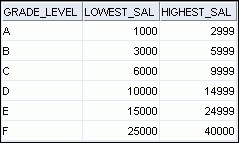
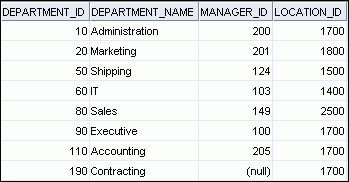
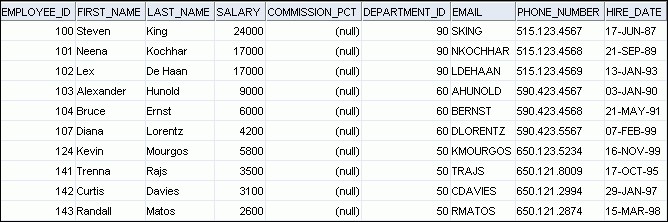
## Tables used in this handout

**EMPLOYEES**



**DEPARTMENTS JOB\_GRADES**

## Retrieving Data Using the SQL SELECT Statement Objectives

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

* List the capabilities of SQL SELECT statements
* Execute a basic SELECT statement

## Lesson Agenda

* Basic SELECT statement
* Arithmetic expressions and NULL values in the SELECT statement
* Column aliases
* Use of concatenation operator, literal character strings, alternative quote operator, and the DISTINCT keyword
* DESCRIBE command

## Capabilities of SQL SELECT Statements

**Projection Selection**

|  |  |  |
| --- | --- | --- |
| **Table 1** | **Table 1** | |
| **Join** | | | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | | |

**Table 1 Table 2**

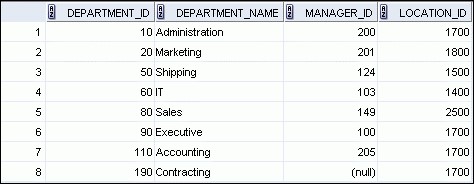
## Basic SELECT Statement

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

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

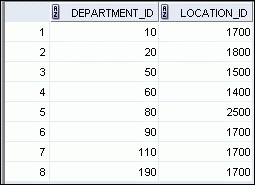
## Selecting All Columns

|  |
| --- |
| **SELECT \***  **FROM departments;** |



## Selecting Specific Columns

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  | **SELECT department\_id, location\_id** |  |   **FROM departments;** |



## 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 optionally be 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 (;).

## Column Heading Defaults

* SQL Developer:
  + Default heading alignment: Left-aligned
  + Default heading display: Uppercase

## Lesson Agenda

* Basic SELECT statement
* Arithmetic expressions and NULL values in the SELECT statement
* Column Aliases
* Use of concatenation operator, literal character strings, alternative quote operator, and the DISTINCT keyword
* DESCRIBE command

## Arithmetic Expressions

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

|  |  |
| --- | --- |
| Operator | Description |
| + | Add |
| - | Subtract |
| \* | Multiply |
| / | Divide |

## Using Arithmetic Operators

**SELECT last\_name, salary, salary + 300**

**FROM employees;**



**…**

## Operator Precedence

|  |  |
| --- | --- |
| **SELECT last\_name, salary, 12\*salary+100 FROM employees;** | **1** |



**…**

|  |  |
| --- | --- |
| **SELECT last\_name, salary, 12\*(salary+100) FROM employees;** | **2** |



**…**

## 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;**



**…**



**…**



## Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

**SELECT last\_name, 12\*salary\*commission\_pct**

**FROM employees;**

**…**

**…**





## Lesson Agenda

* Basic SELECT statement
* Arithmetic expressions and NULL values in the SELECT statement
* Column aliases
* Use of concatenation operator, literal character strings, alternative quote operator, and the DISTINCT keyword
* DESCRIBE command

## 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 alias.)
* Requires double quotation marks if it contains spaces or special characters, or if it is case-sensitive

## Using Column Aliases

**SELECT last\_name AS name, commission\_pct comm**

**FROM employees;**



**…**

**SELECT last\_name "Name" , salary\*12 "Annual Salary"**

**FROM employees;**



**…**

## Lesson Agenda

* Basic SELECT Statement
* Arithmetic Expressions and NULL values in SELECT statement
* Column Aliases
* Use of concatenation operator, literal character strings, alternative quote operator, and the DISTINCT keyword
* DESCRIBE command

## 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;**



**…**

## 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.

## Using Literal Character Strings

**SELECT last\_name ||' is a '||job\_id AS "Employee Details"**

**FROM employees;**



**…**



## Alternative Quote (q) Operator

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

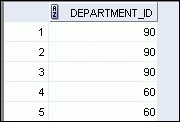
|  |
| --- |
| **SELECT department\_name || q'[ Department's Manager Id: ]'**  **|| manager\_id**  **AS "Department and Manager" FROM departments;** |



## Duplicate Rows

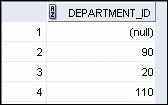
The default display of queries is all rows, including duplicate rows.

|  |  |
| --- | --- |
| **SELECT department\_id FROM employees;** | **1** |



**…**

|  |  |
| --- | --- |
| **SELECT DISTINCT department\_id FROM employees;** | **2** |



**…**

## Lesson Agenda

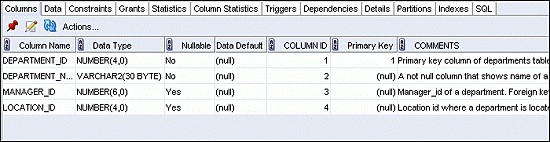
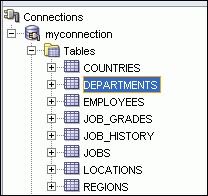
* Basic SELECT statement
* Arithmetic expressions and NULL values in the SELECT statement
* Column aliases
* Use of concatenation operator, literal character strings, alternative quote operator, and the DISTINCT keyword
* DESCRIBE command

## 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.

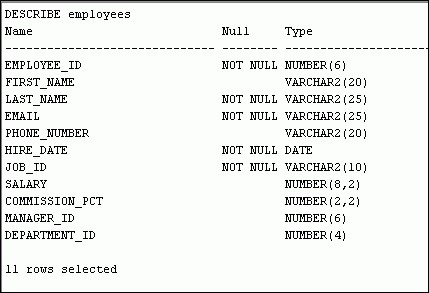
**DESC[RIBE]**

***tablename***



## Using the DESCRIBE Command

**DESCRIBE employees**



**Restricting and Sorting Data**

## Objectives

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

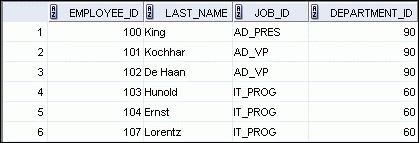
* Limit the rows that are retrieved by a query
* Sort the rows that are retrieved by a query
* Use ampersand substitution to restrict and sort output at run time

## Lesson Agenda

* Limiting rows with:
  + The WHERE clause
  + The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL conditions
  + Logical conditions using AND, OR, and NOT operators
* Rules of precedence for operators in an expression
* Sorting rows using the ORDERBY clause
* Substitution variables
* DEFINE and VERIFY commands

## Limiting Rows Using a Selection

### EMPLOYEES



**…**

**“retrieve all**

**employees in**

**department 90”**



## 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.

## Using the WHERE Clause

**SELECT employee\_id, last\_name, job\_id, department\_id**

**FROM employees**

**WHERE department\_id = 90 ;**



## 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' ;**

## 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 |

## Using Comparison Operators

**SELECT last\_name, salary**

**FROM employees**

**WHERE salary <= 3000 ;**



## 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**



## 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) ;**



## 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%' ;**

## 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%' ;**



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

## Using the NULL Conditions

Test for nulls with the ISNULL operator.

**SELECT last\_name, manager\_id**

**FROM employees**

**WHERE manager\_id IS NULL ;**



## Defining Conditions Using the Logical Operators

|  |  |
| --- | --- |
| Operator | Meaning |
| AND | Returns TRUE if *both* component conditions are true |
| OR | Returns TRUE if *either* component condition is true |
| NOT | Returns TRUE if the condition is false |

## Using the AND Operator

AND requires both the component conditions to be true:

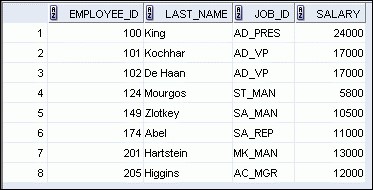
|  |
| --- |
| **SELECT employee\_id, last\_name, job\_id, salary**  **FROM employees**  **WHERE salary >= 10000**  **AND job\_id LIKE '%MAN%' ;** |



## 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%** | **' ;** |



## Using the NOT Operator

|  |  |  |
| --- | --- | --- |
| **SELECT last\_name, job\_id FROM employees** | | |
| **WHERE** | **job\_id**  **NOT IN ('IT\_PROG', 'ST\_CLERK', 'SA\_REP'** | **;**  **)** |



## Lesson Agenda

* Limiting rows with:
  + The WHERE clause
  + The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL operators
  + Logical conditions using AND, OR, and NOT operators
* Rules of precedence for operators in an expression
* Sorting rows using the ORDERBY clause
* Substitution variables
* DEFINE and VERIFY commands

## 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.**

## Rules of Precedence

**SELECT last\_name, job\_id, salary**

**FROM employees**

**WHERE job\_id = 'SA\_REP'**

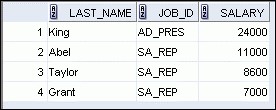
**OR**

**job\_id = 'AD\_PRES'**

**AND**

**salary > 15000;**

**1**



|  |  |  |
| --- | --- | --- |
| **SELECT last\_name, job\_id, salary**  **FROM employees**  **WHERE (job\_id = 'SA\_REP'** | | **2** |
| **OR AND** | **job\_id = 'AD\_PRES') salary > 15000;** |



## Lesson Agenda

* Limiting rows with:
  + The WHERE clause
  + The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL operators
  + Logical conditions using AND, OR, and NOT operators
* Rules of precedence for operators in an expression
* Sorting rows using the ORDERBY clause
* Substitution variables
* DEFINE and VERIFY commands

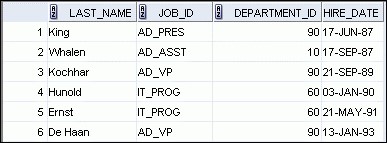
## Using the ORDERBY Clause

* Sort retrieved rows with the ORDERBY clause:
  + ASC: Ascending order, default
  + DESC: Descending order
* The ORDERBY clause comes last in the SELECT statement:

**SELECT last\_name, job\_id, department\_id, hire\_date**

**FROM employees**

**ORDER BY hire\_date ;**



**…**

## Sorting

* Sorting in descending order:

|  |
| --- |
| **SELECT last\_name, job\_id, department\_id, hire\_date**  **FROM employees**  **1**  **ORDER BY hire\_date DESC ;** |

* Sorting by column alias:

|  |  |
| --- | --- |
| **SELECT employee\_id, last\_name, salary\*12 annsal**  **FROM employees**  **ORDER BY annsal ;** | **2** |

## Sorting

* Sorting by using the column’s numeric position:

|  |
| --- |
| **SELECT last\_name, job\_id, department\_id, hire\_date**  **FROM employees**  **3**  **ORDER BY 3;** |

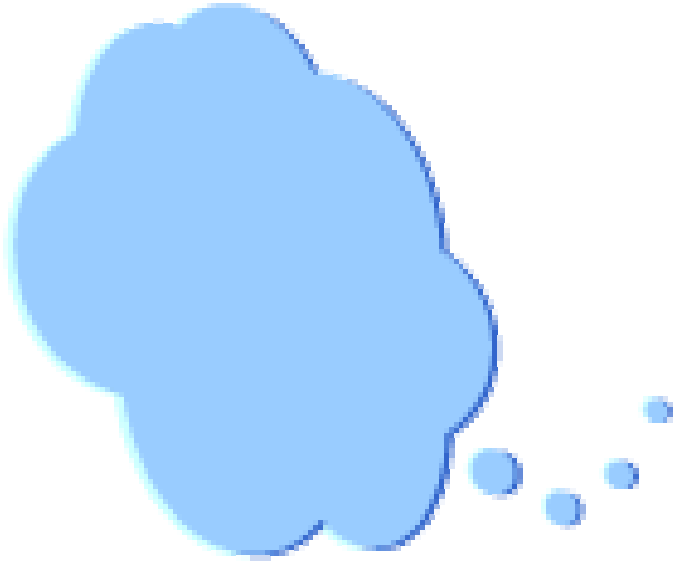
* Sorting by multiple columns:

|  |  |
| --- | --- |
| **SELECT last\_name, department\_id, salary**  **FROM employees**  **ORDER BY department\_id, salary DESC;** | **4** |

## Lesson Agenda

* Limiting rows with:
  + The WHERE clause
  + The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL operators
  + Logical conditions using AND, OR, and NOT operators
* Rules of precedence for operators in an expression
* Sorting rows using the ORDERBY clause
* Substitution variables
* DEFINE and VERIFY commands

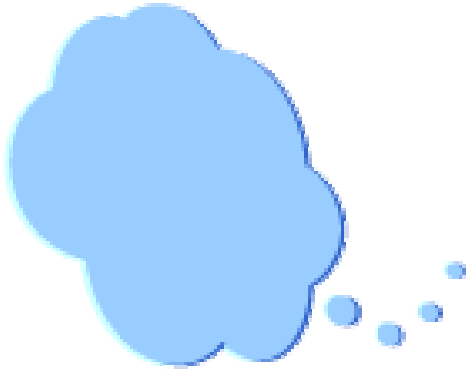
## Substitution Variables



**... salary = ? …**

**… department\_id = ? …**

**... last\_name = ? ...**

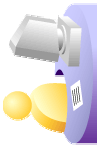


**I want**

**to query**

**different**

**values.**



## 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
  + ORDERBY clauses
  + Column expressions
  + Table names
  + Entire SELECT statements

## Using the Single-Ampersand Substitution Variable

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

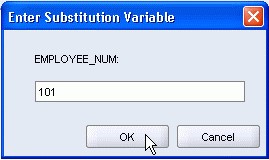
**SELECT employee\_id, last\_name, salary, department\_id**

**FROM employees**

**WHERE employee\_id = &employee\_num ;**



## Using the Single-Ampersand Substitution Variable





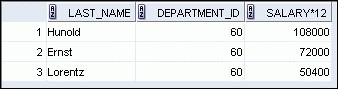
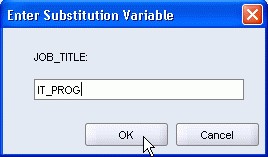
## Character and Date Values with Substitution Variables

Use single quotation marks for date and character values:

**SELECT last\_name, department\_id, salary\*12**

**FROM employees**

**WHERE job\_id = '&job\_title' ;**



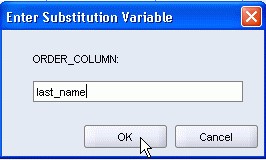
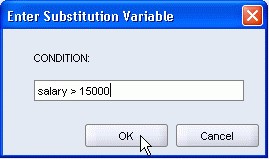
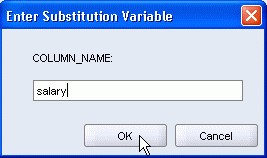
## Specifying Column Names, Expressions, and Text

**SELECT employee\_id, last\_name, job\_id,&column\_name**

**FROM employees**

**WHERE &condition**

**ORDER BY &order\_column ;**



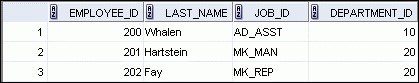
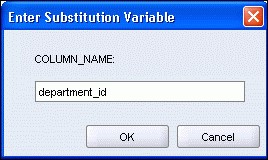
## Using the Double-Ampersand Substitution Variable

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 ;**



**…**

## Lesson Agenda

* Limiting rows with:
  + The WHERE clause
  + The comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL operators
  + Logical conditions using AND, OR, and NOT operators
* Rules of precedence for operators in an expression
* Sorting rows using the ORDERBY clause
* Substitution variables
* DEFINE and VERIFY commands

## 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**

## Using the VERIFY Command

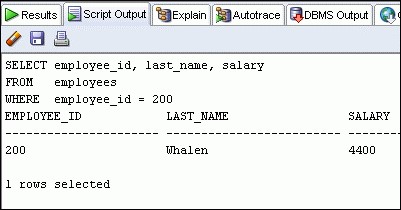
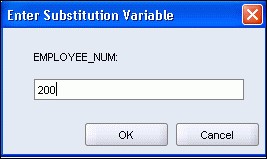
Use the VERIFY command to toggle the display of the substitution variable, both before and after SQL Developer replaces substitution variables with values:

**SET VERIFY ON**

**SELECT employee\_id, last\_name, salary**

**FROM employees**

**WHERE employee\_id = &employee\_num;**



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

## Objectives

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

* Describe various types of functions available in SQL
* Use character, number, and date functions in SELECT statements

## Lesson Agenda

* Single-row SQL functions
* Character functions
* Number functions
* Working with dates
* Date functions

## SQL Functions

**Function**

**Input**

**arg 1**

**arg 2**

**arg n**

**Function performs**

**action**

**Output**

**Result**

**value**

## Two Types of SQL Functions

**Single-row**

**functions**

**Multiple-row**

**functions**

**Return one result**

**per row**

**Return one result**

**per set of rows**

**Functions**

## 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,...*)]**

## Single-Row Functions

**Conversion**

**Character**

**Number**

**Date**

**General**

**Single-row**

**functions**

## Lesson Agenda

* Single-row SQL functions
* Character functions
* Number functions
* Working with dates
* Date functions

## Character Functions

**Character**

**functions**

**Case-co**

**nversion**

**func**

**tions**

**Character-manipulation**

**functions**

|  |  |
| --- | --- |
| **LOWER**  **UPPER**  **INITCAP** | **CONCAT**  **SUBSTR**  **LENGTH** |

**INSTR**

**LPAD | RPAD**

**TRIM**

**REPLACE**

## 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 |

## 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';** |

**SELECT employee\_id, last\_name, department\_id**

**FROM employees**

**WHERE LOWER(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 |

## 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'?"**

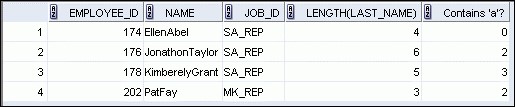
**FROM employees**

**WHERE SUBSTR(job\_id, 4) = 'REP';**

**2**

**1**

**3**



**3**

**1**

**2**

## Lesson Agenda

* Single-row SQL functions
* Character functions
* Number functions
* Working with dates
* Date Functions

## 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 |

## Using the ROUND Function



**SELECT ROUND(45.923,2), ROUND(45.923,0),**

**ROUND(45.923,-1)**

**FROM DUAL;**

**3**

**3**

**1**

**2**

**1**

**2**

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

## Using the TRUNC Function



**SELECT TRUNC(45.923,2), TRUNC(45.923),**

**TRUNC(45.923,-1)**

**FROM DUAL;**

**3**

**3**

**1**

**2**

**1**

**2**

## 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';**



## Lesson Agenda

* Single-row SQL functions
* Character functions
* Number functions
* Working with dates
* Date functions

## 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';**



## RR Date Format

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

**If the specified two-digit year is:**

**0–49 50–99**

**If two digits The return date is in The return date is in of the 0–49 the current century the century before**

**current the current one**

**year are: The return date is in The return date is in**

**50–99 the century after the current century the current one**

## Using the SYSDATE Function

SYSDATE is a function that returns:

* Date
* Time

**SELECT sysdate**

**FROM dual;**



## 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.

## Using Arithmetic Operators with Dates

**SELECT last\_name, (SYSDATE-hire\_date)/7 AS WEEKS**

**FROM employees**

**WHERE department\_id = 90;**



## Lesson Agenda

* Single-row SQL functions
* Character functions
* Number functions
* Working with dates
* Date functions

## 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 |

## 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' |

**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 |

## Using Conversion Functions and Conditional Expressions Objectives

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

* Describe various types of conversion functions that are available in SQL
* Use the TO\_CHAR, TO\_NUMBER, and TO\_DATE conversion functions
* Apply conditional expressions in a SELECT statement

## Lesson Agenda

* Implicit and explicit data type conversion
* TO\_CHAR, TO\_DATE, TO\_NUMBER functions
* Nesting functions
* General functions:
  + NVL
  + NVL2
  + NULLIF
  + COALESCE
* Conditional expressions:
  + CASE
  + DECODE

## Conversion Functions

**Implicit**

**data type**

**conversion**

**Explicit data type**

**conversion**

**Data type**

**conv**

**ersion**

## Implicit Data Type Conversion

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

|  |  |
| --- | --- |
| **From** | **To** |
| VARCHAR2 or CHAR | NUMBER |
| VARCHAR2 or CHAR | DATE |

## 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 |

## Explicit Data Type Conversion

**TO\_NUMBER TO\_DATE**

### NUMBER CHARACTER DATE

**TO\_CHAR TO\_CHAR**

## Explicit Data Type Conversion

**TO\_NUMBER TO\_DATE**

### NUMBER CHARACTER DATE

**TO\_CHAR TO\_CHAR**

## Lesson Agenda

* Implicit and explicit data type conversion
* TO\_CHAR, TO\_DATE, TO\_NUMBER functions
* Nesting functions
* General functions:
  + NVL
  + NVL2
  + NULLIF
  + COALESCE
* Conditional expressions:
  + CASE
  + DECODE

## 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

## 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 |

## 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** |

4 - 14

## Using the TO\_CHAR Function with Dates

|  |  |
| --- | --- |
| **SELECT last\_name,**   |  | | --- | | **TO\_CHAR(hire\_date, 'fmDD Month YYYY') AS HIREDATE** |   **FROM employees;** |



**…**



4 - 16

## 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 |

4 - 17

## Using the TO\_CHAR Function with Numbers

**SELECT TO\_CHAR(salary, '$99,999.00') SALARY**

**FROM employees**

**WHERE last\_name = 'Ernst';**



4 - 19

## 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.

4 - 20

## Using the TO\_CHAR and TO\_DATE Function with 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');**



## Lesson Agenda

* Implicit and explicit data type conversion
* TO\_CHAR, TO\_DATE, TO\_NUMBER functions
* Nesting functions
* General functions:
  + NVL
  + NVL2
  + NULLIF
  + COALESCE
* Conditional expressions:
  + CASE
  + DECODE

## 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**

**)**

**,arg**

**3)**

**Step 1 = Result**

**1**

**Step 2 = Result**

**2**

**Step 3 = Result 3**

## Nesting Functions

|  |
| --- |
| **SELECT last\_name,**  **UPPER(CONCAT(SUBSTR (LAST\_NAME, 1, 8), '\_US'))**  **FROM employees**  **WHERE department\_id = 60;** |



## Lesson Agenda

* Implicit and explicit data type conversion
* TO\_CHAR, TO\_DATE, TO\_NUMBER functions
* Nesting functions
* General functions:
  + NVL
  + NVL2
  + NULLIF
  + COALESCE
* Conditional expressions:
  + CASE
  + DECODE

## General Functions

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

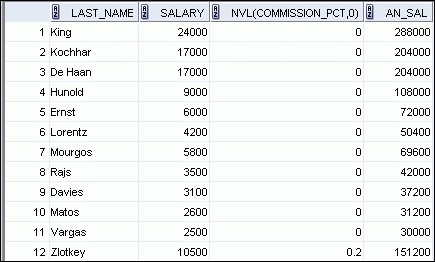
* NVL(expr1, expr2)
* NVL2(expr1, expr2, expr3)
* NULLIF(expr1, expr2)
* COALESCE(expr1, expr2, ..., exprn)

## 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')

## Using the NVL Function



**SELECT last\_name, salary, NVL(commission\_pct, 0),**

**(**

**salary\*12) + (salary\*12\*NVL(commission\_pct, 0)) AN\_SAL**

**FROM employees;**

**…**

**1**

**1**

**2**

**2**

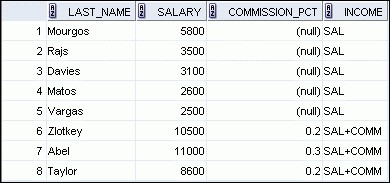
## Using the NVL2 Function

**SELECT last\_name, salary, commission\_pct~~,~~ 1**

**NVL2(commission\_pct,**

**'SAL+COMM', 'SAL') income 2**

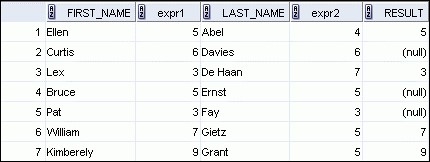
**FROM employees WHERE department\_id IN (50, 80);**



**1**

**2**

## Using the NULLIF Function



**SELECT first\_name,**

**LENGTH(first\_name) "expr1",**

**last\_name, LENGTH(last\_name) "expr2",**

**NULLIF(LENGTH(first\_name), LENGTH(last\_name)) result**

**FROM employees;**

**…**

**1**

**2**

**3**



**1**

**2**

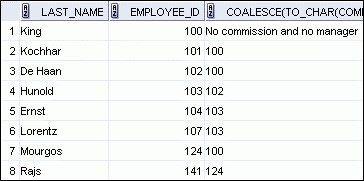
**3**

## 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.

## 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;** |



**…**



**…**

## Lesson Agenda

* Implicit and explicit data type conversion
* TO\_CHAR, TO\_DATE, TO\_NUMBER functions
* Nesting functions
* General functions:
  + NVL
  + NVL2
  + NULLIF
  + COALESCE
* Conditional expressions:
  + CASE
  + DECODE

## Conditional Expressions

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

## 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** |

## 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;** |



**…**

**…**

**…**

## 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*])** |

## Using the DECODE Function

|  |  |
| --- | --- |
| |  | | --- | | **DECODE(job\_id, 'IT\_PROG', 1.10\*salary, 'ST\_CLERK', 1.15\*salary,**  **'SA\_REP', 1.20\*salary,**  **salary) REVISED\_SALARY** |   **SELECT last\_name, job\_id, salary,**  **FROM employees;** |



**…**

**…**

**…**

## 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;** |

## Reporting Aggregated Data Using the Group Functions Objectives

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

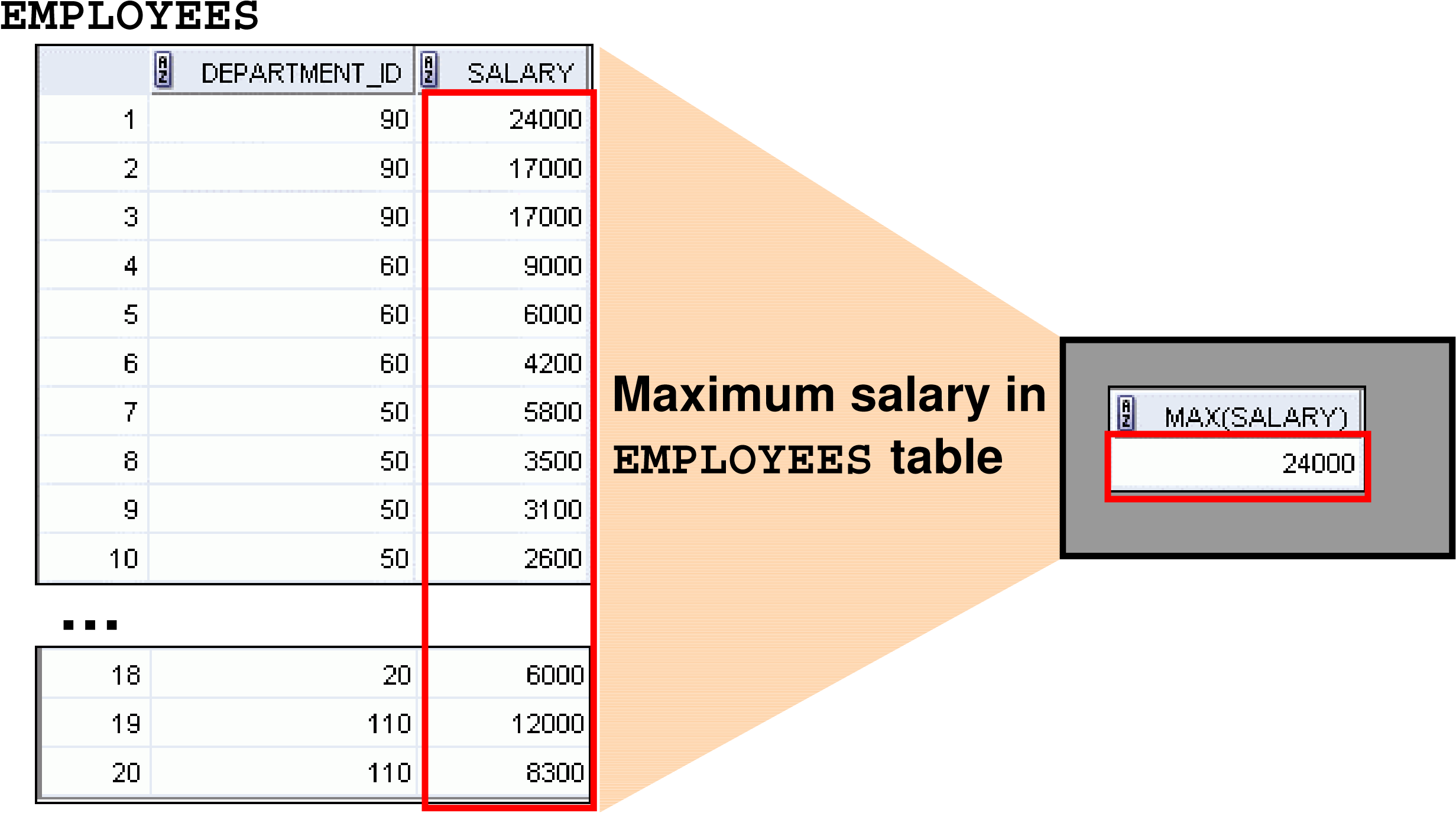
* Identify the available group functions
* Describe the use of group functions
* Group data by using the GROUPBY clause
* Include or exclude grouped rows by using the HAVING clause

## Lesson Agenda

* Group functions:
  + Types and syntax
  + Use AVG, SUM, MIN, MAX, COUNT
  + Use DISTINCT keyword within group functions
  + NULL values in a group function
* Grouping rows:
  + GROUPBY clause
  + HAVING clause
* Nesting group functions

**What Are Group Functions?**

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



## Types of Group Functions

* AVG
* COUNT
* MAX
* MIN

**Group**

**functions**

* STDDEV
* SUM
* VARIANCE

## Group Functions: Syntax

**SELECT *group\_function(column), ...***

**FROM *table***

**[WHERE *condition*]**

**[ORDER BY *column*];**

## 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%';** |



## 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;**



## Using the COUNT Function

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

|  |  |  |
| --- | --- | --- |
| **1** | |  | | --- | | **SELECT COUNT(\*)**  **FROM employees**  **WHERE department\_id = 50;** | |



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

|  |  |  |
| --- | --- | --- |
| **2** | |  | | --- | | **SELECT COUNT(commission\_pct)**  **FROM employees**  **WHERE department\_id = 80;** | |



## Using the DISTINCT Keyword

* COUNT(DISTINCTexpr) 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;**



## Group Functions and Null Values

Group functions ignore null values in the column:

|  |  |  |
| --- | --- | --- |
| **1** | |  | | --- | | **SELECT AVG(commission\_pct) FROM employees;** | |



The NVL function forces group functions to include null values:

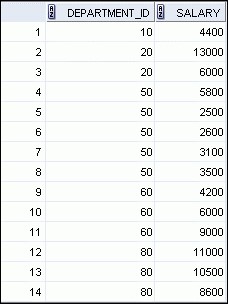
|  |  |  |
| --- | --- | --- |
| **2** | |  | | --- | | **SELECT AVG(NVL(commission\_pct, 0)) FROM employees;** | |



## Lesson Agenda

* Group functions:
  + Types and syntax
  + Use AVG, SUM, MIN, MAX, COUNT
  + Use DISTINCT keyword within group functions
  + NULL values in a group function
* Grouping rows:
  + GROUPBY clause
  + HAVING clause
* Nesting group functions

## Creating Groups of Data



**EMPLOYEES**

**…**

**4400**

**9500**

**3500**

**6400**

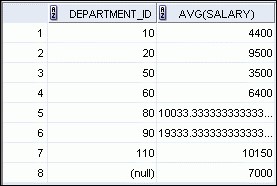
**10033**

**Average salary in**

**EMPLOYEES**

**table for**

**each department**



**Creating Groups of Data:**

## GROUPBY Clause Syntax

|  |  |
| --- | --- |
| **SELECT *column*, *group\_function(column)***  **FROM *table***  **[WHERE *condition*]**   |  | | --- | | **[GROUP BY *group\_by\_expression*]** |   **[ORDER BY *column*];** |

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

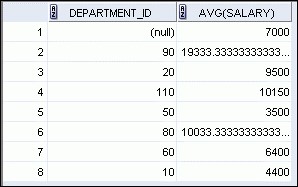
## Using the GROUPBY Clause

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

**SELECT department\_id, AVG(salary)**

**FROM employees**

**GROUP BY department\_id ;**



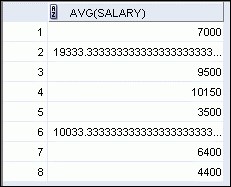
## Using the GROUPBY Clause

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

**SELECT AVG(salary)**

**FROM employees**

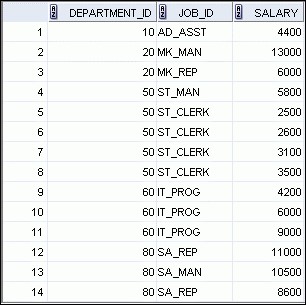
**GROUP BY department\_id ;**



## Grouping by More than One Column

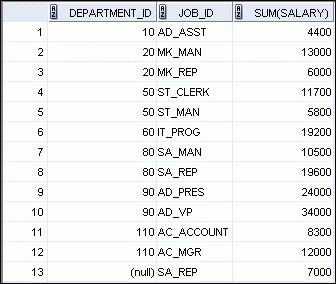
**EMPLOYEES Add the salaries in the EMPLOYEES**

**table for each job, grouped by**



**department.**

**…**



## Using the GROUPBY Clause on Multiple Columns

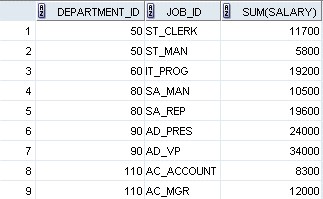
**SELECT department\_id, job\_id, SUM(salary)**

**FROM employees**

**WHERE department\_id > 40**

**GROUP BY department\_id, job\_id**

**ORDER BY department\_id;**



## Illegal Queries Using Group Functions

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

**SELECT department\_id, COUNT(last\_name)**

**FROM employees;**

**A GROUPBY 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;**

**Either add job\_id in the GROUPBY or remove the job\_id column from the SELECT list.**



## 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;**

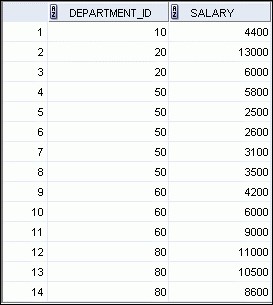
**Cannot use the**

**WHERE clause to**

**restrict groups**

## Restricting Group Results

**EMPLOYEES**

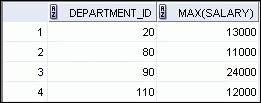


**…**

**The maximum salary per**

**department when it is**

**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*];** |

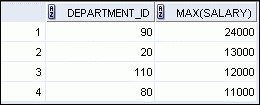
## Using the HAVING Clause

**SELECT department\_id, MAX(salary)**

**FROM employees**

**GROUP BY department\_id**

**HAVING MAX(salary)>10000 ;**



## 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);**



## Lesson Agenda

* Group functions:
  + Types and syntax
  + Use AVG, SUM, MIN, MAX, COUNT
  + Use DISTINCT keyword within group functions
  + NULL values in a group function
* Grouping rows:
  + GROUPBY clause
  + HAVING clause
* Nesting group functions

## Nesting Group Functions

Display the maximum average salary:

**SELECT MAX(AVG(salary))**

**FROM employees**

**GROUP BY department\_id;**



## Displaying Data from Multiple Tables Objectives

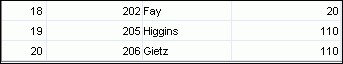
After completing this lesson, you should be able to do the following:

* Write SELECT statements to access data from more than one table using equijoins and nonequijoins
* Join a table to itself by using a self-join
* View data that generally does not meet a join condition by using OUTER joins
* Generate a Cartesian product of all rows from two or more tables

## Lesson Agenda

* Types of JOINS and its syntax
* Natural join:
  + USING clause
  + ON clause
* Self-join
* Nonequijoins
* OUTER join:
  + LEFTOUTER join
  + RIGHTOUTER join
  + FULLOUTER join
* Cartesian product – Cross join

## Obtaining Data from Multiple Tables

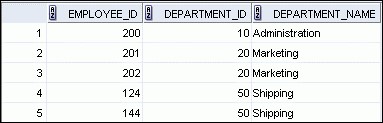
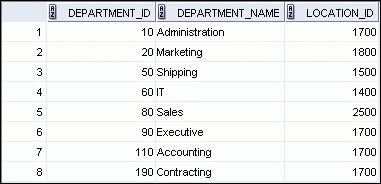


**EMPLOYEES**

**DEPARTMENTS**

**…**

**…**



## Types of Joins

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

* Natural joins:
  + NATURALJOIN clause
  + USING clause
  + ON clause • OUTER joins:
  + LEFTOUTERJOIN
  + RIGHTOUTERJOIN
  + FULLOUTERJOIN
* Cross joins

## 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*];**

## 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.

## Lesson Agenda

* Types of JOINS and its syntax
* Natural join:
  + USING clause
  + ON clause
* Self-join
* Nonequijoins
* OUTER join:
  + LEFTOUTER join
  + RIGHTOUTER join
  + FULLOUTER join
* Cartesian product – Cross join

## Creating Natural Joins

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

## Retrieving Records with Natural Joins

**SELECT department\_id, department\_name, location\_id, city**

**FROM departments**



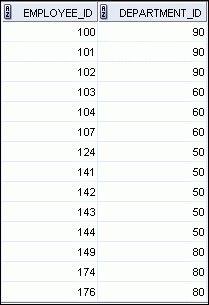
**NATURAL JOIN locations**

**;**

## 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 NATURALJOIN and USING clauses are mutually exclusive.

## Joining Column Names



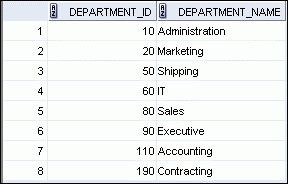
**EMPLOYEES**

**DEPARTMENTS**

**Foreign key**

**Primary key**

**…**



## Retrieving Records with the USING Clause

**SELECT employee\_id, last\_name, location\_id, department\_id FROM employees JOIN departments**

**USING (department\_id) ;**



**…**



## Using Table Aliases with the USING Clause

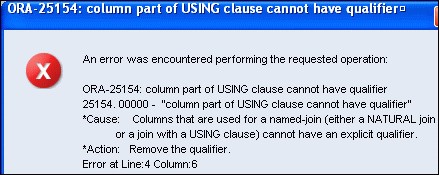
* Do not qualify a column that is used in the USING clause.
* If the same column is used elsewhere in the SQL statement, do not alias it.

**SELECT l.city, d.department\_name**

**FROM locations l JOIN departments d**

**USING (location\_id)**

**WHERE d.location\_id = 1400;**



## 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.

|  |  |
| --- | --- |
|  |  |
|  |

## Retrieving Records with the ON Clause

|  |
| --- |
| **SELECT e.employee\_id, e.last\_name, e.department\_id,**  **d.department\_id, d.location\_id FROM employees e JOIN departments d**  **ON (e.department\_id = d.department\_id);** |

**…**

## 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;** |  |



**…**

## 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**  **ON (e.department\_id = d.department\_id)**  **AND e.manager\_id = 149 ;** |

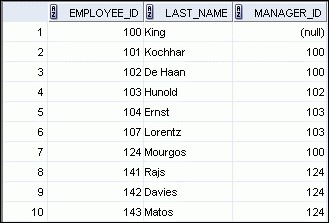
**Or**

|  |  |
| --- | --- |
| **SELECT e.employee\_id, e.last\_name, e.department\_id,**  **d.department\_id, d.location\_id FROM employees e JOIN departments d**  **ON (e.department\_id = d.department\_id)** | |
| **WHERE e.manager\_id = 149** | **;** |

## Lesson Agenda

* Types of JOINS and its syntax
* Natural join:
  + USING clause
  + ON clause
* Self-join
* Nonequijoins
* OUTER join:
  + LEFTOUTER join
  + RIGHTOUTER join
  + FULLOUTER join
* Cartesian product – Cross join

## Joining a Table to Itself



**EMPLOYEES (WORKER)**

**EMPLOYEES (MANAGER)**

**…**

**…**



**MANAGER\_ID in the WORKER table is equal to EMPLOYEE\_ID in the MANAGER table.**

## 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);**

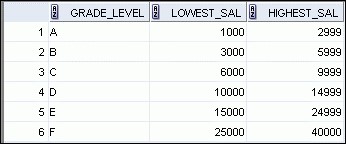


**…**

## Lesson Agenda

* Types of JOINS and its syntax
* Natural join:
  + USING clause
  + ON clause
* Self-join
* Nonequijoins
* OUTER join:
  + LEFTOUTER join
  + RIGHTOUTER join
  + FULLOUTER join
* Cartesian product – Cross join

## Nonequijoins



**EMPLOYEES**

**JOB\_GRADES**

**…**

**JOB\_GRADES**

**table defines the**

**LOWEST\_SAL**

**and**

**HIGHEST\_SAL**

**range**

**of values for each**

**GRADE\_LEVEL**

**.**

**Hence, the**

**GRADE\_LEVEL**

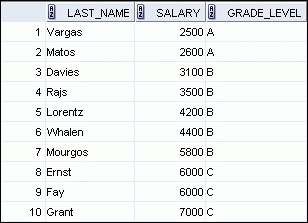
**column can**

**be used to assign grades to each**

**employee.**

## 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\_sa** | **l;** |



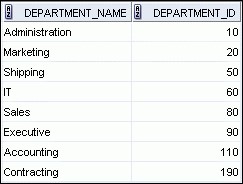
**…**

## Lesson Agenda

* Types of JOINS and its syntax
* Natural join:
  + USING clause
  + ON clause
* Self-join
* Nonequijoins
* OUTER join:
  + LEFTOUTER join
  + RIGHTOUTER join
  + FULLOUTER join
* Cartesian product – Cross join

## Returning Records with No Direct Match Using OUTER Joins

**There are no employees in department 190.**



**Equijoin with**

**EMPLOYEES**

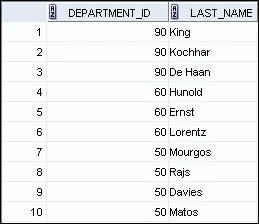
**DEPARTMENTS**

**Employee “Grant” has**

**not been assigned a**

**department ID.**

**…**

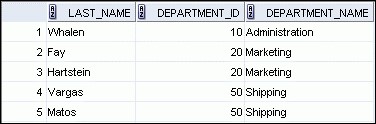


## INNER Versus OUTER Joins

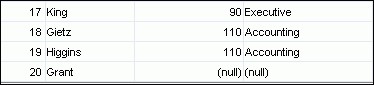
* In SQL:1999, the join of two tables returning only matched rows is called an INNER join.
* A join between two tables that returns the results of the INNER join as well as the unmatched rows from the left (or right) table is called a left (or right) OUTER join.
* A join between two tables that returns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

# LEFTOUTERJOIN

|  |
| --- |
| **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) ;** |



**…**



# RIGHTOUTERJOIN

**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) ;**



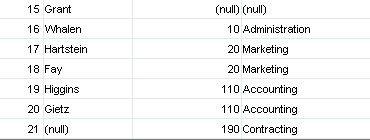
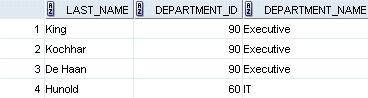
**…**



# FULLOUTERJOIN

|  |
| --- |
| **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) ;** |

**…**



## Lesson Agenda

* Types of JOINS and its syntax
* Natural join:
  + USING clause
  + ON clause
* Self-join • Nonequiijoin
* OUTER join:
  + LEFTOUTER join
  + RIGHTOUTER join
  + FULLOUTER join
* Cartesian product – Cross join

## 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
* To avoid a Cartesian product, always include a valid join condition.

## Generating a Cartesian Product

**Cartesian product:**

**20**

**x 8 = 160 rows**

**EMPLOYEES**

**(20**

**rows**

**)**

**DEPARTMENTS**

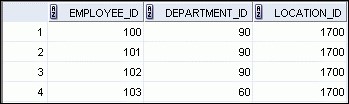
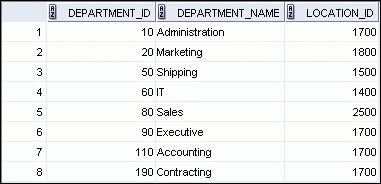
**(8**

**rows**

**)**

**…**

**…**



## Creating Cross Joins

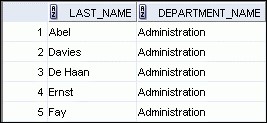
* The CROSSJOIN clause produces the cross-product of two tables.
* This is also called a Cartesian product between the two tables.

**SELECT last\_name, department\_name**

**FROM employees**

**CROSS JOIN departments ;**

**…**



**Using Subqueries to Solve Queries**

## Objectives

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

* Define subqueries
* Describe the types of problems that the subqueries can solve
* List the types of subqueries
* Write single-row and multiple-row subqueries

## Lesson Agenda

* Subquery: Types, syntax, and guidelines
* Single-row subqueries:
  + Group functions in a subquery
  + HAVING clause with subqueries
* Multiple-row subqueries
  + Use ALL or ANY operator.
* Null values in a subquery

## Using a Subquery to Solve a Problem

Who has a salary greater than Abel’s?

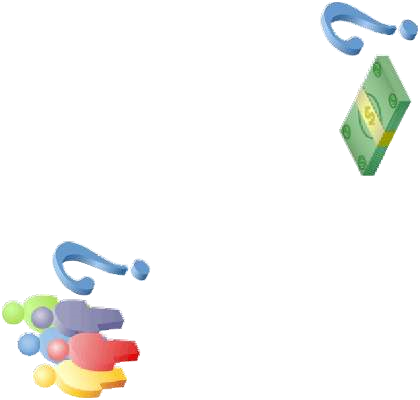
**Which employees have salaries greater than Abel’s**

**salary?**

**Main query:**

**What is Abel’s salary?**

**Subquery:**



## 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.

## Using a Subquery

**SELECT last\_name, salary**

**FROM employees**

**WHERE salary >**

**(**

**SELECT salary**

**FROM employees**

**WHERE last\_name = 'Abel');**

**11000**



## 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.

## Types of Subqueries

* Single-row subquery

**ST\_CLERK**

**Main query**

**Subquery**

**returns**

* Multiple-row subquery

**ST\_CLERK**

**Main query**

**Subquery**

**returns**

**SA\_MAN**

## Lesson Agenda

* Subquery: Types, syntax, and guidelines
* Single-row subqueries:
  + Group functions in a subquery
  + HAVING clause with subqueries
* Multiple-row subqueries
  + Use ALL or ANY operator
* Null values in a subquery

## 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 |

## 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’);**



## Using Group Functions in a Subquery

|  |  |  |
| --- | --- | --- |
| **SELECT last\_name, job\_id, salary**  **FROM employees**  **WHERE salary =  2500** | | |
|  | **(SELECT MIN(salary) FROM employees);** |  |



## The 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)**

**>**

**(**

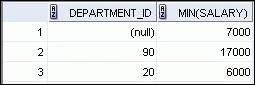
**SELECT MIN(salary**

**)**

**FROM employees**

**WHERE department\_id = 50);**

**2500**

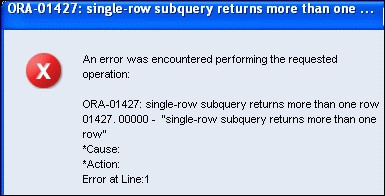


**…**



**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**

## No Rows Returned by the Inner Query

|  |  |
| --- | --- |
| **SELECT last\_name, job\_id**  **FROM employees**   |  | | --- | | **(SELECT job\_id**  **FROM employees**  **WHERE last\_name = 'Haas');** |   **WHERE job\_id =** |

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

## Lesson Agenda

* Subquery: Types, syntax, and guidelines
* Single-row subqueries:
  + Group functions in a subquery
  + HAVING clause with subqueries
* Multiple-row subqueries
  + Use ALL or ANY operator
* Null values in a subquery

## 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. |

## Using the ANY Operator in Multiple-Row Subqueries

**SELECT employee\_id, last\_name, job\_id, salary**

**FROM employees**

**WHERE salary <**

**ANY**

**(**

**SELECT salary**

**FROM employees**

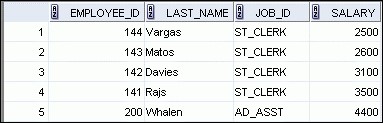
**WHERE job\_id = 'IT\_PROG')**

**AND job\_id <> 'IT\_PROG';**

**9000**

**, 6000,**

**4200**



**…**

## Using the ALL Operator in Multiple-Row Subqueries

**SELECT employee\_id, last\_name, job\_id, salary**

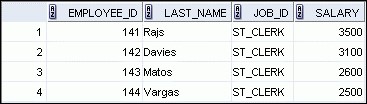
**FROM employees 9000, 6000, 4200**

**WHERE salary < ALL**

**(SELECT salary**

**FROM employees**

**WHERE job\_id = 'IT\_PROG') AND job\_id <> 'IT\_PROG';**



## Lesson Agenda

* Subquery: Types, syntax, and guidelines
* Single-row subqueries:
  + Group functions in a subquery
  + HAVING clause with subqueries
* Multiple-row subqueries
  + Use ALL or ANY operator
* Null values in a subquery

## 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);** |

**Using the Set Operators**

## Objectives

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

* Describe set operators
* Use a set operator to combine multiple queries into a single query
* Control the order of rows returned

## Lesson Agenda

* Set Operators: Types and guidelines
* Tables used in this lesson
* UNION and UNIONALL operator
* INTERSECT operator
* MINUS operator
* Matching the SELECT statements
* Using the ORDERBY clause in set operations

## Set Operators

**UNION/UNIONALL**

**A**

**B**

**A**

**B**

**A**

**B**

**INTERSECT MINUS**

**A B**

## 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.
* ORDERBY clause can appear only at the very end of the statement.

## The Oracle Server and Set Operators

* Duplicate rows are automatically eliminated except in UNIONALL.
* Column names from the first query appear in the result.
* The output is sorted in ascending order by default except in UNIONALL.

## Lesson Agenda

* Set Operators: Types and guidelines
* Tables used in this lesson
* UNION and UNIONALL operator
* INTERSECT operator
* MINUS operator
* Matching the SELECT statements
* Using the ORDERBY clause in set operations

## 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

## Lesson Agenda

* Set Operators: Types and guidelines
* Tables used in this lesson
* UNION and UNIONALL operator
* INTERSECT operator
* MINUS operator
* Matching the SELECT statements
* Using the ORDERBY clause in set operations

## UNION Operator

**A**

**B**

**The UNION operator returns rows from both queries after eliminating duplications.**

## 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;** |

**…**



**…**



# UNIONALL Operator

**A**

**B**

**The UNIONALL operator returns rows from both queries, including all duplications.**

## Using the UNIONALL Operator

Display the current and previous departments of all employees.

**SELECT employee\_id, job\_id, department\_id**

**FROM employees**

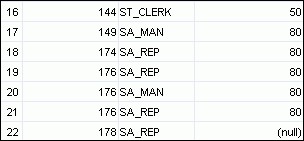
**UNION ALL**

**SELECT employee\_id, job\_id, department\_id**

**FROM job\_history**

**ORDER BY employee\_id;**

**…**



**…**

## Lesson Agenda

* Set Operators: Types and guidelines
* Tables used in this lesson
* UNION and UNIONALL operator
* INTERSECT operator
* MINUS operator
* Matching the SELECT statements
* Using ORDERBY clause in set operations

# INTERSECT Operator

**A**

**B**

**The INTERSECT operator returns rows that are common to both queries.**

## 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;**



## Lesson Agenda

* Set Operators: Types and guidelines
* Tables used in this lesson
* UNION and UNIONALL operator
* INTERSECT operator
* MINUS operator
* Matching the SELECT statements
* Using the ORDERBY clause in set operations

## MINUS Operator

**A**

**B**

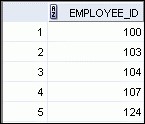
**The MINUS operator returns all the distinct rows selected by the first query, but not present in the second query result set.**

## 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;** |

**…**



## Lesson Agenda

* Set Operators: Types and guidelines
* Tables used in this lesson
* UNION and UNIONALL operator
* INTERSECT operator
* MINUS operator
* Matching the SELECT statements
* Using ORDERBY clause in set operations

## 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;**

## Matching the SELECT Statement: Example

Using the UNION operator, display the employee ID, job ID, and salary of all employees.

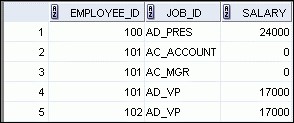
**SELECT employee\_id, job\_id,salary**

**FROM employees**

**UNION**

**SELECT employee\_id, job\_id,0**

**FROM job\_history;**



**…**



## Lesson Agenda

* Set Operators: Types and guidelines
* Tables used in this lesson
* UNION and UNIONALL operator
* INTERSECT operator
* MINUS operator
* Matching the SELECT statements
* Using the ORDERBY clause in set operations

## Using the ORDERBY Clause in Set Operations

* The ORDERBY clause can appear only once at the end of the compound query.
* Component queries cannot have individual ORDERBY clauses.
* ORDERBY 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.

**Manipulating Data**

## Objectives

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

* Describe each data manipulation language (DML) statement
* Insert rows into a table
* Update rows in a table
* Delete rows from a table
* Control transactions

## Lesson Agenda

* Adding new rows in a table – INSERT statement
* Changing data in a table – UPDATE statement
* Removing rows from a table:
  + DELETE statement
  + TRUNCATE statement
* Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
* Read consistency
* FORUPDATE clause in a SELECT statement

## 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.

## Adding a New Row to a Table

**DEPARTMENTS**

**New**

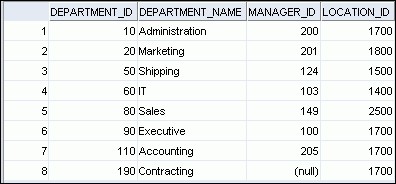
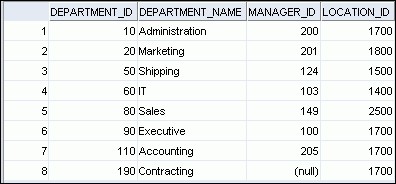
**row**

**Insert new row**

**into 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.

## 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);** |

* Enclose character and date 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');** |

* Explicit method: Specify the NULL keyword in the VALUES clause.

|  |
| --- |
| **INSERT INTO departments**  **VALUES (100, 'Finance', NULL, NULL);** |

**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);** |

## 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);** |

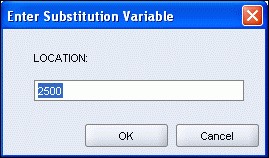
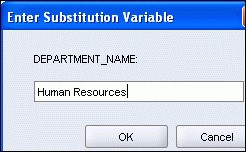
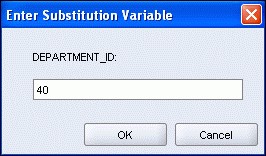
* Verify your addition.



## Creating a Script

* Use & 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);** |



**Copying Rows from Another Table** • Write your INSERT statement with a subquery:

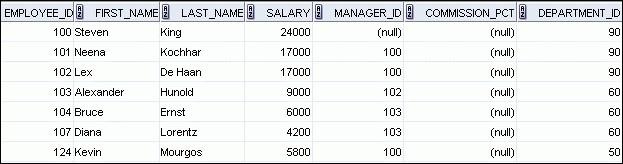
|  |  |
| --- | --- |
| **INSERT INTO sales\_reps(id, name, salary, commission\_pct)**   |  | | --- | | **SELECT employee\_id, last\_name, salary, commission\_pct**  **FROM employees**  **WHERE job\_id LIKE '%REP%';** | |

* 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.

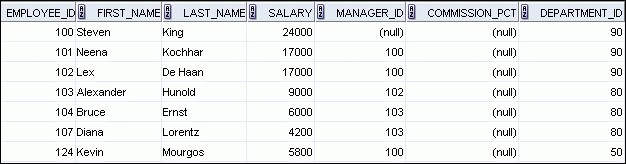
## Lesson Agenda

* Adding new rows in a table – INSERT statement
* Changing data in a table – UPDATE statement
* Removing rows from a table:
  + DELETE statement
  + TRUNCATE statement
* Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
* Read consistency
* FORUPDATE clause in a SELECT statement

## Changing Data in a Table



**EMPLOYEES**



**Update rows in the**

**EMPLOYEES**

**table:**

## 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).

## 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;** |

* Values for all the rows in the table are modified if you omit the WHERE clause:

|  |
| --- |
| **UPDATE copy\_emp**  **SET department\_id = 110;** |

* Specify SET*column\_name*=NULL to update a column value to NULL.

## Updating Two Columns with a Subquery

Update employee 113’s job and salary to match those of employee 205.

|  |  |
| --- | --- |
| **UPDATE employees**   |  | | --- | | **FROM employees**  **WHERE employee\_id = 205),**  **FROM employees**  **WHERE employee\_id = 205)** |   **SET job\_id = (SELECT job\_id**  **salary = (SELECT salary**  **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);** |

## Lesson Agenda

* Adding new rows in a table – INSERT statement
* Changing data in a table – UPDATE statement
* Removing rows from a table:
  + DELETE statement
  + TRUNCATE statement
* Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
* Read consistency
* FORUPDATE clause in a SELECT statement

## Removing a Row from a Table



**DEPARTMENTS**

**Delete a row from the DEPARTMENTS table:**



## DELETE Statement

You can remove existing rows from a table by using the DELETE statement:

|  |  |
| --- | --- |
| **DELETE [FROM]** | ***table*** |
| **[WHERE** | ***condition*];** |

## Deleting Rows from a Table

* Specific rows are deleted if you specify the WHERE clause:

|  |
| --- |
| **DELETE FROM departments**  **WHERE department\_name = ‘Finance';** |

* All rows in the table are deleted if you omit the WHERE clause:

|  |
| --- |
| **DELETE FROM copy\_emp;** |

## 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%');** | |

## 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;**

## Lesson Agenda

* Adding new rows in a table – INSERT statement
* Changing data in a table – UPDATE statement
* Removing rows from a table:
  + DELETE statement
  + TRUNCATE statement
* Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
* Read consistency
* FORUPDATE clause in a SELECT statement

## 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

## 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 the SQL environment.
  + The system crashes.

## 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

## Explicit Transaction Control Statements

***Time COMMIT***

**ROLLBACK ROLLBACK ROLLBACK**

**SAVEPOINTB**

**SAVEPOINTA**

**DELETE**

**INSERT**

**UPDATE**

**INSERT**

**Transaction**

**to SAVEPOINT B to SAVEPOINT A**

## 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 ROLLBACKTO SAVEPOINT statement.

|  |
| --- |
| **UPDATE...**  **SAVEPOINT update\_done;**    **INSERT...**  **ROLLBACK TO update\_done;** |

## Implicit Transaction Processing

* An automatic commit occurs in the following circumstances:
  + A DDL statement is issued
  + A DCL statement is 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.

**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.

## 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.

## Committing Data

* Make the changes:

|  |
| --- |
| **DELETE FROM employees**  **WHERE employee\_id = 99999;**    **INSERT INTO departments**  **VALUES (290, 'Corporate Tax', NULL, 1700);** |

* Commit the changes:

**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 ;**

## 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.** |

## 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.

## Lesson Agenda

* Adding new rows in a table – INSERT statement
* Changing data in a table – UPDATE statement
* Removing rows from a table:
  + DELETE statement
  + TRUNCATE statement
* Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
* Read consistency
* FORUPDATE clause in a SELECT 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

## Implementing Read Consistency

**User A**

**SELECT \***

**FROM userA.employees;**

**UPDATE employees**

**SET salary = 7000**

**WHERE last\_name = 'Grant';**

**Data**

**blocks**

**Undo**

**segments**

**Changed**

**and**

**unchanged**

**data**

**Before**

**change**

**)**

**“old” data**

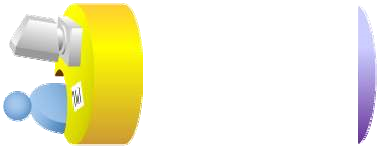
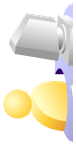
**(**

**User B**

**Read-**

**consistent**

**image**



## Lesson Agenda

* Adding new rows in a table – INSERT statement
* Changing data in a table – UPDATE statement
* Removing rows from a table:
  + DELETE statement
  + TRUNCATE statement
* Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
* Read consistency
* FORUPDATE clause in a SELECT statement

## FORUPDATE Clause in a SELECT Statement

* Locks the rows in the EMPLOYEES table where job\_id is SA\_REP.

**SELECT employee\_id, salary, commission\_pct, job\_id**

**FROM employees**

**WHERE job\_id = 'SA\_REP'**

**FOR UPDATE**

**ORDER BY employee\_id;**

* Lock is released only when you issue a ROLLBACK or a COMMIT.
* If the SELECT statement attempts to lock a row that is locked by another user, then the database waits until the row is available, and then returns the results of the SELECT statement.

## FORUPDATE Clause: Examples

* You can use the FORUPDATE clause in a SELECT statement against multiple tables.

**SELECT e.employee\_id, e.salary, e.commission\_pct**

**FROM employees e JOIN departments d**

**USING (department\_id)**

**WHERE job\_id = 'ST\_CLERK‘**

**AND location\_id = 1500**

**FOR UPDATE**

**ORDER BY e.employee\_id;**

* Rows from both the EMPLOYEES and DEPARTMENTS tables are locked.
* Use FORUPDATEOF*column\_name* to qualify the column

you intend to change, then only the rows from that specific table are locked.

## Using DDL Statements to Create and Manage Tables Objectives

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

* Categorize the main database objects
* Review the table structure
* List the data types that are available for columns
* Create a simple table
* Explain how constraints are created at the time of table creation
* Describe how schema objects work

## Lesson Agenda

* Database objects
  + Naming rules
* CREATETABLE statement:
  + Access another user’s tables
  + DEFAULT option
* Data types
* Overview of constraints: NOTNULL, UNIQUE, PRIMARY KEY, FOREIGNKEY, CHECK constraints
* Creating a table using a subquery
* ALTERTABLE
  + Read-only tables
* DROPTABLE statement

## 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 |

## Naming Rules

Table names and column names:

* Must begin with a letter
* Must be 1–30 characters long
* Must contain only A–Z, a–z, 0–9, \_, $, and #
* Must not duplicate the name of another object owned by the same user
* Must not be an Oracle server–reserved word

## Lesson Agenda

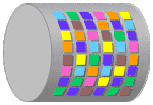
* Database objects
  + Naming rules
* CREATETABLE statement:
  + Access another user’s tables
  + DEFAULT option
* Data types
* Overview of constraints: NOTNULL, UNIQUE, PRIMARY KEY, FOREIGNKEY, CHECK constraints
* Creating a table using a subquery
* ALTERTABLE
  + Read-only tables
* DROPTABLE statement

# CREATETABLE Statement

* You must have:
  + CREATETABLE privilege – A storage area

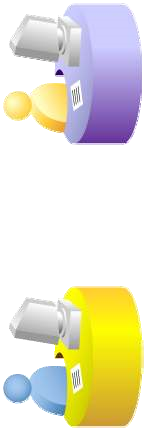
|  |
| --- |
| **CREATE TABLE [*schema*.]*table***  **(*column datatype* [DEFAULT *expr*][, ...]);** |

* You specify:
  + Table name
  + Column name, column data type, and column size



## 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.



**USERB**

**USERA**

**SELECT \***

**SELECT \***

**FROM userB.employees; 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);** | |

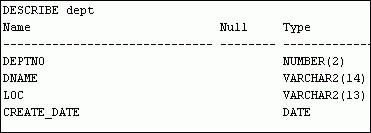
## Creating Tables

* Create the table:

|  |
| --- |
| **CREATE TABLE dept**  **(deptno NUMBER(2), dname VARCHAR2(14), loc VARCHAR2(13), create\_date DATE DEFAULT SYSDATE);** |

* Confirm table creation:

**DESCRIBE dept**



## Lesson Agenda

* Database objects
  + Naming rules
* CREATETABLE statement:
  + Access another user’s tables
  + DEFAULT option
* Data types
* Overview of constraints: NOTNULL, UNIQUE, PRIMARY KEY, FOREIGNKEY, CHECK constraints
* Creating a table using a subquery
* ALTERTABLE
  + Read-only tables
* DROPTABLE statement

## Data Types

|  |  |
| --- | --- |
| Data Type | Description |
| VARCHAR2(*size*) | Variable-length character data |
| CHAR(*size*) | Fixed-length character data |
| NUMBER(*p*,*s)* | 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 |

## 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 |

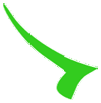


## Lesson Agenda

* Database objects
  + Naming rules
* CREATETABLE statement:
  + Access another user’s tables
  + DEFAULT option
* Data types
* Overview of constraints: NOTNULL, UNIQUE, PRIMARY KEY, FOREIGNKEY, CHECK constraints
* Creating a table using a subquery
* ALTERTABLE
  + Read-only tables
* DROPTABLE statement

## 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:
  + NOTNULL
  + UNIQUE



* + PRIMARYKEY
  + FOREIGNKEY
  + CHECK

## Constraint Guidelines

* You can name a constraint, or the Oracle server generates a name by using the SYS\_C*n* 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.

## 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*, ...),** |

## 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** |

## NOTNULL Constraint

Ensures that null values are not permitted for the column:



**NOTNULL**

**constraint**

**(**

**Primary Key enforces**

**NOTNULL**

**constraint.)**

**Absence of**

**NOTNULL**

**constraint (Any row can**

**contain a null value for**

**NOTNULL**

**constraint**

**…**

**this column.)**

## UNIQUE Constraint



**EMPLOYEES**

**UNIQUE**

**constraint**

**INSERT INTO**

**Not allowed:**

**Allowed**

**…**



**already exists**

## 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));** |

# PRIMARYKEY



**DEPARTMENTS**

**PRIMARY KEY**

**INSERT INTO**

**Not allowed**

**(**

**null value**

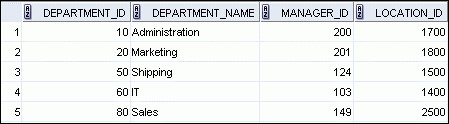
**)**

**Not allowed**



**(50 already exists)**

# FOREIGNKEY



**DEPARTMENTS**

**EMPLOYEES**

**FOREIGN**

**KEY**

**INSERT INTO**

**Not allowed**

**(9**

**does not**

**exist)**

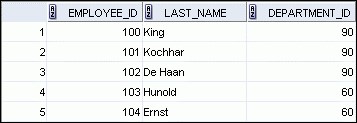
**Allowed**

**PRIMARY**

**KEY**

**…**

**…**



# FOREIGNKEY

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));** |

**FOREIGNKEY Constraint:**

## Keywords

* FOREIGNKEY: Defines the column in the child table at the table-constraint level
* REFERENCES: Identifies the table and column in the parent table
* ONDELETECASCADE: Deletes the dependent rows in the child table when a row in the parent table is deleted
* ONDELETESETNULL: Converts dependent foreign key values to null

## 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),...** |  |

# CREATETABLE: 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));** |

## Violating Constraints

**UPDATE employees**

**SET department\_id = 55**

**WHERE department\_id = 110;**



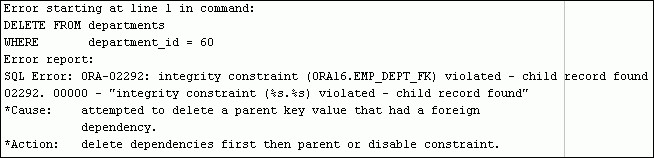
Department 55 does not exist.

## 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;**



## Lesson Agenda

* Database objects
  + Naming rules
* CREATETABLE statement:
  + Access another user’s tables
  + DEFAULT option
* Data types
* Overview of constraints: NOTNULL, UNIQUE, PRIMARY KEY, FOREIGNKEY, CHECK constraints
* Creating a table using a subquery
* ALTERTABLE
  + Read-only tables
* DROPTABLE statement

## 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***

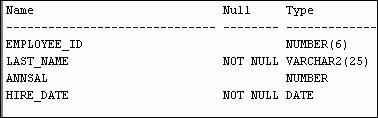
**[(*column*, *column*...)] AS *subquery;***

* Match the number of specified columns to the number of subquery columns.
* Define columns with column names and default values.

## Creating a Table Using a Subquery

|  |  |
| --- | --- |
| |  | | --- | | **SELECT employee\_id, last\_name, salary\*12 ANNSAL,**  **hire\_date FROM employees**  **WHERE department\_id = 80;** |   **CREATE TABLE dept80 AS** |

**DESCRIBE dept80**



## Lesson Agenda

* Database objects
  + Naming rules
* CREATETABLE statement:
  + Access another user’s tables
  + DEFAULT option
* Data types
* Overview of constraints: NOTNULL, UNIQUE, PRIMARY KEY, FOREIGNKEY, CHECK constraints
* Creating a table using a subquery
* ALTERTABLE
  + Read-only tables
* DROPTABLE statement

# ALTERTABLE Statement

Use the ALTERTABLE 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

## Read-Only Tables

You can use the ALTERTABLE 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;**

## Lesson Agenda

* Database objects
  + Naming rules
* CREATETABLE statement:
  + Access another user’s tables
  + DEFAULT option
* Data types
* Overview of constraints: NOTNULL, UNIQUE, PRIMARY KEY, FOREIGNKEY, CHECK constraints
* Creating a table using a subquery
* ALTERTABLE
  + Read-only tables
* DROPTABLE statement

## 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;**



**Creating Other Schema Objects**

## Objectives

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

* Create simple and complex views
* Retrieve data from views
* Create, maintain, and use sequences
* Create and maintain indexes
* Create private and public synonyms

## Lesson Agenda

* Overview of views:
  + Creating, modifying, and retrieving data from a view
  + Data manipulation language (DML) operations on a view
  + Dropping a view • Overview of sequences:
  + Creating, using, and modifying a sequence
  + Cache sequence values
  + NEXTVAL and CURRVAL pseudocolumns
* Overview of indexes
  + Creating, dropping indexes
* Overview of synonyms
  + Creating, dropping synonyms

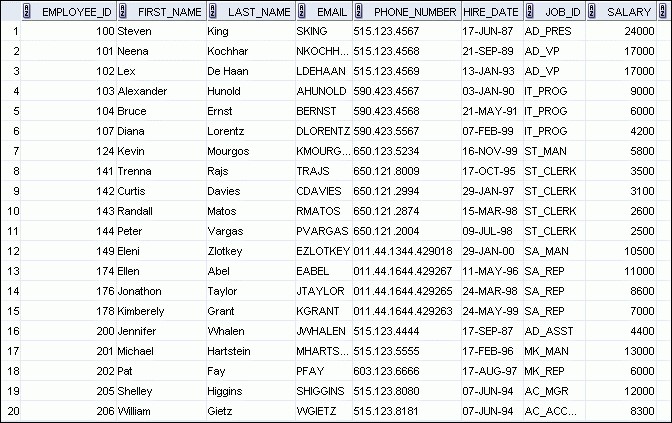
## Database Objects

|  |  |
| --- | --- |
| **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 |

**What Is a View?**

**EMPLOYEES**

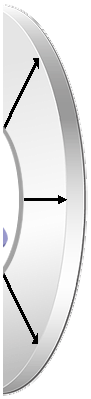
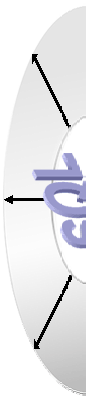
**table**



|  |
| --- |
| **To present different views of**  **the same data** |

|  |
| --- |
| **To provide data**  **independence** |

## Advantages of Views



|  |  |  |
| --- | --- | --- |
| **To restrict data access** |  | **To make complex queries easy** |

## 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 |

* You embed a subquery in the CREATEVIEW statement:

|  |
| --- |
| **CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW *view***  **[(*alias*[, *alias*]...)]**  **AS *subquery***  **[WITH CHECK OPTION [CONSTRAINT *constraint*]]**  **[WITH READ ONLY [CONSTRAINT *constraint*]];** |

* The subquery can contain complex SELECT syntax.
* 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;** |

* Describe the structure of the view by using the *i*SQL\*Plus

DESCRIBE command:

**DESCRIBE empvu80**

* Create a view by using column aliases in the subquery:

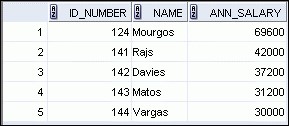
|  |
| --- |
| **CREATE VIEW salvu50**  **AS SELECT employee\_id ID\_NUMBER, last\_name NAME, salary\*12 ANN\_SALARY**  **FROM employees**  **WHERE department\_id = 50;** |

* Select the columns from this view by the given alias names.

## Retrieving Data from a View

**SELECT \***

**FROM salvu50;**



## Modifying a View

* Modify the EMPVU80 view by using a CREATEORREPLACE 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;** |

* Column aliases in the CREATEORREPLACEVIEW clause 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;** |

* You can usually perform DML operations on simple views.
* You cannot remove a row if the view contains the following:



* + Group functions
  + A GROUPBY clause
  + The DISTINCT keyword
  + The pseudocolumn ROWNUM keyword

You cannot modify data in a view if it contains:

* Group functions
* A GROUPBY clause
* The DISTINCT keyword
* The pseudocolumn ROWNUM keyword
* Columns defined by expressions

You cannot add data through a view if the view includes:

* Group functions
* A GROUPBY clause
* The DISTINCT keyword
* The pseudocolumn ROWNUM keyword
* Columns defined by expressions
* NOTNULL columns in the base tables that are not selected by the view

## Using the WITHCHECKOPTION Clause

* You can ensure that DML operations performed on the view stay in the domain of the view by using the WITH CHECKOPTION clause:

|  |
| --- |
| **CREATE OR REPLACE VIEW empvu20**  **AS SELECT \***  **FROM employees**  **WHERE department\_id = 20**  **WITH CHECK OPTION CONSTRAINT empvu20\_ck ;** |

* 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 CHECKOPTION constraint.

## Denying DML Operations

* You can ensure that no DML operations occur by adding the WITHREADONLY option to your view definition.
* Any attempt to perform a DML operation on any row in the view results in an Oracle server error.



## 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 ;** |

## 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;**



## Practice 11: Overview of Part 1

This practice covers the following topics:

* Creating a simple view
* Creating a complex view
* Creating a view with a check constraint
* Attempting to modify data in the view
* Removing views

## Lesson Agenda

* Overview of views:
  + Creating, modifying, and retrieving data from a view
  + DML operations on a view
  + Dropping a view• Overview of sequences:
  + Creating, using, and modifying a sequence
  + Cache sequence values
  + NEXTVAL and CURRVAL pseudocolumns
* Overview of indexes
  + Creating, dropping indexes
* Overview of synonyms
  + Creating, dropping synonyms

## 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 |

## 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

**2**

**4**

**3**

**5**

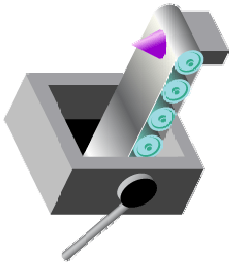
**6**

**8**

**7**

**10**

**9**



**1**

**CREATESEQUENCE 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}];** |

## 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;** |

## 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.

## 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);** |

* View the current value for the DEPT\_DEPTID\_SEQ sequence:

**SELECT dept\_deptid\_seq.CURRVAL**

**FROM dual;**

## 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

## 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;** |

## 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;**



## Lesson Agenda

* Overview of views:
  + Creating, modifying, and retrieving data from a view
  + DML operations on a view
  + Dropping a view• Overview of sequences:
  + Creating, using, and modifying a sequence
  + Cache sequence values
  + NEXTVAL and CURRVAL pseudocolumns
* Overview of indexes
  + Creating, dropping indexes
* Overview of synonyms
  + Creating, dropping synonyms

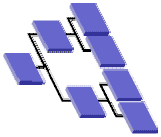
## Indexes

|  |  |
| --- | --- |
| **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 |

## Indexes

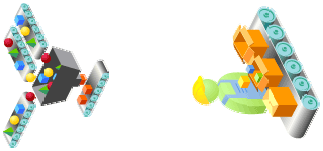
An index:

* Is a schema object
* May 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



**How Are Indexes Created?**

* Automatically: A unique index is created automatically when you define a PRIMARYKEY or UNIQUE constraint in a table definition.
* Manually: Users can create nonunique indexes on columns to speed up access to the rows.



## 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);** |

## 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 |

## Removing an Index

* Remove an index from the data dictionary by using the DROPINDEX command:

**DROP INDEX *index*;**

* Remove the emp\_last\_name\_idx index from the data dictionary:

**DROP INDEX emp\_last\_name\_idx;**



* To drop an index, you must be the owner of the index or have the DROPANYINDEX privilege.

## Lesson Agenda

* Overview of views:
  + Creating, modifying, and retrieving data from a view
  + DML operations on a view
  + Dropping a view• Overview of sequences:
  + Creating, using, and modifying a sequence
  + Cache sequence values
  + NEXTVAL and CURRVAL pseudocolumns
* Overview of indexes
  + Creating, dropping indexes
* Overview of synonyms
  + Creating, dropping synonyms

## 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 |

## 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*;**

## Creating and Removing Synonyms

* Create a shortened name for the DEPT\_SUM\_VU view:

|  |
| --- |
| **CREATE SYNONYM d\_sum FOR dept\_sum\_vu;** |

* Drop a synonym:

**DROP SYNONYM d\_sum;**



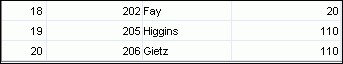
**Appendix**

## Objectives

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

* Write SELECT statements to access data from more than one table using equijoins and nonequijoins
* Join a table to itself by using a self-join
* View data that generally does not meet a join condition by using outer joins
* Generate a Cartesian product of all rows from two or more tables

## Obtaining Data from Multiple Tables

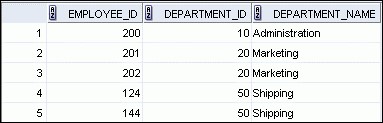
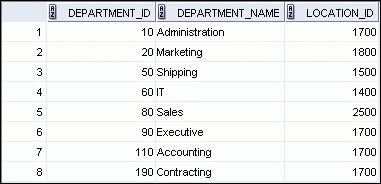


**EMPLOYEES**

**DEPARTMENTS**

**…**

**…**



## 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
* To avoid a Cartesian product, always include a valid join condition in a WHERE clause.

## Generating a Cartesian Product

**Cartesian product:**

**20**

**x 8 = 160 rows**

**EMPLOYEES**

**(20**

**rows**

**)**

**DEPARTMENTS**

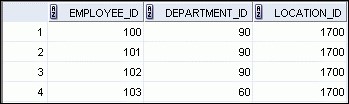
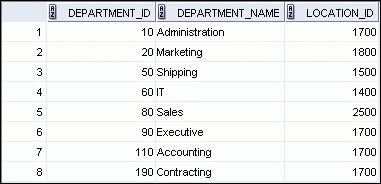
**(8**

**rows**

**)**

**…**

**…**



## Types of Oracle-Proprietary Joins

* Equijoin
* Nonequijoin
* Outer join
* Self-join

## Joining Tables Using Oracle Syntax

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

|  |  |
| --- | --- |
| **SELECT** | ***table1.column, table2.column*** |
| **FROM** | ***table1, table2*** |
| **WHERE** | ***table1.column1* = *table2.column2;*** |

* Write the join condition in the WHERE clause.
* Prefix the column name with the table name when the same column name appears in more than one table.

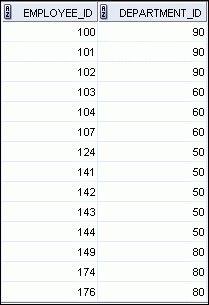
## Qualifying Ambiguous Column Names

* 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 aliases give 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.

## Equijoins



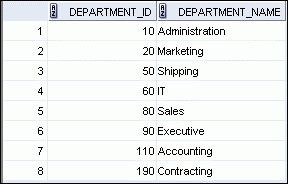
**EMPLOYEES**

**DEPARTMENTS**

**Foreign key**

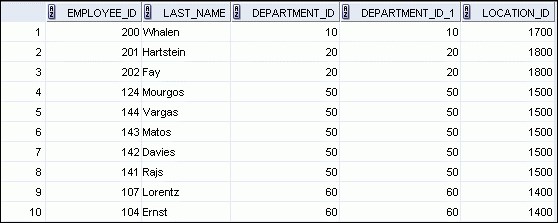
**Primary key**

**…**



## Retrieving Records with Equijoins

|  |  |
| --- | --- |
| **SELECT e.employee\_id, e.last\_name, e.department\_id,**  **d.department\_id, d.location\_id FROM employees e, departments d** | |
| **WHERE e.department\_id = d.department\_id;** |  |



**…**

## Retrieving Records with Equijoins: Example

|  |
| --- |
| **SELECT d.department\_id, d.department\_name,**  **d.location\_id, l.city**  **FROM departments d, locations l**  **WHERE d.location\_id = l.location\_id;** |



## Additional Search Conditions Using the AND Operator

**SELECT d.department\_id, d.department\_name, l.city**

**FROM departments d, locations l**

**WHERE d.location\_id = l.location\_id**

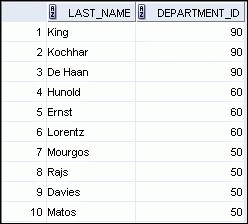
**AND d.department\_id IN (20, 50);**



## Joining More than Two Tables

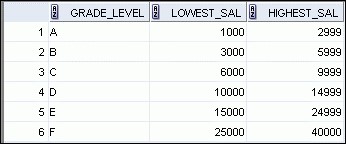
### EMPLOYEES DEPARTMENTS LOCATIONS

|  |  |
| --- | --- |
|  |  |
|  |

**…**

To join *n* tables together, you need a minimum of n–1 join conditions. For example, to join three tables, a minimum of two joins is required.

## Nonequijoins



**EMPLOYEES**

**JOB\_GRADES**

**…**

**JOB\_GRADES**

**table defines**

**LOWEST\_SAL**

**and**

**HIGHEST\_SAL**

**range of values for**

**each**

**GRADE\_LEVEL**

**. Hence, the**

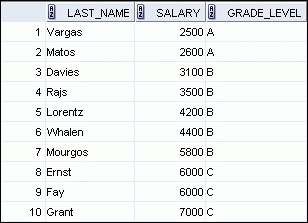
**GRADE\_LEVEL**

**column can be used to**

**assign grades to each employee.**

## Retrieving Records with Nonequijoins

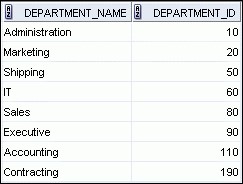
|  |  |  |
| --- | --- | --- |
| **SELECT e.last\_name, e.salary, j.grade\_level FROM employees e, job\_grades j** | | |
| **WHERE** | **e.salary**  **BETWEEN j.lowest\_sal AND j.highest\_sa** | **l;** |



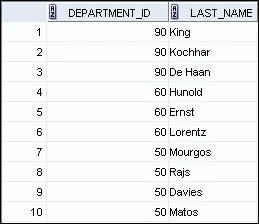
**…**

## Returning Records with No Direct Match with Outer Joins

### DEPARTMENTS EMPLOYEES



**…**



**There are no employees in department 190.**

## Outer Joins: Syntax

* You use an outer join to see rows that do not meet the join condition.
* The outer join operator is the plus sign (+).

**SELECT *table1.column, table2.column***

**FROM *table1, table2***

**WHERE *table1.column(+)* = *table2.column;***

**SELECT *table1.column, table2.column***

**FROM *table1, table2***

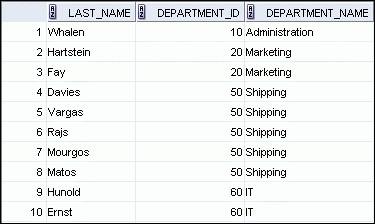
**WHERE *table1.column* = *table2.column(+);***

## Using Outer Joins

**SELECT e.last\_name, e.department\_id, d.department\_name**

**FROM employees e, departments d**

**WHERE e.department\_id(+) = d.department\_id ;**



**…**

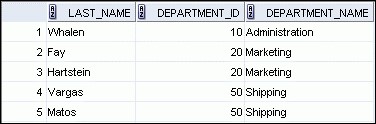


## Outer Join: Another Example

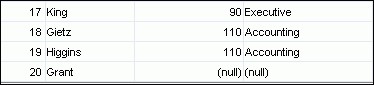
**SELECT e.last\_name, e.department\_id, d.department\_name**

**FROM employees e, departments d**

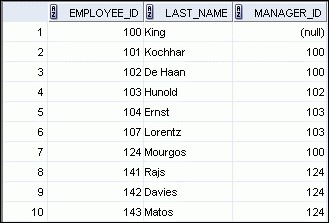
**WHERE e.department\_id = d.department\_id(+) ;**



**…**



## Joining a Table to Itself



**EMPLOYEES(WORKER)**

**EMPLOYEES(MANAGER)**

**…**

**…**



**MANAGER\_ID in the WORKER table is equal to EMPLOYEE\_ID in the MANAGER table.**

## Self-Join: Example

|  |
| --- |
| **SELECT worker.last\_name || ' works for ' || manager.last\_name**  **FROM employees worker, employees manager**  **WHERE worker.manager\_id = manager.employee\_id ;** |

