, the table is altered and the data type of the `hire_date` column is made into `TIMESTAMP`. The output shows the differences in display. 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 time stamp. If none is specified, as in this example, it defaults to 6.

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

```
ALTER TABLE employees  
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, AM, and PM. To obtain the date in this format, you can apply a format mask or a function to the date value.

## Comparing TIMESTAMP Data Types

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

```
INSERT INTO web_orders values  
(current_date, current_timestamp + 2);
```

```
SELECT * FROM web_orders;
```

	ORDER_DATE	DELIVERY_TIME
1	23-JUN-09 01.56.39.000000000 AM -05:00	25-JUN-09 01.56.39.000000000 AM

ORACLE

## Comparing TIMESTAMP Data Types

In the example in the slide, a new table `web_orders` is created with a column of data type `TIMESTAMP WITH TIME ZONE` and a column of data type `TIMESTAMP WITH LOCAL TIME ZONE`. This table is populated whenever a `web_order` is placed. The time stamp and time zone for the user placing the order is inserted based on the `CURRENT_DATE` value. The local time stamp and time zone is populated by inserting two days from the `CURRENT_TIMESTAMP` value into it every time an order is placed. When a Web-based company guarantees shipping, they can estimate their delivery time based on the time zone of the person placing the order.

## 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 - 12

Copyright © 2009, 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 is 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 Fields

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

ORACLE

### 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: 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
1	123 0-8
2	155 200-0
3	678 200-11

ORACLE

5 - 15

Copyright © 2009, 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.

### Examples

- INTERVAL '123-2' YEAR(3) TO MONTH  
Indicates an interval of 123 years, 2 months
- INTERVAL '123' YEAR(3)  
Indicates an interval of 123 years, 0 months
- INTERVAL '300' MONTH(3)  
Indicates an interval of 300 months
- INTERVAL '123' YEAR  
Returns an error because the default precision is 2, and 123 has three digits

## **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
1	100012	90 0:0:0.0
2	56098	6 3:30:16.0

**ORACLE**

### **INTERVAL DAY TO SECOND Data Type: Example**

In the example in the slide, you create the lab table with a `test_time` column of the INTERVAL DAY TO SECOND data type. You then insert into it the value `'90 00:00:00'` to indicate 90 days and 0 hours, 0 minutes, and 0 seconds, and `INTERVAL '6 03:30:16' DAY TO SECOND` to indicate 6 days, 3 hours, 30 minutes, and 16 seconds. 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)
1	2009

- 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)
1	Hartstein	17-FEB-1996 00:00:00	2
2	Kochhar	21-SEP-1989 00:00:00	9
3	De Haan	13-JAN-1993 00:00:00	1
4	Raphaely	07-DEC-1994 00:00:00	12
5	Weiss	18-JUL-1996 00:00:00	7

ORACLE

## 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 in the slide, the EXTRACT function is used to extract the YEAR from SYSDATE. In the second example in the slide, the EXTRACT function is used to extract the MONTH from the 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 'US/Eastern', 'Canada/Yukon' and 'Europe/London' time zones:**

```
SELECT TZ_OFFSET('US/Eastern') ,  
       TZ_OFFSET('Canada/Yukon') ,  
       TZ_OFFSET('Europe/London')  
  FROM DUAL;
```

	TZ_OFFSET('US/EASTERN')	TZ_OFFSET('CANADA/YUKON')	TZ_OFFSET('EUROPE/LONDON')
1	-04:00	-07:00	+01:00

**ORACLE**

### **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 its headquarters in Michigan, USA, which is in the 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 'US/Eastern' time zone is four hours behind UTC.
- The 'Canada/Yukon' time zone is seven hours behind UTC.
- The 'Europe/London' time zone is one hour ahead of UTC.

## **FROM\_TZ**

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

```
SELECT FROM_TZ(TIMESTAMP  
  '2000-07-12 08:00:00', 'Australia/North')  
FROM DUAL;
```

FROM_TZ(TIMESTAMP'2000-07-1208:00:00','AUSTRALIA/NORTH')
1 12-JUL-00 08.00.00.000000000 AM AUSTRALIA/NORTH

**ORACLE**

## **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 an 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, '`PDT`' for US/Pacific daylight time, and so on.

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

**Note:** To see a listing of valid values for the `TZR` and `TZD` format elements, query the `V$TIMEZONE_NAMES` dynamic performance view.

## **TO\_TIMESTAMP**

Display the character string '2007-03-06 11:00:00'  
as a TIMESTAMP value:

```
SELECT TO_TIMESTAMP ('2007-03-06 11:00:00',
                     'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```

```
TO_TIMESTAMP('2007-03-0611:00:00','YYYY-MM-DDHH:MI:SS')
06-MAR-07 11.00.00.000000000
```

**ORACLE**

### **TO\_TIMESTAMP**

The TO\_TIMESTAMP function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the 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 you omit fmt, 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 in the slide converts a character string to a value of TIMESTAMP.

**Note:** You use the TO\_TIMESTAMP\_TZ function to convert a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the TIMESTAMP WITH TIME ZONE data type. For more information about this function, see *Oracle Database SQL Language Reference 11g Release 1 (11.1)*.

## **TO\_YMINTERVAL**

Display a date that is one year and 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_YMININTERVAL
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**

### **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 in 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.

## **TO\_DSINTERVAL**

Display a date that is 100 days and 10 hours after the hire date for all the employees.

```
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
1	OConnell	06-21-99:12:00:00	09-29-99:10:00:00
2	Grant	01-13-00:12:00:00	04-22-00:10:00:00
3	Whalen	09-17-87:12:00:00	12-26-87:10:00:00
4	Hartstein	02-17-96:12:00:00	05-27-96:10:00:00
5	Fay	08-17-97:12:00:00	11-25-97:10:00:00
6	Mavris	06-07-94:12:00:00	09-15-94:10:00:00
7	Baer	06-07-94:12:00:00	09-15-94:10:00:00
8	Higgins	06-07-94:12:00:00	09-15-94:10:00:00
...			

**ORACLE**

## **TO\_DSINTERVAL**

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

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

# Daylight Saving Time

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

ORACLE

5 - 25

Copyright © 2009, 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 AM to 03:00:00 AM. The one-hour interval between 02:00:00 AM and 02:59:59 AM. does not exist. When daylight saving time goes out of effect, the time changes from 02:00:00 AM back to 01:00:01 AM, and the one-hour interval between 01:00:01 AM and 02:00:00 AM is repeated.

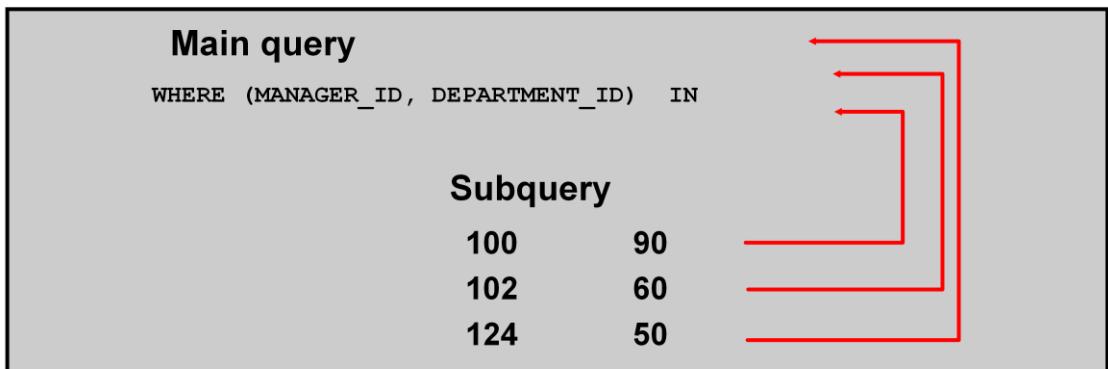
# **Retrieving Data by Using Subqueries**



**ORACLE®**

Copyright © 2009, Oracle. All rights reserved.

# Multiple-Column Subqueries



**Each row of the main query is compared to values from a multiple-row and multiple-column subquery.**

ORACLE

6 - 2

Copyright © 2009, 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 in the slide illustrates that the values of 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 is more than one, the example qualifies as a multiple-column subquery.

# Column Comparisons

Multiple-column comparisons involving subqueries can be:

- Nonpairwise comparisons
- Pairwise comparisons



## Pairwise Versus Nonpairwise Comparisons

Multiple-column comparisons involving subqueries can be nonpairwise comparisons or pairwise comparisons. If you consider the example “Display the details of the employees who work in the same department, and have the same manager, as ‘Daniel’? ,” you get the correct result with the following statement:

```
SELECT first_name, last_name, manager_id, department_id
  FROM empl_demo
 WHERE manager_id IN (SELECT manager_id
                        FROM empl_demo
                       WHERE first_name = 'Daniel')
   AND department_id IN (SELECT department_id
                        FROM empl_demo
                       WHERE first_name = 'Daniel');
```

There is only one “Daniel” in the EMPL\_DEMO table (Daniel Faviet, who is managed by employee 108 and works in department 100). However, if the subqueries return more than one row, the result might not be correct. For example, if you run the same query but substitute “John” for “Daniel,” you get an incorrect result. This is because the combination of department\_id and manager\_id is important. To get the correct result for this query, you need 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 employees with the first name of “John.”

```
SELECT employee_id, manager_id, department_id
  FROM empl_demo
 WHERE (manager_id, department_id) IN
       (SELECT manager_id, department_id
        FROM empl_demo
        WHERE first_name = 'John')
  AND first_name <> 'John';
```

ORACLE

6 - 4

Copyright © 2009, Oracle. All rights reserved.

## Pairwise Comparison Subquery

The example in the slide compares the combination of values in the MANAGER\_ID column and the DEPARTMENT\_ID column of each row in the EMPL\_DEMO table with the values in the MANAGER\_ID column and the DEPARTMENT\_ID column for the employees with the FIRST\_NAME of “John.” First, the subquery to retrieve the MANAGER\_ID and DEPARTMENT\_ID values for the employees with the FIRST\_NAME of “John” is executed. This subquery returns the following:

	MANAGER_ID	DEPARTMENT_ID
1	108	100
2	123	50
3	100	80

## Nonpairwise Comparison Subquery

Display the details of the employees who are managed by the same manager as the employees with the first name of “John” and work in the same department as the employees with the first name of “John.”

```
SELECT employee_id, manager_id, department_id
FROM   empl_demo
WHERE  manager_id IN
       (SELECT manager_id
        FROM empl_demo
        WHERE first_name = 'John')
AND department_id IN
       (SELECT department_id
        FROM empl_demo
        WHERE first_name = 'John')
AND first_name <> 'John';
```

ORACLE

6 - 6

Copyright © 2009, Oracle. All rights reserved.

## Nonpairwise Comparison Subquery

The example shows a nonpairwise comparison of the columns. First, the subquery to retrieve the MANAGER\_ID values for the employees with the FIRST\_NAME of “John” is executed. Similarly, the second subquery to retrieve the DEPARTMENT\_ID values for the employees with the FIRST\_NAME of “John” is executed. The retrieved values of the MANAGER\_ID and DEPARTMENT\_ID columns are compared with the MANAGER\_ID and DEPARTMENT\_ID columns for each row in the EMPL\_DEMO table. If the MANAGER\_ID column of the row in the EMPL\_DEMO table matches with any of the values of MANAGER\_ID retrieved by the inner subquery and if the DEPARTMENT\_ID column of the row in the EMPL\_DEMO table matches with any of the values of DEPARTMENT\_ID retrieved by the second subquery, the record is displayed.

# 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:
  - 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

ORACLE®

6 - 7

Copyright © 2009, 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 data manipulation language (DML) statements
- As the basis of a function-based index
- In GROUP BY clauses, CHECK constraints, and WHEN conditions
- In CONNECT BY clauses

## 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 the 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 © 2009, Oracle. All rights reserved.

## Scalar Subqueries: Examples

The first example in 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.

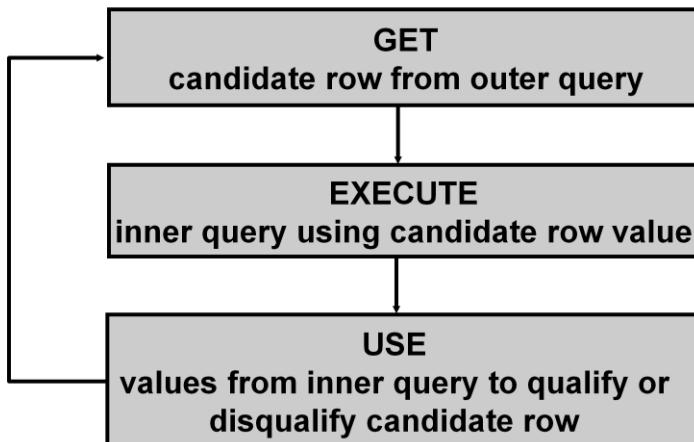
The following is the result of the first example in the slide:

...

	EMPLOYEE_ID	LAST_NAME	LOCATION
1	198	OConnell	USA
2	199	Grant	USA
3	200	Whalen	USA
4	201	Hartstein	Canada
5	202	Fay	Canada
6	203	Mavris	USA

# 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 © 2009, 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. That is, 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_table
WHERE  column1 operator
       (SELECT column1, column2
        FROM   table2
        WHERE  expr1 =
               Outer_table.expr2) ;
```

ORACLE

6 - 11

Copyright © 2009, 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. That is, 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_table  
WHERE  salary >  
       (SELECT AVG(salary)  
        FROM   employees inner_table  
        WHERE inner_table.department_id =  
              outer_table.department_id);
```

Each time a row from  
the outer query  
is processed, the  
inner query is  
evaluated.

ORACLE

6 - 12

Copyright © 2009, Oracle. All rights reserved.

## Using Correlated Subqueries

The example in 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. The alias makes the entire SELECT statement more readable. 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
1	200	Whalen	AD_ASST
2	101	Kochhar	AD_VP
3	176	Taylor	SA_REP

ORACLE

6 - 13

Copyright © 2009, Oracle. All rights reserved.

## Using Correlated Subqueries (continued)

The example in 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 in 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 in the slide, the COUNT (\*) group function 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.

# 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 © 2009, Oracle. All rights reserved.

## 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.

# Using the EXISTS Operator

```
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
1	201	Hartstein	MK_MAN	20
2	205	Higgins	AC_MGR	110
3	100	King	AD_PRES	90
4	101	Kochhar	AD_VP	90
5	102	De Haan	AD_VP	90
6	103	Hunold	IT_PROG	60
7	108	Greenberg	FI_MGR	100
8	114	Raphaely	PU_MAN	30

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
1	120	Treasury
2	130	Corporate Tax
3	140	Control And Credit
4	150	Shareholder Services
5	160	Benefits
6	170	Manufacturing
7	180	Construction

...

All Rows Fetched: 16

ORACLE

6 - 16

Copyright © 2009, 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);
```

All Rows Fetched: 0

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 © 2009, 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 emp16
ADD (department_name VARCHAR2(25));
```

```
UPDATE emp16 e
SET department_name =
    (SELECT department_name
     FROM departments d
     WHERE e.department_id = d.department_id);
```

ORACLE

6 - 18

Copyright © 2009, Oracle. All rights reserved.

## Correlated UPDATE (continued)

The example in 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 emp16
SET salary = (SELECT empl6.salary + rewards.pay_raise
              FROM rewards
              WHERE employee_id =
                    empl6.employee_id
              AND payraise_date =
                  (SELECT MAX(payraise_date)
                   FROM rewards
                   WHERE employee_id = empl6.employee_id))
WHERE empl6.employee_id
IN (SELECT employee_id FROM rewards);
```

## 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 © 2009, 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, 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 emp16 E
WHERE employee_id =
      (SELECT employee_id
       FROM   emp_history
       WHERE  employee_id = E.employee_id);
```

ORACLE

6 - 21

Copyright © 2009, 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 in the slide.

## 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 may improve performance.

ORACLE

6 - 22

Copyright © 2009, Oracle. All rights reserved.

### 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 © 2009, Oracle. All rights reserved.

## **WITH Clause: Example**

The problem in 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, 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 © 2009, Oracle. All rights reserved.

## WITH Clause: Example (continued)

The SQL code in 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 inline 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 in the slide is as follows:

	DEPARTMENT_NAME	DEPT_TOTAL
1	Sales	304500
2	Shipping	156400

## 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).



# **Regular Expression Support**

**ORACLE®**

Copyright © 2009, Oracle. All rights reserved.

# What Are Regular Expressions?

- You use regular expressions to search for (and manipulate) simple and complex patterns in string data by using standard syntax conventions.
- You use a set of SQL functions and conditions to search for and manipulate strings in SQL and PL/SQL.
- You specify a regular expression by using:
  - Metacharacters, which are operators that specify the search algorithms
  - Literals, which are the characters for which you are searching

ORACLE

7 - 2

Copyright © 2009, Oracle. All rights reserved.

## What Are Regular Expressions?

Oracle Database provides 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, such as finding the word "San Francisco" in a specified text, to the complex task of extracting all URLs from the text and the more complex task of finding 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 involve complex programming.

## Benefits of Using Regular Expressions

Regular expressions enable you to implement complex match logic in the database with the following benefits:

- By centralizing match logic in Oracle Database, you avoid intensive string processing of SQL results sets by middle-tier applications.
- Using server-side regular expressions to enforce constraints, you eliminate the need to code data validation logic on the client.
- The built-in SQL and PL/SQL regular expression functions and conditions make string manipulations more powerful and easier than in previous releases of Oracle Database 11g.

ORACLE

7 - 3

Copyright © 2009, Oracle. All rights reserved.

## Benefits of Using Regular Expressions

Regular expressions are a powerful text-processing component of programming languages such as PERL and Java. For example, a PERL script can process each HTML file in a directory, read its contents into a scalar variable as a single string, and then use regular expressions to search for URLs in the string. One reason for many developers writing in PERL is that it has a robust pattern-matching functionality. Oracle's support of regular expressions enables developers to implement complex match logic in the database. This technique is useful for the following reasons:

- By centralizing match logic in Oracle Database, you avoid intensive string processing of SQL results sets by middle-tier applications. The SQL regular expression functions move the processing logic closer to the data, thereby providing a more efficient solution.
- Before Oracle Database 10g, developers often coded data validation logic on the client, requiring the same validation logic to be duplicated for multiple clients. Using server-side regular expressions to enforce constraints solves this problem.
- The built-in SQL and PL/SQL regular expression functions and conditions make string manipulations more powerful and less cumbersome than in previous releases of Oracle Database 10g.

# Using the Regular Expressions Functions and Conditions in SQL and PL/SQL

Function or Condition Name	Description
REGEXP_LIKE	Is similar to the <code>LIKE</code> operator, but performs regular expression matching instead of simple pattern matching (condition)
REGEXP_REPLACE	Searches for a regular expression pattern and replaces it with a replacement string
REGEXP_INSTR	Searches a string for a regular expression pattern and returns the position where the match is found
REGEXP_SUBSTR	Searches for a regular expression pattern within a given string and extracts the matched substring
REGEXP_COUNT	Returns the number of times a pattern match is found in an input string

ORACLE

## Using the Regular Expressions Functions and Conditions in SQL and PL/SQL

Oracle Database provides a set of SQL functions that you use to search and manipulate strings by using regular expressions. You use these functions on a text literal, bind variable, or any column that holds character data such as CHAR, NCHAR, CLOB, NCLOB, NVARCHAR2, and VARCHAR2 (but not LONG). A regular expression must be enclosed within single quotation marks. This ensures that the entire expression is interpreted by the SQL function and can improve the readability of your code.

- REGEXP\_LIKE: This condition searches a character column for a pattern. Use this condition in the `WHERE` clause of a query to return rows matching the regular expression that you specify.
- REGEXP\_REPLACE: This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern that you specify.
- REGEXP\_INSTR: This function searches a string for a given occurrence of a regular expression pattern.
- REGEXP\_SUBSTR: This function returns the actual substring matching the regular expression pattern that you specify.
- REGEXP\_COUNT: This function returns the number of times a pattern match is found in the input string.

## What Are Metacharacters?

- Metacharacters are special characters that have a special meaning such as a wildcard, a repeating character, a nonmatching character, or a range of characters.
- You can use several predefined metacharacter symbols in the pattern matching.
- For example, the `^ (f | ht) tps? : $` regular expression searches for the following from the beginning of the string:
  - The literals `f` or `ht`
  - The `t` literal
  - The `p` literal, optionally followed by the `s` literal
  - The colon “`:`” literal at the end of the string

ORACLE®

## What Are Metacharacters?

The regular expression in the slide matches the `http:`, `https:`, `ftp:`, and `ftps:` strings.

**Note:** For a complete list of the regular expressions’ metacharacters, see the *Oracle Database Advanced Application Developer’s Guide 11g Release 2*.

# Using Metacharacters with Regular Expressions

Syntax	Description
.	Matches any character in the supported character set, except NULL
+	Matches one or more occurrences
?	Matches zero or one occurrence
*	Matches zero or more occurrences of the preceding subexpression
{m}	Matches exactly <i>m</i> occurrences of the preceding expression
{m, }	Matches at least <i>m</i> occurrences of the preceding subexpression
{m, n}	Matches at least <i>m</i> , but not more than <i>n</i> , occurrences of the preceding subexpression
[ ... ]	Matches any single character in the list within the brackets
	Matches one of the alternatives
( . . . )	Treats the enclosed expression within the parentheses as a unit. The subexpression can be a string of literals or a complex expression containing operators.

ORACLE

## Using Metacharacters in Regular Expressions Functions

**Any character, “ . ”:** `a.b` matches the strings `abb`, `acb`, and `adb`, but not `acc`.

**One or more, “ + ”:** `a+` matches the strings `a`, `aa`, and `aaa`, but does not match `bbb`.

**Zero or one, “ ? ”:** `ab?c` matches the strings `abc` and `ac`, but does not match `abbc`.

**Zero or more, “ \* ”:** `ab*c` matches the strings `ac`, `abc`, and `abbc`, but does not match `abb`.

**Exact count, “ {m} ”:** `a{3}` matches the strings `aaa`, but does not match `aa`.

**At least count, “ {m,} ”:** `a{3,}` matches the strings `aaa` and `aaaa`, but not `aa`.

**Between count, “ {m,n} ”:** `a{3,5}` matches the strings `aaa`, `aaaa`, and `aaaaaa`, but not `aa`.

**Matching character list, “ [...] ”:** `[abc]` matches the first character in the strings `all`, `bill`, and `cold`, but does not match any characters in `doll`.

**Or, “ | ”:** `a|b` matches character `a` or character `b`.

**Subexpression, “ (...) ”:** `(abc)?def` matches the optional string `abc`, followed by `def`. The expression matches `abcdefghi` and `def`, but does not match `ghi`. The subexpression can be a string of literals or a complex expression containing operators.

# Using Metacharacters with Regular Expressions

Syntax	Description
^	Matches the beginning of a string
\$	Matches the end of a string
\	Treats the subsequent metacharacter in the expression as a literal
\n	Matches the <i>n</i> th (1–9) preceding subexpression of whatever is grouped within parentheses. The parentheses cause an expression to be remembered; a backreference refers to it.
\d	A digit character
[ <b>:class:</b> ]	Matches any character belonging to the specified POSIX character class
[ <b>^:class:</b> ]	Matches any single character <i>not</i> in the list within the brackets

ORACLE

## Using Metacharacters in Regular Expressions Functions (continued)

**Beginning/end of line anchor, “ ^ ” and “\$”:** ^def matches def in the string defghi but does not match def in abcdef. def\$ matches def in the string abcdef but does not match def in the string defghi.

**Escape character “ \ ”:** \+ searches for a +. It matches the plus character in the string abc+def, but does not match Abcdef.

**Backreference, “ \n ”:** (abc|def) xy\1 matches the strings abcxyabc and defxydef, but does not match abcxydef or abcxy. A backreference enables you to search for a repeated string without knowing the actual string ahead of time. For example, the expression ^(.\*)\1\$ matches a line consisting of two adjacent instances of the same string.

**Digit character, “ \d ”:** The expression ^[\d{3}]\ \d{3}-\d{4}\\$ matches [650] 555-1212 but does not match 650-555-1212.

**Character class, “ [**:class:**] ”:** [[:upper:]]+ searches for one or more consecutive uppercase characters. This matches DEF in the string abcDEFghi but does not match the string abcdefghi.

**Nonmatching character list (or class), “ [**^...:**] ”:** [^abc] matches the character d in the string abcdef, but not a, b, or c.

# Regular Expressions Functions and Conditions: Syntax

```
REGEXP_LIKE    (source_char, pattern [,match_option])
```

```
REGEXP_INSTR   (source_char, pattern [, position  
[, occurrence [, return_option  
[, match_option [, subexpr]]]])
```

```
REGEXP_SUBSTR  (source_char, pattern [, position  
[, occurrence [, match_option  
[, subexpr]]]])
```

```
REGEXP_REPLACE (source_char, pattern [,replacestr  
[, position [, occurrence  
[, match_option]]]])
```

```
REGEXP_COUNT   (source_char, pattern [, position  
[, occurrence [, match_option]]])
```

ORACLE

7 - 8

Copyright © 2009, Oracle. All rights reserved.

## Regular Expressions Functions and Conditions: Syntax

The syntax for the regular expressions functions and conditions is as follows:

- **source\_char**: A character expression that serves as the search value
- **pattern**: A regular expression, a text literal
- **occurrence**: A positive integer indicating which occurrence of pattern in source\_char Oracle Server should search for. The default is 1.
- **position**: A positive integer indicating the character of source\_char where Oracle Server should begin the search. The default is 1.
- **return\_option**:
  - 0: Returns the position of the first character of the occurrence (default)
  - 1: Returns the position of the character following the occurrence
- **Replacestr**: Character string replacing pattern
- **match\_parameter**:
  - “c”: Uses case-sensitive matching (default)
  - “i”: Uses non-case-sensitive matching
  - “n”: Allows match-any-character operator
  - “m”: Treats source string as multiple lines
- **subexpr**: Fragment of pattern enclosed in parentheses. You learn more about subexpressions later in this lesson.

# Performing a Basic Search by Using the REGEXP\_LIKE Condition

```
REGEXP_LIKE(source_char, pattern [, match_parameter ])
```

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

	FIRST_NAME	LAST_NAME
1	Steven	King
2	Steven	Markle
3	Stephen	Stiles

ORACLE

## Performing a Basic Search by Using the REGEXP\_LIKE Condition

REGEXP\_LIKE is similar to the LIKE condition, except that REGEXP\_LIKE performs regular expression matching instead of the simple pattern matching performed by LIKE. This condition evaluates strings by using characters as defined by the input character set.

### 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 expression
- \$ indicates the end of the expression
- | indicates either/or

# Replacing Patterns by Using the REGEXP\_REPLACE Function

```
REGEXP_REPLACE(source_char, pattern [,replacestr  
[, position [, occurrence [, match_option]]]])
```

```
SELECT REGEXP_REPLACE(phone_number, '\.', '-') AS phone  
FROM employees;
```

Original

	LAST_NAME	PHONE
1	OConnell	650.507.9833
2	Grant	650.507.9844
3	Whalen	515.123.4444
4	Hartstein	515.123.5555

Partial results

	LAST_NAME	PHONE
1	OConnell	650-507-9833
2	Grant	650-507-9844
3	Whalen	515-123-4444
4	Hartstein	515-123-5555

ORACLE

## Replacing Patterns by Using the REGEXP\_REPLACE Function

Using the REGEXP\_REPLACE function, you reformat the phone number to replace the period (.) delimiter with a dash (-) delimiter. Here is an explanation of each of the elements used in the regular expression example:

- phone\_number is the source column.
- '\.' is the search pattern.
  - Use single quotation marks (' ') to search for the literal character period (.).
  - Use a backslash (\) to search for a character that is normally treated as a metacharacter.
- '-' is the replace string.

## Finding Patterns by Using the REGEXP\_INSTR Function

```
REGEXP_INSTR  (source_char, pattern [, position [, occurrence [, return_option [, match_option]]]])
```

```
SELECT street_address,
REGEXP_INSTR(street_address,'[:alpha:]') AS
First_Alpha_Position
FROM locations;
```

STREET_ADDRESS	FIRST_ALPHA_POSITION
1 1297 Via Cola di Rie	6
2 93091 Calle della Testa	7
3 2017 Shinjuku-ku	6
4 9450 Kamiya-cho	6

ORACLE

7 - 11

Copyright © 2009, Oracle. All rights reserved.

### Finding Patterns by Using the REGEXP\_INSTR Function

In this example, the REGEXP\_INSTR function is used to search the street address to find the location of the first alphabetic character, regardless of whether it is in uppercase or lowercase. Note that [:<class>:] implies a character class and matches any character from within that class; [:alpha:] matches with any alphabetic character. The partial results are displayed.

In the expression used in the query '[:alpha:]':

- [ starts the expression
- [:alpha:] indicates alphabetic 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:] + .

## Extracting Substrings by Using the REGEXP\_SUBSTR Function

```
REGEXP_SUBSTR (source_char, pattern [, position  
[, occurrence [, match_option]]])
```

```
SELECT REGEXP_SUBSTR(street_address , ' [^ ]+' ) AS Road  
FROM locations;
```

ROAD
1 Via
2 Calle
3 (null)
4 (null)
5 Jabberwocky

ORACLE

## Extracting Substrings by Using the REGEXP\_SUBSTR Function

In this example, the road names are extracted from the LOCATIONS table. To do this, the contents in the STREET\_ADDRESS column that are after the first space are returned by using the REGEXP\_SUBSTR function. In the expression used in the query ' [^ ]+' :

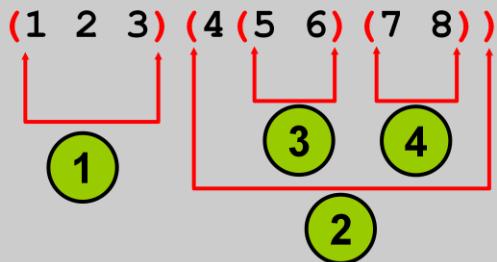
- [ starts the expression
- ^ indicates NOT
- indicates space
- ] ends the expression
- + indicates 1 or more
- indicates space

# Subexpressions

Examine this expression:

```
(1 2 3) (4(5 6) (7 8))
```

The subexpressions are:



ORACLE

7 - 13

Copyright © 2009, Oracle. All rights reserved.

## Subexpressions

Oracle Database 11g provides regular expression support parameter to access a subexpression. In the slide example, a string of digits is shown. The parentheses identify the subexpressions within the string of digits. Reading from left to right, and from outer parentheses to the inner parentheses, the subexpressions in the string of digits are:

1. 123
2. 45678
3. 56
4. 78

You can search for any of those subexpressions with the REGEXP\_INSTR and REGEXP\_SUBSTR functions.

# Using Subexpressions with Regular Expression Support

```
SELECT
  REGEXP_INSTR
  (1) ('0123456789',           -- source char or search value
  (2) '(123)(4(56)(78))',    -- regular expression patterns
  (3) 1,                      -- position to start searching
  (4) 1,                      -- occurrence
  (5) 0,                      -- return option
  (6) 'i',                    -- match option (case insensitive)
  (7) 1)                     -- sub-expression on which to search
    "Position"
  FROM dual;
```

Position
1
2

ORACLE

7 - 14

Copyright © 2009, Oracle. All rights reserved.

## Using Subexpressions with Regular Expression Support

REGEXP\_INSTR and REGEXP\_SUBSTR have an optional SUBEXPR parameter that lets you target a particular substring of the regular expression being evaluated.

In the example shown in the slide, you may want to search for the first subexpression pattern in your list of subexpressions. The example shown identifies several parameters for the REGEXP\_INSTR function.

1. The string you are searching is identified.
2. The subexpressions are identified. The first subexpression is 123. The second subexpression is 45678, the third is 56, and the fourth is 78.
3. The third parameter identifies from which position to start searching.
4. The fourth parameter identifies the occurrence of the pattern you want to find. 1 means find the first occurrence.
5. The fifth parameter is the return option. This is the position of the first character of the occurrence. (If you specify 1, the position of the character following the occurrence is returned.)
6. The sixth parameter identifies whether your search should be case-sensitive or not.
7. The last parameter is the parameter added in Oracle Database 11g. This parameter specifies which subexpression you want to find. In the example shown, you are searching for the first subexpression, which is 123.

## Why Access the *n*th Subexpression?

- A more realistic use: DNA sequencing
- You may need to find a specific subpattern that identifies a protein needed for immunity in mouse DNA.

```
SELECT
  REGEXP_INSTR('ccaccttcctccactcctcacgttctcacctgtaaagcgccctc
cctcatccccatgcccccttaccctgcaggtagagtagggtagaaaccagagagctccaagc
tccatctgtggagaggtgccatcctggctcagagagagaggagaattgccccaaagctgcc
tgcaagcttaccacccttagtctcacaaagccttgagttcatagcattttca
ccctgcccagcaggacactgcagcacccaaaggcttccaggagtagggttgcctcaagag
gctcttgggtctgtatggccacatccttggattttcaagttgtatggtcacagccctgaggc
atgttagggcgtgggatgcgtctgtctcctcctgaaccctgaaccctgtggc
taccccaagagcacttagagccag',
  '(gtc(tcac)(aaag))',
  1, 1, 0, 'i',
  1) "Position"
FROM dual;
```

Position
1 195

ORACLE

7 - 15

Copyright © 2009, Oracle. All rights reserved.

## Why Access the *n*th Subexpression?

In life sciences, you may need to extract the offsets of subexpression matches from a DNA sequence for further processing. For example, you may need to find a specific protein sequence, such as the begin offset for the DNA sequence preceded by gtc and followed by tcac followed by aaag. To accomplish this goal, you can use the REGEXP\_INSTR function, which returns the position where a match is found.

In the slide example, the position of the first subexpression (gtc) is returned. gtc appears starting in position 195 of the DNA string.

If you modify the slide example to search for the second subexpression (tcac), the query results in the following output. tcac appears starting in position 198 of the DNA string.

Position
1 198

If you modify the slide example to search for the third subexpression (aaag), the query results in the following output. aaag appears starting in position 202 of the DNA string.

Position
1 202

## REGEXP\_SUBSTR: Example

```
SELECT
  REGEXP_SUBSTR
  (1)('acgctgcactgca', -- source char or search value
  2) 'acg(.*)gca',    -- regular expression pattern
  3) 1,                -- position to start searching
  4) 1,                -- occurrence
  5) 'i',              -- match option (case insensitive)
  6) 1)                -- sub-expression
  "Value"
FROM dual;
```

Value
1 ctgcact

ORACLE

7 - 16

Copyright © 2009, Oracle. All rights reserved.

## REGEXP\_SUBSTR: Example

In the example shown in the slide:

1. acgctgcactgca is the source to be searched
2. acg (.\*) gca is the pattern to be searched. Find acg followed by gca with potential characters between the acg and the gca.
3. Start searching at the first character of the source
4. Search for the first occurrence of the pattern
5. Use non-case-sensitive matching on the source
6. Use a nonnegative integer value that identifies the *n*th subexpression to be targeted. This is the subexpression parameter. In this example, 1 indicates the first subexpression. You can use a value from 0–9. A zero means that no subexpression is targeted. The default value for this parameter is 0.

## Using the REGEXP\_COUNT Function

```
REGEXP_COUNT (source_char, pattern [, position  
[, occurrence [, match_option]]])
```

```
SELECT REGEXP_COUNT(  
  'ccaccccccactcctcaegtttcacctgtaaagcgccccatgcccccttaccctgcag  
  ggttagagtggctagaaaccagagagctccaagctccatctgtggagaggtgcaccccttggctgcagagagaggag  
  aatttgcacccaaagctgcctgcagagcttcaccacccttagtctcacaaagcctttagtcatagcattttgagtt  
  ttcacccctgcccagcaggacactgcagcacccaaagggtttccaggagtaggggtgcctcaagaggtcttgggtc  
  tgatggccacatcctgaaattttcaagttgtatggtcacagccctgaggcatgttagggcgtgggatgcgctctg  
  ctctgcctctctctgaaccctgtggctaccctggactagccagagcacttagagccag',  
  'gtc') AS Count  
  
FROM dual;
```

	COUNT
1	4

ORACLE

7 - 17

Copyright © 2009, Oracle. All rights reserved.

## Using the REGEXP\_COUNT Function

The REGEXP\_COUNT function evaluates strings by using characters as defined by the input character set. It returns an integer indicating the number of occurrences of pattern. If no match is found, the function returns 0.

In the slide example, the number of occurrences for a DNA substring is determined by using the REGEXP\_COUNT function.

The following example shows that the number of times the pattern 123 occurs in the string 123123123123 is three times. The search starts from the second position of the string.

```
SELECT REGEXP_COUNT  
  ('123123123123', -- source char or search value  
   '123',           -- regular expression pattern  
   2,               -- position where the search should start  
   'i')             -- match option (case insensitive)  
  As Count  
FROM dual;
```

	COUNT
1	3