INF3707 - Database Design & Implementation

Summary 2014

Oracle 10g - Introduction to SQL (Part 1 & 2)

Ver 1.00 - April 2014

Ron Barnard

Retrieving data using the SQL select statement

Objectives

- List the capabilities of SQL SELECT statements
- Execute a basic SELECT statement
- Differentiate between SQL statements and *i*SQL*Plus commands

---

Content

Selecting All Columns

You can display all columns of data in a table by using an asterisk * after the SELECT keyword.

```
SELECT *
FROM departments;
```

You can also display all columns in a table by listing all the columns after the SELECT keyword.

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

Selecting Specific Columns

You can use the SELECT statement to display specific columns of a table by specifying the column names, separated by commas. Specify the columns that you want, in the order that you want them to appear in the output.

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

Writing SQL Statements

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

Column Heading Defaults

- Character and Date column headings are left aligned.
- Number column headings are right-aligned.
- Default heading display: Uppercase
- Can override the column heading display with an Alias.

Arithmetic Expressions

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

Available arithmetic operators - + Add, - Subtract, * Multiply, / Divide.

You can use arithmetic operators in any clause of an SQL statement, except the FROM clause.
DATE and TIMESTAMP data types can only use the ADD and SUBTRACT operators.

Using Arithmetic Operators

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

The output also displays a "SALARY + 300" column.
The calculated column is NOT a new column in the EMPLOYEES table, it is for display only.
Blank spaces before and after the arithmetic operator are ignored.

Operator Precedence

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

Null Values

If a row lacks a data value for a particular column, that value is said to be *null* or to contain a null.
A null is a value that is unavailable, unassigned, unknown, or inapplicable.
A null is not the same as a zero or a space. Zero is a number, and a space is a character.

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

Null Values in Arithmetic Expressions

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

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.
- Uppercase by default.

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;
```

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

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

Literal Character Strings

- A literal is a character, a number, or a date that is included in the SELECT statement, and that is not a column name or a column alias.
- Literal strings of free-format text can be included in the query result, and are treated the same as a column in the SELECT list.
- Date and character literal values must be enclosed by single quotation marks, number literals need not be enclosed.
- Each character string is output once for each row returned.

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

Alternative Quote (q) Operator

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

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

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

Duplicate Rows

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

To eliminate duplicate rows in the result, include the DISTINCT keyword in the SELECT clause immediately after the SELECT keyword.

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

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

Displaying Table Structure

You can display the structure of a table by using the DESCRIBE command.

The command displays the column names and data types, and it shows you whether a column *must* contain data (that is, whether the column has a NOT NULL constraint).

DESC[RIBE] *tablename*

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

Summary

In this lesson, you should have learned how to retrieve data from a database table with the SELECT statement.

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

In the syntax:

SELECT	is a list of one or more columns
*	selects all columns
DISTINCT	suppresses duplicates
column|*expression*	selects the named column or the expression
alias	gives selected columns different headings
FROM *table*	specifies the table containing the columns

_i_SQL*Plus

*i*SQL*Plus is an execution environment that you can use to send SQL statements to the database server and to edit and save SQL statements. Statements can be executed from the SQL prompt or from a script file.

--ooOoo--

Restricting and Sorting data

Objectives

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

---

Content

Limiting Rows Using a Selection

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

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

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

```
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 by single quotation marks. (Number constants should not be enclosed by single quotes).
- Character values are case-sensitive, and date values are format-sensitive.
- The default date format is DD-MON-RR.

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

Comparison Conditions

Comparison conditions are used in conditions that compare one expression to another value or expression.

=	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

Examples -

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

An alias cannot be used in the WHERE clause.
The symbols != and ^= can also represent the *not equal to* condition.

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

Using the BETWEEN Condition

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

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

Values that are specified with the BETWEEN condition are inclusive. You must specify the lower limit first.

You can also use the BETWEEN condition on character values:

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

Using the IN Condition

The IN membership condition is used to test for values in a list.
The IN condition is also known as the *membership condition.*

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

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

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

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

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

Using the LIKE Condition

Use the LIKE condition 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%' ;
```

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

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

You can combine pattern matching characters -

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

This will return the names of all employees whose last names have the letter o as the second character.

ESCAPE Option -

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

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

The backslash causes the following character to be interpreted literally.

Using the NULL Conditions

The NULL conditions include the IS NULL condition and the IS NOT NULL condition.
The IS NULL condition tests for nulls. A null value means the value is unavailable, unassigned, unknown, or inapplicable.

Therefore, you cannot test with = because a null cannot be equal or unequal to any value. The example retrieves the last names and managers of all employees who do not have a manager.

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

Logical Conditions

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 following condition is false

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

Using the AND Operator

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

AND requires both conditions to be true for any record to be selected.
All character searches are case-sensitive. No rows are returned if 'MAN' is not uppercase.
Character strings must be enclosed by quotation marks.

Using the OR Operator

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

OR requires either condition to be true for any record to be selected.

Using the NOT Operator

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

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

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

Rules of Precedence

Precedence	Operator Type
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

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

Using the ORDER BY Clause

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

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

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

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

Sorting

Default Ordering of Data -

The default sort order is ascending -

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

Examples -

- To reverse the order in which rows are displayed, specify the DESC keyword after the column name in the ORDER BY clause. The example sorts the result by the most recently hired employee.
- You can use a column alias in the ORDER BY clause. The slide example sorts the data by annual salary.
- You can sort query results by more than one column. The sort limit is the number of columns in the given table. In the ORDER BY clause, specify the columns and separate the column names using commas. If you want to reverse the order of a column, specify DESC after its name.

Sorting in descending order -

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

Sorting by column alias -

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

Sorting by multiple columns -

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

Substitution Variables

The examples so far have been hard-coded. In a finished application, the user would trigger the report, and the report would run without further prompting. The range of data would be predetermined by the fixed WHERE clause.

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

Use substitution variables to -

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

Note: Use colon : NOT ampersand &.

Using the & (:) Substitution Variable -

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

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

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;
```

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

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
```

The defined variable is automatically substituted in the SELECT statement.

Using the VERIFY Command

```
SET VERIFY ON

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

Setting SET VERIFY ON force display of the text of a command before and after it replaces substitution variables with values.

Summary

In this lesson, you should have learned how to:

- Use the WHERE clause to restrict rows of output:
  - Use the comparison conditions
  - Use the BETWEEN, IN, LIKE, and NULL conditions
  - Apply the logical AND, OR, and NOT operators

- Use the ORDER BY clause to sort rows of output:

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

- Use ampersand substitution in *i*SQL*Plus to restrict and sort output at run time.

- By using the *i*SQL*Plus substitution variables, you can add flexibility to your SQL statements. You can query users at run time and enable them to specify criteria.

--ooOoo-

Using single row functions to customize output

Objectives

- Describe various types of functions that are available in SQL
- Use character, number, and date functions in SELECT statements
- Describe the use of conversion functions

Content

SQL Functions

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

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

SQL functions sometimes take arguments and always return a value.

Note: Most of the functions that are described in this section are specific to the Oracle version of SQL.

There are two types of functions -

- **Single-row functions** - These functions operate on single rows only and return one result per row. There are different types of single-row functions. This section covers the following ones -
  - Character
  - Number
  - Date
  - Conversion
  - General

- **Multiple-row functions** - Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as *group functions* (covered in a later section).

Single-Row Functions

- Manipulate data items;
- Accept arguments and return one value. An argument can be one of the following -
  - User-supplied constant;
  - Variable value;
  - Column name;
  - Expression;
- Act on each row that is returned;
- Return one result per row;
- May modify the data type;
- Can be nested;
- Possibly returning a data value of a different type than the one that is referenced;
- Can be used in SELECT, WHERE, and ORDER BY clauses.

1-L3 - Using single row functions to customize output

This section covers the following single-row functions -

- **Character functions**: Accept character input and can return both character and number values;
- **Number functions**: Accept numeric input and return numeric values;
- **Date functions**: Operate on values of the DATE data type (All date functions return a value of DATE data type except the MONTHS_BETWEEN function, which returns a number);
- **Conversion functions**: Convert a value from one data type to another;
- **General functions** -
  - NVL;
  - NVL2;
  - NULLIF;
  - COALESCE;
  - CASE;
  - DECODE.

Character Functions

Character functions can be divided into the following -

- **Case-manipulation functions** -

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

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

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

- **Character-manipulation functions** -

  - **CONCAT** - Joins values together (You are limited to using two parameters with CONCAT);
  - **SUBSTR** - Extracts a string of determined length;
  - **LENGTH** - Shows the length of a string as a numeric value;
  - **INSTR** - Finds the numeric position of a named character;
  - **LPAD** - Pads the character value right-justified;
  - **RPAD** - Pads the character value left-justified;
  - **TRIM** - Trims heading or trailing characters (or both) from a character string (If *trim_character* or *trim_source* is a character literal, you must enclose it in single quotation marks.)

```
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';
```

Number Functions

Number functions accept numeric input and return numeric values.

- ROUND(*column*|*expression*, *n*) - Rounds the column, expression, or value to *n* decimal places or, if *n* is omitted, no decimal places (If *n* is negative, numbers to left of the decimal point are rounded);

```
SELECT ROUND(45.923,2), ROUND(45.923,0),
ROUND(45.923, -1)
FROM DUAL;                                          45.92  46  50
```

- TRUNC(*column*|*expression*, *n*) - Truncates the column, expression, or value to *n* decimal places or, if *n* is omitted, *n* defaults to zero;

```
SELECT TRUNC(45.923,2), TRUNC(45.923),
TRUNC(45.923,-1)
FROM DUAL;                                          45.92  45  40
```

- MOD(*m*,*n*) - Returns the remainder of *m* divided by *n.*

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

Note: The MOD function is often used to determine if a value is odd or even.

DUAL Table

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

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

Working with Dates

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

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

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

When a record with a date column is inserted into a table, the *century* information is picked up from the SYSDATE function. However, when the date column is displayed on the screen, the century component is not displayed (by default).

The DATE data type always stores year information as a four-digit number internally: two digits for the century and two digits for the year. For example, the Oracle database stores the year as 1987 or 2004, and not just as 87 or 04.

SYSDATE Function

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

Display the current date using the DUAL table -

```
SELECT SYSDATE
FROM DUAL;
```

Arithmetic with Dates

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

You can perform the following operations:

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

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

Date Functions

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

- **MONTHS_BETWEEN(*date1, date2*)** - Finds the number of months between *date1* and *date2*. The result can be positive or negative. If *date1* is later than *date2*, the result is positive; if *date1* is earlier than *date2*, the result is negative.

  The non-integer part of the result represents a portion of the month.

- **ADD_MONTHS(*date, n*)** - Adds *n* number of calendar months to *date*. The value of *n* must be an integer and can be negative.

- **NEXT_DAY(*date, 'char'*)** - Finds the date of the next specified day of the week ('*char*') following *date*. The value of *char* may be a number representing a day or a character string.

- **LAST_DAY(*date*)** - Finds the date of the last day of the month that contains *date* .

- **ROUND(*date*[,'*fmt*'])** - Returns *date* rounded to the unit that is specified by the format model *fmt.* If the format model *fmt* is omitted, *date* is rounded to the nearest day.

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

Example -

Display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and last day of the hire month for all employees who have been employed for fewer than 70 months -

```
SELECT employee_id, hire_date,
MONTHS_BETWEEN (SYSDATE, hire_date) TENURE,
ADD_MONTHS (hire_date, 6) REVIEW,
NEXT_DAY (hire_date, 'FRIDAY'), LAST_DAY(hire_date)
FROM employees
WHERE MONTHS_BETWEEN (SYSDATE, hire_date) < 70;
```

The ROUND and TRUNC functions can be used for number and date values. When used with dates, these functions round or truncate to the specified format model. Therefore, you can round dates to the nearest year or month.

Example -

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

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

Using Date Functions

Assume SYSDATE = '25-JUL-03'.

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

Implicit Data Type Conversion

For assignments, the Oracle server can automatically convert the following -

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

For expression evaluation, the Oracle Server can automatically convert the following -

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

Explicit Data Type Conversion

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

- TO_CHAR()
- TO_NUMBER()
- TO_DATE()

Conversion Functions

Using the TO_CHAR Function with Dates

TO_CHAR(*date, 'format_model'*)

The format model -

- Must be enclosed by 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.

Displaying a Date in a Specific Format -

Previously, all Oracle date values were displayed in the DD-MON-YY format. You can use the TO_CHAR function to convert a date from this default format to one that you specify.

Guidelines -

- The format model must be enclosed by single quotation marks and is case-sensitive.
- The format model can include any valid date format element. Be sure to separate the date value from the format model by a comma.

- The names of days and months in the output are automatically padded with blanks.
- To remove padded blanks or to suppress leading zeros, use the fill mode *fm* element.

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

Elements of the Date Format Model

Element	Result
YYYY	Full year in numbers
YEAR	Year spelled out (In English)
MM	Two-digit value for 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

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

Using the TO_CHAR Function with Numbers

TO_CHAR(*number*, '*format_model*')

Format elements that can be used 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 thousands indicator

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

Nesting Functions

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

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

General Functions

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

- NVL (expr1, expr2) - Converts a null value to an actual value. *expr1* is the source value or expression that may contain a null. *expr2* is the target value for converting the null.

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

(Converts null values of commission_pct to zero)

- NVL2 (expr1, expr2, expr3) - If expr1 is not null, NVL2 returns expr2. If expr1 is null, NVL2 returns expr3. The argument expr1 can have any data type;

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

- NULLIF (expr1, expr2) - Compares two expressions and returns null if they are equal, returns the first expression if they are not equal;

```
SELECT first_name, LENGTH(first_name) "expr1",
      last_name, LENGTH(last_name) "expr2",
      NULLIF(LENGTH(first_name), LENGTH(last_name)) result
FROM employees;
```

- COALESCE (expr1, expr2, ..., expr*n*) - Returns the first non-null expression in the expression list.
  - The advantage of the COALESCE function over the NVL function is that the COALESCE function can take multiple alternate values;
  - All expressions must be of the same data type.

```
SELECT last_name,
      COALESCE(manager_id, commission_pct, -1) comm
FROM employees
ORDER BY commission_pct;
```

Conditional Expressions

Provide the use of IF-THEN-ELSE logic within a SQL statement.
Use two methods -

- CASE expression;
- DECODE function.

Using the CASE Expression

```
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;
```

Using the DECODE Function

The DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic that is used in various languages.

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

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

Summary

Single-row functions can be nested to any level. Single-row functions can manipulate the following:

- Character data: LOWER, UPPER, INITCAP, CONCAT, SUBSTR, INSTR, LENGTH
- Number data: ROUND, TRUNC, MOD
- Date data: MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, LAST_DAY, ROUND, TRUNC

Remember the following:

- Date values can also use arithmetic operators.
- Conversion functions can convert character, date, and numeric values: TO_CHAR, TO_DATE, TO_NUMBER
- There are several functions that pertain to nulls, including NVL, NVL2, NULLIF, and COALESCE.
- IF-THEN-ELSE logic can be applied within a SQL statement by using the CASE expression or the DECODE function.

SYSDATE and DUAL

SYSDATE is a date function that returns the current date and time. It is customary to select SYSDATE from a dummy table called DUAL.

--ooOoo--

Reporting aggregated data using the group functions

Objectives

- Identify the available group functions
- Describe the use of group functions
- Group data by using the GROUP BY clause
- Include or exclude grouped rows by using the HAVING clause.

Content

Group Functions

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

Types of Group Functions

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

Guidelines for Using Group Functions

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

Using the AVG and SUM Functions

You can use AVG, SUM, MIN, and MAX functions against columns that can store 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 the MAX and MIN functions for numeric, character, and date data types.

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

Using the Count Function

The COUNT function has three formats -

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

COUNT(*) returns the number of rows in a table that satisfy the criteria of the SELECT statement, including duplicate rows and rows containing null values in any of the columns.

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

If a WHERE clause is included in the SELECT statement, COUNT(*) returns the number of rows that satisfy the condition in the WHERE clause.

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

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

COUNT(DISTINCT *expr*) returns the number of unique, non-null values that are in the column identified by *expr*. Use the DISTINCT keyword to suppress the counting of any duplicate values in a column.

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

Group Functions and Null Values

All group functions ignore null values in the column.

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

The NVL function forces group functions to include null values.

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

Creating Groups of Data

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

In the syntax -

group_by_expression specifies columns whose values determine the basis for grouping rows

Guidelines -

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

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

The GROUP BY column does not have to be in the SELECT list. However, the results may not look meaningful -

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

Grouping by More than One Column

Sometimes you need to see results for groups within groups.

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

The EMPLOYEES table is grouped first by department number, and then by job title within that grouping. So the SUM function is applied to the salary column for all job IDs in each department number group.

Illegal Queries using Group Functions

Whenever you use a mixture of individual items (DEPARTMENT_ID) and group functions (COUNT) in the same SELECT statement, you must include a GROUP BY clause that specifies the individual items (in this case, DEPARTMENT_ID). If the GROUP BY clause is missing, then the error message “not a single-group group function” appears and an asterisk (*) points to the offending column.

- 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.

Restricting Group Results

In the same way that you use the WHERE clause to restrict the rows that you select, you use the HAVING clause to restrict groups.

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

- Rows are grouped.
- The group function is applied.
- Groups matching the HAVING clause are displayed.

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

Nesting Group Functions

Group functions can be nested to a depth of two.

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

Summary

In this lesson, you should have learned how to:

- Use the group functions COUNT, MAX, MIN, and AVG
- Write queries that use the GROUP BY clause
- Write queries that use the HAVING clause

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

Several group functions are available in SQL, such as the following:

AVG, COUNT, MAX, MIN, SUM, STDDEV, and VARIANCE

You can create subgroups by using the GROUP BY clause. Groups can be restricted using the HAVING clause.

Place the HAVING and GROUP BY clauses after the WHERE clause in a statement. The order of the HAVING and GROUP clauses following the WHERE clause is not important. Place the ORDER BY clause last.

The Oracle server evaluates the clauses in the following order:

1. If the statement contains a WHERE clause, the server establishes the candidate rows.
2. The server identifies the groups that are specified in the GROUP BY clause.
3. The HAVING clause further restricts result groups that do not meet the group criteria in the HAVING clause.

Note: For a complete list of the group functions, see *Oracle SQL Reference*.

--ooOoo--

Displaying Data from Multiple Tables

Objectives

- 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.

Content

Obtaining Data from Multiple Tables

Sometimes you need to use data from more than one table. You need to link the tables and access data from both of them.

Types of Joins

- Cross joins;
- Natural joins;
- USING clause;
- Full (or two-sided) outer joins;
- Arbitrary join conditions for outer joins.

```
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];
```

Creating Natural Joins

- The NATURAL JOIN 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.

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

location_id is the only column of the same name in both tables.

Creating Joins with the USING Clause

- If several columns have the same names but the data types do not match, the NATURAL JOIN clause can be modified with the USING clause to specify the columns that should be used for an equijoin.
- Use the USING clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns.
- The NATURAL JOIN and USING clauses are mutually exclusive

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

The columns that are referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement eg: in the WHERE clause - will cause an error "ORA25154: column part of USING clause cannot have qualifier".

Joining Column Names

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

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

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

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance. If there are no common column names between the two tables, there is no need to qualify the columns. However, using the table prefix improves performance, because you tell the Oracle server exactly where to find the columns.
- Use column aliases to distinguish columns that have identical names but reside in different tables.
- Do not use aliases on columns that are identified in the USING clause and listed elsewhere in the SQL statement.

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

Qualifying column names with table names can be very time consuming, particularly if table names are lengthy. You can use *table aliases* instead of table names. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore using less memory.

Notice how table aliases are identified in the FROM clause in the example. The table name is specified in full, followed by a space and then the table alias. The EMPLOYEES table has been given an alias of e, and the DEPARTMENTS table has an alias of d.

Guidelines -

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

Creating 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.
- Lets you specify join conditions separate from any search or filter conditions in the WHERE clause.
- You can also use the ON clause to join columns that have different names.

```
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);
```

Self-Joins Using the ON Clause

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

- Find Lorentz in the EMPLOYEES table by looking at the LAST_NAME column.
- Find the manager number for Lorentz by looking at the MANAGER_ID column. Lorentz's manager number is 103.
- Find the name of the manager with EMPLOYEE_ID 103 by looking at the LAST_NAME column. Hunold's employee number is 103, so Hunold is Lorentz's manager.

In this process, you look in the table twice. The first time you look in the table to find Lorentz in the LAST_NAME column and MANAGER_ID value of 103. The second time you look in the EMPLOYEE_ID column to find 103 and the LAST_NAME column to find Hunold.

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

Applying Additional Conditions to a Join

You can apply additional conditions to the join.

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

```
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;

WHERE e.manager_id = 149;
```

Creating Three-Way Joins with the ON Clause

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

```
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;
```

Non-Equijoins

A non-equijoin is a join condition containing something other than an equality operator.

The relationship between the EMPLOYEES table and the JOB_GRADES table is an example of a non-equijoin. A relationship between the two tables is that the SALARY column in the EMPLOYEES table must be between the values in the LOWEST_SALARY and HIGHEST_SALARY columns of the JOB_GRADES table. The relationship is obtained using an operator other than equality (=).

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

Note: Other conditions (such as <= and >=) can be used, but BETWEEN is the simplest.

Outer Joins

If a row does not satisfy a join condition, the row does not appear in the query result. For example, in the equijoin condition of EMPLOYEES and DEPARTMENTS tables, department ID 190 does not appear because there are no employees with that department ID recorded in the EMPLOYEES table. Instead of seeing 20 employees in the result set, you see 19 records. To return the department record that does not have any employees, you can use an outer join.

INNER vs 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) tables 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.

Joining tables with the NATURAL JOIN, USING, or ON clauses results in an inner join.

Any unmatched rows are not displayed in the output. To return the unmatched rows, you can use an outer join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other table satisfy the join condition.

There are three types of outer joins -

- LEFT OUTER;
- RIGHT OUTER;
- FULL OUTER

LEFT OUTER JOIN

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

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

RIGHT OUTER JOIN

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

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

FULL OUTER JOIN

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

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

Cartesian Products

A Cartesian product is formed when -

- A join condition is omitted;
- A join condition is invalid;
- All rows in the first table are joined to all rows in the second table.

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

A Cartesian product tends to generate a large number of rows, and the result is rarely useful.

You should always include a valid join condition unless you have a specific need to combine all rows from all tables.

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

The result of a Cartesian Product is the product of the rows of both tables - If Table A has 20 rows, and Table B has 8 rows, the Cartesian Product is 20 x 8 = 160 rows.

The CROSS JOIN clause produces the Cartesian product between 2 tables -

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

Summary

There are multiple ways to join tables.

Types of Joins

- Equijoins
- Non-equijoins
- Outer joins
- Self-joins
- Cross joins
- Natural joins
- Full (or two-sided) outer joins

Cartesian Products

A Cartesian product results in a display of all combinations of rows. This is done by either omitting the WHERE clause or specifying the CROSS JOIN clause.

Table Aliases

- Table aliases speed up database access.
- Table aliases can help to keep SQL code smaller by conserving memory.

--ooOoo--

Using Sub-queries to Solve Queries

Objectives

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

Content

Using a Subquery to Solve a Problem

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

To solve this problem, you need *two* queries: one to find how much Abel earns, and a second query to find who earns more than that amount. You can solve this problem by combining the two queries, placing one query *inside* the other query.

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

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

- The subquery (inner query) executes once before the main query (outer query).
- The result of the subquery is used by the main query.
- You can place the subquery in a number of SQL clauses, including the following -
    - WHERE clause
    - HAVING clause
    - FROM clause

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

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

1-L6 - Using Sub-queries to Solve Queries

Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- The ORDER BY clause in the subquery is not needed.
- Use single-row operators with single-row subqueries, and use multiple-row operators with multiple-row subqueries.

Types of Subqueries

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

Note: There are also multiple-column subqueries, which are queries that return more than one column from the inner SELECT statement. (Covered in Part II).

Single-Row Subqueries

- Return only one row from the inner SELECT statement;
- Use single-row comparison operators -
  - >= , < , <= , = , <> , > .

```
SELECT last_name, job_id, salary
FROM employees
WHERE job_id =
      (SELECT job_id
      FROM employees
      WHERE employee_id = 141)
AND salary >
      (SELECT salary
      FROM employees
      WHERE employee_id = 143);
```

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

Using Group Functions in a Subquery

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

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

The HAVING Clause with Subqueries

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

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

Errors with Subqueries

One common error with subqueries occurs when more than one row is returned for a singlerow subquery.

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

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

Multiple-Row Subqueries

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

Operator	Meaning
IN	Equal to any member in the list.
ANY	Compare value to each value returned by the subquery
ALL	Compare value to every value returned by the subquery

Using the IN Operator in Multiple-Row Subqueries

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

Using the ANY Operator in Multiple-Row Subqueries

The ANY operator (and its synonym, the SOME operator) compares a value to *each* value returned by a subquery

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

```
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';
```

Using the ALL Operator in Multiple-Row Subqueries

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

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

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

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

Null Values in a Subquery

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

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

The NOT IN operator is equivalent to <> ALL.

Notice that the null value as part of the results set of a subquery is not a problem if you use the IN operator. The IN operator is equivalent to =ANY.

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

Alternatively, a WHERE clause can be included in the subquery to display all employees who do not have any subordinates -

```
SELECT last_name FROM employees
WHERE employee_id NOT IN
      (SELECT manager_id
      FROM employees
      WHERE manager_id IS NOT NULL);
```

Summary

A subquery is a SELECT statement that is embedded in a clause of another SQL statement.
Subqueries are useful when a query is based on a search criterion with unknown intermediate values.

Subqueries have the following characteristics:

- Can pass one row of data to a main statement that contains a single-row operator, such as =, <>, >, >=, <, or <=
- Can pass multiple rows of data to a main statement that contains a multiple-row operator, such as IN
- Are processed first by the Oracle server, after which the WHERE or HAVING clause uses the results
- Can contain group functions.

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

--ooOoo--

Using the Set Operators

Objectives

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

Content

Set Operators

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

Operator	Returns
UNION	All distinct rows selected by either query.
UNION ALL	All rows selected by either query, including all duplicates.
INTERSECT	All distinct rows selected by both queries.
MINUS	All distinct rows that are selected by the first SELECT statement and not selected in the second SELECT statement

All set operators have equal precedence. If a SQL statement contains multiple set operators, the Oracle server evaluates them from left (top) to right (bottom) if no parentheses explicitly specify another order. You should use parentheses to specify the order of evaluation explicitly in queries that use the INTERSECT operator with other set operators.

UNION Operator

The UNION operator returns all rows that are selected by either query, after eliminating duplications.
Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows.

- The number of columns and the data types of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- UNION operates over all of the columns being selected.
- NULL values are not ignored during duplicate checking.
- The IN operator has a higher precedence than the UNION operator.
- By default, the output is sorted in ascending order of the first column of the SELECT clause.

Using the UNION Operator

The UNION operator eliminates any duplicate records. If records that occur in both the EMPLOYEES and the JOB_HISTORY tables are identical, the records are displayed only once. Observe in the output shown in the example that the record for the employee with the EMPLOYEE_ID 200 appears twice because the JOB_ID is different in each row.

```
SELECT employee_id, job_id
FROM employees
UNION
SELECT employee_id, job_id
FROM job_history;
```

UNION ALL Operator

Use the UNION ALL operator to return all rows from multiple queries, including duplications.

The guidelines for UNION and UNION ALL are the same, with the following two exceptions that pertain to UNION ALL -

- Unlike UNION, duplicate rows are not eliminated and the **output is not sorted** by default;
- The DISTINCT keyword cannot be used.

Using the UNION ALL Operator

The UNION ALL operator does not eliminate duplicate rows. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates.

```
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;
```

INTERSECT Operator

Use the INTERSECT operator to return all rows that are common to multiple queries.

- The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- INTERSECT does not ignore NULL values.

Using the INTERSECT Operator

The query returns only the records that have the same values in the selected columns in both tables.

```
SELECT employee_id, job_id
FROM employees
INTERSECT
SELECT employee_id, job_id
FROM job_history;
```

MINUS Operator

Use the MINUS operator to return rows returned by the first query that are not present in the second query (the first SELECT statement MINUS the second SELECT statement).

- The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- All of the columns in the WHERE clause must be in the SELECT clause for the MINUS operator to work.

```
SELECT employee_id,job_id
FROM employees
MINUS
SELECT employee_id,job_id
FROM job_history;
```

Set Operator Guidelines

- The expressions in the select lists of the queries must match in number and data type. Queries that use UNION, UNION ALL, INTERSECT, and MINUS operators in their WHERE clause must have the same number and type of columns in their SELECT list.
- Parentheses can be used to alter the sequence of execution.
- The ORDER BY clause -
  - Can appear only at the very end of the statement;
  - Will accept the column name, an alias, or the positional notation;
- The column name or alias, if used in an ORDER BY clause, must be from the first SELECT list;
- Set operators can be used in subqueries.

Summary

- The UNION operator returns all rows selected by either query. Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows.
- Use the UNION ALL operator to return all rows from multiple queries. Unlike the case with the UNION operator, duplicate rows are not eliminated and the output is not sorted by default.
- Use the INTERSECT operator to return all rows that are common to multiple queries.
- Use the MINUS operator to return rows returned by the first query that are not present in the second query.
- Remember to use the ORDER BY clause only at the very end of the compound statement.
- Make sure that the corresponding expressions in the SELECT lists match in number and data type.

--ooOoo--

Manipulating Data

Objectives

- Describe each data manipulation language (DML) statement
- Insert rows into a table
- Update rows in a table
- Delete rows from a table
- Control transactions with COMMIT, SAVEPOINT, and ROLLBACK statements.

Content

Data Manipulation Language (DML)

Data manipulation language (DML) is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a *transaction*.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations -

- decrease the savings account;
- increase the checking account;
- and record the transaction in the transaction journal.

  The Oracle server must guarantee that all three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

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

Because you can insert a new row that contains values for each column, the column list is not required in the INSERT clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column.

For clarity, use the column list in the INSERT clause.

Enclose character and date values in single quotation marks; it is not recommended that you enclose numeric values in single quotation marks.

Number values should not be enclosed in single quotation marks, because implicit conversion may take place for numeric values that are assigned to NUMBER data type columns if single quotation marks are included.

```
INSERT INTO departments(department_id,
      department_name, manager_id, location_id)
VALUES (70, 'Public Relations', 100, 1700);
```

Inserting Rows with Null Values

Implicit method - Omit the column from the column list.

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

Inserting Special Values

You can use functions to enter special values in your table.

SYSDATE function - current date and time.
USER function - records the current username.

Inserting Specific Date Values

The DD-MON-YY format is usually used to insert a date value. With this format the century defaults to the current century. Because the date also contains time information, the default time is midnight (00:00:00).

If a date must be entered in a format other than the default format (for example, with another century or a specific time), you must use the TO_DATE function.

```
INSERT INTO employees
VALUES (114,
      'Den', 'Raphealy',
      'DRAPHEAL', '515.127.4561',
      TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
      'AC_ACCOUNT', 11000, NULL, 100, 30);
```

Creating a Script

You can save commands with substitution variables to a file and execute the commands in the file.

Run the script file and you are prompted for input for each of the & substitution variables.

After entering a value for the substitution variable, click the Continue button. The values that you input are then substituted into the statement.

This enables you to run the same script file over and over but supply a different set of values each time you run it.

```
INSERT INTO departments
      (department_id, department_name, location_id)
VALUES (&department_id, '&department_name',&location);
```

Note: Use colon (:), instead of ampersand (&).

Copying Rows from Another Table

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

The number of columns and their data types in the column list of the INSERT clause must match the number of values and their data types in the subquery.

To create a copy of the rows of a table, use SELECT * in the subquery.

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

Changing Data in a Table

Updating Rows

You can modify existing rows by using the UPDATE statement.

The UPDATE statement modifies specific rows if the WHERE clause is specified.
If you omit the WHERE clause, all the rows in the table are modified.

```
UPDATE employees
SET department_id = 70
WHERE employee_id = 113;
```

Updating Two Columns with a Subquery

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

```
UPDATE employees
SET   job_id = (SELECT job_id
                FROM employees
                WHERE employee_id = 205),
      salary = (SELECT salary
                FROM employees
                WHERE employee_id = 205)
WHERE employee_id = 114;
```

Updating Rows based on another Table

You can use subqueries in UPDATE statements to update rows 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);
```

Removing Rows from a Table

You can remove existing rows by using the DELETE statement.

You can delete specific rows by specifying the WHERE clause in the DELETE statement.
If you omit the WHERE clause, all rows in the table are deleted.

```
DELETE FROM departments
WHERE department_name = 'Finance';
```

Deleting Rows based on another Table

You can use subqueries to delete 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

A more efficient method of emptying a table is with the TRUNCATE statement.

You can use the TRUNCATE statement to quickly remove all rows from a table or cluster.

Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons -

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lesson.
- Truncating a table does not fire the delete triggers of the table.

```
TRUNCATE TABLE copy_emp;
```

Using a Subquery in an INSERT Statement

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

The select list of this subquery must have the same number of columns as the column list of the VALUES clause. Any rules on the columns of the base table must be followed if the INSERT statement is to work successfully. For example, you could not put in a duplicate employee ID or omit a value for a mandatory not-null column.

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

Database Transactions

The Oracle server ensures data consistency based on transactions. Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.

Transactions consist of DML statements that make up one consistent change to the data. For example, a transfer of funds between two accounts should include the debit to one account and the credit to another account in the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

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.

Type	Description
Data manipulation language (DML)	Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work.
Data definitional language (DDL)	Consists of only one DDL statement.
Data control language (DCL)	Consists of only one DCL statement.

A transaction begins when the first DML statement is encountered and ends when one of the following occurs -

- A COMMIT or ROLLBACK statement is issued.
- A DDL statement, such as CREATE, is issued.
- A DCL statement is issued.
- The user exits *i*SQL*Plus.
- A machine fails or the system crashes.

After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.

Advantages of COMMIT and ROLLBACK -

With the COMMIT and ROLLBACK statements, you have control over making changes to the data permanent.

Controlling Transactions

You can control the logic of transactions by using the COMMIT, SAVEPOINT, and ROLLBACK statements.

Note: SAVEPOINT is not ANSI standard SQL.

Statement	Description
COMMIT	Ends the current transaction by making all pending data changes permanent.
SAVEPOINT *name*	Marks a savepoint within the current transaction.
ROLLBACK	ROLLBACK ends the current transaction by discarding all pending data changes.
ROLLBACK TO *SAVEPOINT name*	ROLLBACK TO SAVEPOINT rolls back the current transaction to the specified savepoint, thereby discarding any changes and or savepoints that were created after the savepoint to which you are rolling back. If you omit the TO SAVEPOINT clause, the ROLLBACK statement rolls back the entire transaction. Because savepoints are logical, there is no way to list the savepoints that you have created.

Rolling Back Changes to a Marker

You can create a marker in the current transaction by using the SAVEPOINT statement, which divides the transaction into smaller sections. You can then discard pending changes up to that marker by using the ROLLBACK TO SAVEPOINT statement.

If you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

```
UPDATE...
SAVEPOINT update_done;
      Savepoint created.

INSERT...
ROLLBACK TO update_done;
      Rollback complete.
```

Implicit Transaction Processing

An automatic commit occurs under the following circumstances -

- DDL statement is issued;
- DCL statement is issued;
- Normal exit from *i*SQL*Plus, without explicitly issuing COMMIT or ROLLBACK statements.

An automatic rollback occurs under an abnormal termination of *i*SQL*Plus or a system failure.

When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to their state at the time of the last commit. In this way, the Oracle server protects the integrity of the tables. Closing the window is interpreted as an abnormal exit.

Committing Changes

Every data change made during the transaction is temporary until the transaction is committed.

The state of the data before COMMIT or ROLLBACK statements are issued can be described as follows -

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current user can review the results of the data manipulation operations by querying the tables.
- Other users cannot view the results of the data manipulation operations made by the current user. The Oracle server institutes read consistency to ensure that each user sees data as it existed at the last commit.
- The affected rows are locked; other users cannot change the data in the affected rows.

All pending changes are made permanent by using the COMMIT statement. State after a COMMIT statement is issued -

- Data changes are written to the database.
- The previous state of the data is no longer available with normal SQL queries.
- All users can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other users to perform new data changes.
- All savepoints are erased.

```
DELETE FROM employees
WHERE employee_id = 99999;

COMMIT;
      Commit complete.
```

Rolling Back Changes

Discard all pending changes by using the ROLLBACK statement, which results in the following -

- Data changes are undone.
- The previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;
      22 rows deleted.
ROLLBACK ;
      Rollback complete.
```

Summary

Manipulate data in the Oracle database by using -

Function	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table

How to control data changes by using -

COMMIT	Makes all pending changes permanent
ROLLBACK	Discards all pending data changes
SAVEPOINT	Is used to roll back to the savepoint marker.

The Oracle server guarantees a consistent view of data at all times.

--ooOoo--

Using DDL Statements to Create and Manage Tables

Objectives

Introduction to the data definition language (DDL) statements. You are taught the basics of how to create simple tables, alter them, and remove them. The data types available in DDL are shown, and schema concepts are introduced. Constraints are tied into this lesson. Exception messages that are generated from violating constraints during DML are shown and explained.

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

Content

Database Objects

An Oracle database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

- **Table:** Stores data;
- **View:** Subset of data from one or more tables;
- **Sequence:** Generates numeric values;
- **Index:** Improves the performance of some queries;
- **Synonym:** Gives alternative names to objects.

Naming Rules

You name database tables and columns according to the standard rules for naming any Oracle database object:

- Table names and column names must begin with a letter and be 1–30 characters long;
- Names must contain only the characters A–Z, a–z, 0–9, _ (underscore), $, and # (legal characters, but their use is discouraged);
- Names must not duplicate the name of another object owned by the same Oracle server user;
- Names must not be an Oracle server reserved word.
- Names are case-insensitive. For example, EMPLOYEES is treated as the same name as eMPloyees or eMpLOYEES.

CREATE TABLE Statement

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements, which are a subset of SQL statements used to create, modify, or remove Oracle database structures. These statements have an immediate effect on the database, and they also record information in the data dictionary.

To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects.

DEFAULT Option

When you define a table, you can specify that a column be given a default value by using the DEFAULT option. This option prevents null values from entering the columns if a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function (such as SYSDATE or USER), but the value cannot be the name of another column or a pseudocolumn (such as NEXTVAL or CURRVAL). The default expression must match the data type of the column.

```
CREATE TABLE hire_dates
      (id NUMBER(8),
      hire_date DATE DEFAULT SYSDATE);
```

Creating Tables

```
CREATE TABLE dept
       (deptno NUMBER(2),
       dname VARCHAR2(14),
       loc VARCHAR2(13),
       create_date DATE DEFAULT SYSDATE);
```

Because creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

Data Types

Data Type	Description
VARCHAR2(*size*)	Variable-length character data (A maximum *size* must be specified: minimum *size* is 1; maximum *size* is 4,000.)
CHAR [(*size*)]	Fixed-length character data of length *size* bytes (Default and minimum *size* is 1; maximum *size* is 2,000.)
NUMBER [(*p,s*)]	Number having precision *p* and scale *s* (The precision is the total number of decimal digits, and the scale is the number of digits to the right of the decimal point; the precision can range from 1 to 38, and the scale can range from –84 to 127.)
DATE	Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.
LONG	Variable-length character data (up to 2 GB).
CLOB	Character data (up to 4 GB).
RAW(*size*)	Raw binary data of length *size* (A maximum *size* must be specified: maximum *size* is 2,000.)
LONG RAW	Raw binary data of variable length (up to 2 GB).
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.

Datetime Data Types

Data Type	Description
TIMESTAMP	Enables the time to be stored as a date with fractional seconds. There are several variations of this data type.
INTERVAL YEAR TO MONTH	Enables time to be stored as an interval of years and months. Used to represent the difference between two datetime values in which the only significant portions are the year and month.
INTERVAL DAY TO SECOND	Enables time to be stored as an interval of days, hours, minutes, and seconds. Used to represent the precise difference between two datetime values.

TIMESTAMP Data Type

The TIMESTAMP data type is an extension of the DATE data type. It stores the year, month, and day of the DATE data type plus hour, minute, and second values. This data type is used for storing precise time values. The fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

```
CREATE TABLE new_employees
      (employee_id NUMBER,
      first_name VARCHAR2(15),
      last_name VARCHAR2(15),
      start_date TIMESTAMP(7));
```

Including Constraints

The Oracle server uses constraints to prevent invalid data entry into tables.

You can use constraints to do the following -

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables.

Constraint	Description
NOT NULL	Specifies that the column cannot contain a null value.
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table.
PRIMARY KEY	Uniquely identifies each row of the table.
FOREIGN KEY	Establishes and enforces a foreign key relationship between the column and a column of the referenced table.
CHECK	Specifies a condition that must be true.

Constraint Guidelines

- You can name a constraint, or the Oracle server generates a name by using the SYS_C*n* format;

- Create a constraint at either of the following times -
  - At the same time as the table is created;
  - After the table has been created;

- Define a constraint at the column or table level;
- View a constraint in the data dictionary.

Defining Constraints

- You can create the constraints at either the column level or table level.
- Constraints defined at the column level are included when the column is defined.
- Table-level constraints are defined at the end of the table definition and must refer to the column or columns on which the constraint pertains in a set of parentheses.

- NOT NULL constraints must be defined at the column level.
- Constraints that apply to more than one column must be defined at the table level.

Column-level constraint:

```
CREATE TABLE employees(
      employee_id NUMBER(6)
            CONSTRAINT emp_emp_id_pk PRIMARY KEY,
      first_name VARCHAR2(20),
      ...);
```

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));
```

NOT NULL Constraint

The NOT NULL constraint ensures that the column contains no null values. Columns without the NOT NULL constraint can contain null values by default. NOT NULL constraints must be defined at the column level.

UNIQUE Constraint

A UNIQUE key integrity constraint requires that every value in a column or set of columns (key) be unique - that is, no two rows of a table can have duplicate values in a specified column or set of columns. The column (or set of columns) included in the definition of the UNIQUE key constraint is called the *unique key*. If the UNIQUE constraint comprises more than one column, that group of columns is called a *composite unique key*.

UNIQUE constraints enable the input of nulls unless you also define NOT NULL constraints for the same

columns. In fact, any number of rows can include nulls for columns without NOT NULL constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite UNIQUE key) always satisfies a UNIQUE constraint.

UNIQUE constraints can be defined at the column level or table level. A composite unique key is created by using the table-level definition.

```
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));
```

PRIMARY KEY Constraint

A PRIMARY KEY constraint creates a primary key for the table. Only one primary key can be created for each table. The PRIMARY KEY constraint is a column or set of columns that uniquely identifies each row in a table. This constraint enforces uniqueness of the column or column combination and ensures that no column that is part of the primary key can contain a null value.

FOREIGN KEY Constraint

The FOREIGN KEY (or referential integrity) constraint designates a column or combination of columns as a foreign key and establishes a relationship between a primary key or a unique key in the same table or a different table.

- A foreign key value must match an existing value in the parent table or be NULL.
- Foreign keys are based on data values and are purely logical, rather than physical, pointers.
- FOREIGN KEY constraints can be defined at the column or table constraint level.
- A composite foreign key must be created by using the table-level definition.

```
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));
```

FOREIGN KEY Constraint: Keywords

The foreign key is defined in the child table, and the table containing the referenced column is the parent table. The foreign key is defined using a combination of the following keywords -

- FOREIGN KEY is used to define the column in the child table at the table-constraint level;
- REFERENCES identifies the table and column in the parent table;
- ON DELETE CASCADE indicates that when the row in the parent table is deleted, the dependent rows in the child table are also deleted;
- ON DELETE SET NULL converts foreign key values to null when the parent value is removed.

CHECK Constraint

The CHECK constraint defines a condition that each row must satisfy. The condition can use the same constructs as query conditions, with the following exceptions -

- References to the CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns;
- Calls to SYSDATE, UID, USER, and USERENV functions;
- Queries that refer to other values in other rows.

A single column can have multiple CHECK constraints that refer to the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

CHECK constraints can be defined at the column level or table level.

```
..., salary NUMBER(2)
CONSTRAINT emp_salary_min
CHECK (salary > 0),...
```

CREATE TABLE Example

```
CREATE TABLE employees
      ( employee_id NUMBER(6)
           CONSTRAINT emp_employee_id PRIMARY KEY
      , first_name VARCHAR2(20)
      , last_name VARCHAR2(25)
           CONSTRAINT emp_last_name_nn NOT NULL
      , email VARCHAR2(25)
           CONSTRAINT emp_email_nn NOT NULL
           CONSTRAINT emp_email_uk UNIQUE
      , phone_number VARCHAR2(20)
      , hire_date DATE
           CONSTRAINT emp_hire_date_nn NOT NULL
      , job_id VARCHAR2(10)
           CONSTRAINT emp_job_nn NOT NULL
      , salary NUMBER(8,2)
           CONSTRAINT emp_salary_ck CHECK (salary>0)
      , commission_pct NUMBER(2,2)
      , manager_id NUMBER(6)
      , department_id NUMBER(4)
           CONSTRAINT emp_dept_fk FOREIGN KEY
           REFERENCES departments (department_id));
```

Violating Constraints

When you have constraints in place on columns, an error is returned to you if you try to violate the constraint rule.

For example, if you attempt to update a record with a value that is tied to an integrity constraint, an error is returned.

Creating a Table by Using a Subquery

A second method for creating a table is to apply the AS *subquery* clause, which both creates the table and inserts rows returned from the subquery.

- The table is created with the specified column names, and the rows retrieved by the SELECT statement are inserted into the table;
- The column definition can contain only the column name and default value;
- If column specifications are given, the number of columns must equal the number of columns in the subquery SELECT list;
- If no column specifications are given, the column names of the table are the same as the column names in the subquery;
- The column data type definitions and the NOT NULL constraint are passed to the new table. The other constraint rules are not passed to the new table. However, you can add constraints in the column definition;
- Be sure to provide a column alias when selecting an expression. The expression SALARY*12 must be given an alias eg: ANNSAL. Without the alias, an error is generated.

ALTER TABLE Statement

Use the ALTER TABLE statement to -

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

Dropping a Table

The DROP TABLE statement removes the definition of an Oracle table. When you drop a table, the database loses all the data in the table and all the indexes associated with it.

- All data and structure in the table are deleted;
- Any pending transactions are committed;
- All indexes are dropped;
- All constraints are dropped;
- You *cannot* roll back the DROP TABLE statement.

```
DROP TABLE dept80;
      Table dropped.
```

Summary

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

CREATE TABLE

- Use the CREATE TABLE statement to create a table and include constraints.
- Create a table based on another table by using a subquery.

DROP TABLE

- Remove rows and a table structure.
- Once executed, this statement cannot be rolled back.

--ooOoo--

Creating Other Schema Objects

Objectives

Introduction to the view, sequence, synonym, and index objects. Taught the basics of creating and using views, sequences, and indexes.

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

Content

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 Sequence numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

What is a View

You can present logical subsets or combinations of data by creating views of tables. A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called *base tables*. The view is stored as a SELECT statement in the data dictionary.

Advantages of Views

- Views restrict access to the data because the view can display selected columns from the table.
- Views can be used to make simple queries to retrieve the results of complicated queries. For example, views can be used to query information from multiple tables without the user knowing how to write a join statement.
- Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- Views provide groups of users access to data according to their particular criteria.

Simple Views and Complex Views

There are two classifications for views: simple and complex. The basic difference is related to the DML (INSERT, UPDATE, and DELETE) operations -

- A simple view is one that -
  - Derives data from only one table;
  - Contains no functions or groups of data;
  - Can perform DML operations through the view.

- A complex view is one that -
  - Derives data from many tables;
  - Contains functions or groups of data;
  - Does not always allow DML operations through the view.

Creating a View

You can create a view by embedding a subquery in the CREATE VIEW statement.

```
CREATE VIEW empvu80
AS SELECT employee_id, last_name, salary
      FROM employees
      WHERE department_id = 80;
```

Retrieving Data from a View

```
SELECT *
FROM salvu50;
```

Modifying a View

With the OR REPLACE option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranting object privileges.

```
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;
```

Performing DML Operations on a View

You can perform DML operations on data through a view if those operations follow certain rules.

You can remove a row from a view, unless it contains any of the following -

- Group functions;
- A GROUP BY clause;
- The DISTINCT keyword;
- The pseudocolumn ROWNUM keyword.

You cannot modify data in a view if it contains -

- Group functions;
- A GROUP BY clause;
- The DISTINCT keyword;
- The pseudocolumn ROWNUM keyword;
- Columns defined by expressions.

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

- Group functions;
- A GROUP BY clause;
- The DISTINCT keyword;
- The pseudocolumn ROWNUM keyword;
- Columns defined by expressions;
- NOT NULL columns in the base tables that are not selected by the view.

Using the WITH CHECK OPTION Clause

It is possible to perform referential integrity checks through views. You can also enforce constraints at the database level. The view can be used to protect data integrity, but the use is very limited.

The WITH CHECK OPTION clause specifies that INSERTs and UPDATEs performed through the view cannot create rows that the view cannot select, and therefore it enables integrity constraints and data validation checks to be enforced on data being inserted or updated. If there is an attempt to perform DML operations on rows that the view has not selected, an error is displayed, along with the constraint name if that has been specified.

Denying DML Operations

You can ensure that no DML operations occur on your view by creating it with the WITH READ ONLY option.

```
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 use the DROP VIEW statement to remove a view. The statement removes the view definition from the database. Dropping views has no effect on the tables on which the view was based.

```
DROP VIEW view;
```

Sequences

A sequence is a database object that creates integer values. You can create sequences and then use them to generate numbers.

- You can define a sequence to generate unique values or to recycle and use the same numbers again.
- A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object because it can reduce the amount of application code needed to write a sequence-generating routine.
- Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.

```
CREATE SEQUENCE sequence
      [INCREMENT BY n]
      [START WITH n]
      [{MAXVALUE n | NOMAXVALUE}]
      [{MINVALUE n | NOMINVALUE}]
      [{CYCLE | NOCYCLE}]
      [{CACHE n | NOCACHE}];
```

```
CREATE SEQUENCE dept_deptid_seq
      INCREMENT BY 10
      START WITH 120
      MAXVALUE 9999
      NOCACHE
      NOCYCLE;
```

NEXTVAL and CURRVAL Pseudocolumns

After you create your sequence, it generates sequential numbers for use in your tables.
Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference *sequence*.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.

The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When you reference *sequence*.CURRVAL, the last value returned to that user's process is displayed.

Using a Sequence

```
INSERT INTO departments(department_id,
      department_name, location_id)
VALUES (dept_deptid_seq.NEXTVAL,
      'Support', 2500);
```

You can view the current value of the sequence -

```
SELECT dept_deptid_seq.CURRVAL
FROM dual;
```

Modifying a Sequence

If you reach the MAXVALUE limit for your sequence, no additional values from the sequence are allocated and you will receive an error indicating that the sequence exceeds the MAXVALUE. To continue to use the sequence, you can modify it by using the ALTER SEQUENCE statement.

```
ALTER SEQUENCE dept_deptid_seq
      INCREMENT BY 20
      MAXVALUE 999999
      NOCACHE
      NOCYCLE;
```

Indexes

Indexes are database objects that you can create to improve the performance of some queries. Indexes can also be created automatically by the server when you create a primary key or unique constraint.

Indexes are logically and physically independent of the table that they index. This means that they can be created or dropped at any time and have no effect on the base tables or other indexes.

When you drop a table, corresponding indexes are also dropped.

Types of Indexes

Two types of indexes can be created -

- Unique index - The Oracle server automatically creates this index when you define a column in a table to have a PRIMARY KEY or a UNIQUE key constraint. The name of the index is the name that is given to the constraint.
- Nonunique index - This is an index that a user can create. For example, you can create a FOREIGN KEY column index for a join in a query to improve retrieval speed.

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

Creating an Index

Create an index on one or more columns by issuing the CREATE INDEX statement.

```
CREATE INDEX emp_last_name_idx
ON employees(last_name);
```

Removing an Index

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

```
DROP INDEX emp_last_name_idx;
```

Synonyms

Synonyms are database objects that enable you to call a table by another name. You can create synonyms to give an alternate name to a table.

To refer to a table that is owned by another user, you need to prefix the table name with the name of the user who created it, followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

```
CREATE SYNONYM d_sum
FOR dept_sum_vu;


DROP SYNONYM d_sum;
```

Summary

Learned about database objects such as views, sequences, indexes, and synonyms.

- Create, use, and remove views;
- Automatically generate sequence numbers by using a sequence generator;
- Create indexes to improve query retrieval speed;
- Use synonyms to provide alternative names for objects.

--ooOoo--

Managing Objects with Dictionary Views

Objectives

Introduction to the data dictionary views. You will learn that the dictionary views can be used to retrieve metadata and create reports about your schema objects.

- Use the data dictionary views to research data on your objects;
- Query various data dictionary views.

Content

The Data Dictionary

User tables are tables created by the user and contain business data, such as EMPLOYEES.

There is another collection of tables and views in the Oracle database known as the *data dictionary*. This collection is created and maintained by the Oracle server and contains information about the database. The data dictionary is structured in tables and views, just like other database data. Not only is the data dictionary central to every Oracle database, but it is an important tool for all users, from end users to application designers and database administrators.

You use SQL statements to access the data dictionary. Because the data dictionary is read-only, you can issue only queries against its tables and views.

You can query the dictionary views that are based on the dictionary tables to find information such as -

- Definitions of all schema objects in the database (tables, views, indexes, synonyms, sequences, procedures, functions, packages, triggers, and so on);
- Default values for columns;
- Integrity constraint information;
- Names of Oracle users;
- Privileges and roles that each user has been granted;
- Other general database information.

Data Dictionary Structure

The data dictionary consists of sets of views. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes. For example, there is a view named USER_OBJECTS, another named ALL_OBJECTS, and a third named DBA_OBJECTS.

These three views contain similar information about objects in the database, except that the scope is different -

- USER_OBJECTS contains information about objects that you own or created.
- ALL_OBJECTS contains information about all objects to which you have access.
- DBA_OBJECTS contains information on all objects that are owned by all users. For views that are prefixed with ALL or DBA, there is usually an additional column in the view named OWNER to identify who owns the object.

How to Use the Dictionary Views

To familiarize yourself with the dictionary views, you can use the dictionary view named DICTIONARY. It contains the name and short description of each dictionary view to which you have access.

```
DESCRIBE DICTIONARY
```

USER_OBJECTS View

You can query the USER_OBJECTS view to see the names and types of all the objects in your schema. There are several columns in this view -

- OBJECT_NAME - Name of the object;
- OBJECT_ID - Dictionary object number of the object;
- OBJECT_TYPE - Type of object (such as TABLE, VIEW, INDEX, SEQUENCE);
- CREATED - Timestamp for the creation of the object;
- LAST_DDL_TIME - Timestamp for the last modification of the object resulting from a DDL command;
- STATUS - Status of the object (VALID, INVALID, or N/A);
- GENERATED - Was the name of this object system-generated? (Y|N).

You can also query the ALL_OBJECTS view to see a listing of all objects to which you have access.

USER_TABLES View

You can use the USER_TABLES view to obtain the names of all of your tables. The USER_TABLES view contains information about your tables. In addition to providing the table name, it contains detailed information on the storage.

The TABS view is a synonym of the USER_TABLES view. You can query it to see a listing of tables that you own -

```
SELECT table_name
FROM tabs;
```

You can also query the ALL_TABLES view to see a listing of all tables to which you have access.

Column Information

You can query the USER_TAB_COLUMNS view to find detailed information about the columns in your tables. While the USER_TABLES view provides information on your table names and storage, detailed column information is found in the USER_TAB_COLUMNS view.

This view contains information such as -

- Column names;
- Column data types;
- Length of data types;
- Precision and scale for NUMBER columns;
- Whether nulls are allowed (Is there a NOT NULL constraint on the column?);
- Default value.

Constraint Information

You can find out the names of your constraints, the type of constraint, the table name to which the constraint applies, the condition for check constraints, foreign key constraint information, deletion rule for foreign key constraints, the status, and many other types of information about your constraints.

- USER_CONSTRAINTS describes the constraint definitions on your tables.
- USER_CONS_COLUMNS describes columns that are owned by you and that are specified in constraints.

View Information

After your view is created, you can query the data dictionary view called USER_VIEWS to see the name of the view and the view definition.

The text of the SELECT statement that constitutes your view is stored in a LONG column. The LENGTH column is the number of characters in the SELECT statement.

Sequence Information

The USER_SEQUENCES view describes all sequences that are owned by you. When you create the sequence, you specify criteria that are stored in the USER_SEQUENCES view.

Synonym Information

The USER_SYNONYMS dictionary view describes private synonyms (synonyms that are owned by you).

You can query this view to find your synonyms. You can query ALL_SYNONYMS to find out the name of all of the synonyms that are available to you and the objects on which these synonyms apply.

Adding Comments to a Table

You can add a comment of up to 4,000 bytes about a column, table, view, or snapshot by using the COMMENT statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the COMMENTS column -

- ALL_COL_COMMENTS;
- USER_COL_COMMENTS;
- ALL_TAB_COMMENTS;
- USER_TAB_COMMENTS.

```
COMMENT ON TABLE employees
IS 'Employee Information';
```

You can drop a comment from the database by setting it to empty string (' ') -

```
COMMENT ON TABLE employees IS ' ';
```

Summary

Learned about some of the dictionary views that are available to you. You can use these dictionary views to find information about your tables, constraints, views, sequences, and synonyms.

Dictionary views -

- DICTIONARY;
- USER_OBJECTS;
- USER_TABLES;
- USER_TAB_COLUMNS;
- USER_CONSTRAINTS;
- USER_CONS_COLUMNS;
- USER_VIEWS;
- USER_SEQUENCES;
- USER_TAB_SYNONYMS.

--ooOoo--

Controlling User Access

Objectives

How to control database access to specific objects and add new users with different levels of access privileges.

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

Content

Controlling User Access

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

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

Database security can be classified into two categories: system security and data security -

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

Privileges

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

Schemas

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

System Privileges

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

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

Creating Users

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

```
CREATE USER USER1
IDENTIFIED BY USER1;
```

User System Privileges

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

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

```
GRANT create session, create table,
      create sequence, create view
TO scott;
```

What is a Role ?

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

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

Creating and Granting Privileges to a Role

Create a role -

```
CREATE ROLE manager;
```

Grant privileges to a role -

```
GRANT create table, create view
TO manager;
```

Grant a role to users -

```
GRANT manager TO BELL, KOCHHAR;
```

Changing Your password

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

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

```
ALTER USER HR
IDENTIFIED BY employ;
```

Object Privileges

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

Different object privileges are available for different types of schema objects. A user automatically has all object privileges for schema objects contained in the user's schema. A user can grant any object privilege on any schema object that the user owns to any other user or role.

If the grant includes WITH GRANT OPTION, then the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

Granting Object Privileges

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

```
GRANT select
ON employees
TO sue, rich;

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

Passing On privileges

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

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

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


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

Confirming Privileges Granted

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

If you receive the Oracle server error message "Table or view does not exist," then you have done either of the following -

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

You can access the data dictionary to view the privileges that you have.

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

Revoking Object Privileges

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

In the syntax - CASCADE is required to remove any referential integrity constraints made to the CONSTRAINTS object by means of the REFERENCES privilege.

```
REVOKE select, insert
ON departments
FROM scott;

[CASCADE CONSTRAINTS];
```

Summary

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

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

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

--ooOoo--

Managing Schema Objects

Objectives

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

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

Content

ALTER TABLE Statement

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

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

Adding a Column

You cannot specify where the column is to appear. The new column becomes the last column.

If a table already contains rows when a column is added, then the new column is initially null for all the rows.

```
ALTER TABLE dept80
ADD (job_id VARCHAR2(9));
```

Modifying a Column

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

```
ALTER TABLE dept80
MODIFY (last_name VARCHAR2(30));
```

Dropping a Column

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

- The column may or may not contain data.
- Using the ALTER TABLE statement, only one column can be dropped at a time.
- The table must have at least one column remaining in it after it is altered.
- After a column is dropped, it cannot be recovered.

- A column cannot be dropped if it is part of a constraint or part of an index key unless the cascade option is added.
- Dropping a column can take a while if the column has a large number of values. In this case, it may be better to set it to be unused and drop it when there are fewer users on the system to avoid extended locks.

```
ALTER TABLE dept80
DROP COLUMN job_id;
```

SET UNUSED Option

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

Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked as unused, you have no access to that column.

A SELECT * query will not retrieve data from unused columns. In addition, the names and types of columns marked unused will not be displayed during a DESCRIBE statement, and you can add to the table a new column with the same name as an unused column. SET UNUSED information is stored in the USER_UNUSED_COL_TABS dictionary view.

```
ALTER TABLE dept80
SET UNUSED (last_name);

ALTER TABLE dept80
DROP UNUSED COLUMNS;
```

Adding a Constraint Syntax

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

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

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

```
ALTER TABLE emp2
MODIFY employee_id Primary Key;


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

ON DELETE CASCADE

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

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

Deferring Constraints

You can defer checking constraints for validity until the end of the transaction. A constraint is deferred if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then commit causes the transaction to roll back. If a constraint is immediate (not deferred), then it is checked at the end of each statement. If it is violated, the statement is rolled back immediately. If a constraint causes an action (for example, DELETE CASCADE), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate.

Use the SET CONSTRAINTS statement to specify, for a particular transaction, whether a deferrable constraint is checked following each DML statement or when the transaction is committed. To create deferrable constraints, you must create a nonunique index for that constraint.

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

Deferring constraint on creation -

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

Changing a specific constraint attribute -

```
SET CONSTRAINTS dept2_id_pk IMMEDIATE
```

Changing all constraints for a session -

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

Dropping a Constraint

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

```
ALTER TABLE emp2
DROP CONSTRAINT emp_mgr_fk;

ALTER TABLE dept2
DROP PRIMARY KEY CASCADE;
```

Disabling Constraints

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

```
ALTER TABLE emp2
DISABLE CONSTRAINT emp_dt_fk;
```

Enabling Constraints

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

```
ALTER TABLE emp2
ENABLE CONSTRAINT emp_dt_fk;
```

Cascading Constraints

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

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

ALTER TABLE test1
DROP (pk, fk, col1) CASCADE CONSTRAINTS;
```

Overview of Indexes

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

The other type of index is a nonunique index, which a user can create. For example, you can create an index for a FOREIGN KEY column to be used in joins to improve retrieval speed. You can create an index on one or more columns by issuing the CREATE INDEX statement.

Indexes are created -

- Automatically -
  - PRIMARY KEY creation;
  - UNIQUE KEY creation;
- Manually -
  - CREATE INDEX statement;
  - CREATE TABLE statement.

CREATE INDEX with the CREATE TABLE Statement

In the example, the CREATE INDEX clause is used with the CREATE TABLE statement to create a primary key index explicitly. You can name your indexes at the time of primary key creation to be different from the name of the PRIMARY KEY constraint.

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

Function-Based Indexes

Function-based indexes defined with the UPPER(*column_name*) or LOWER(*column_name*) keywords allow non-case-sensitive searches. The Oracle server uses the index only when that particular function is used in a query.

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

Removing an Index

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

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

```
DROP INDEX upper_dept_name_idx;
```

DROP TABLE ... PURGE

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

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

You cannot roll back a DROP TABLE statement with the PURGE clause, and you cannot recover the table if you drop it with the PURGE clause.

```
DROP TABLE dept80 PURGE;
```

FLASHBACK TABLE Statement

SQL DDL command, FLASHBACK TABLE, to restore the state of a table to an earlier point in time in case it is inadvertently deleted or modified. The FLASHBACK TABLE command is a self-service repair tool to restore data in a table along with associated attributes such as indexes or views. This is done while the database is online by rolling back only the subsequent changes to the given table.

```
DROP TABLE emp2;

FLASHBACK TABLE emp2 TO BEFORE DROP;
```

External Tables

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

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

Summary

Alter tables to add or modify columns or constraints.

Create indexes and function-based indexes using the CREATE INDEX statement.

Drop unused columns. Use FLASHBACK mechanics to restore tables.

Use the external_table clause to create an external table, which is a readonly table whose metadata is stored in the database but whose data is stored outside the database.

Use external tables to query data without first loading it into the database.

Name your PRIMARY KEY column indexes as you create the table with the CREATE TABLE statement.

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

--ooOoo-

Manipulating Large Data Sets

Objectives

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

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

Content

Using Subqueries to Manipulate Data

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

Copying Rows from Another Table

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

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

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

Inserting Using a Subquery as a Target

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

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

Retrieving Data with a Subquery as Source

You can use a subquery in the FROM clause of a SELECT statement, which is very similar to how views are used. A subquery in the FROM clause of a SELECT statement is also called an *inline* view. A subquery in the FROM clause of a SELECT statement defines a data source for that particular SELECT statement, and only that SELECT statement.

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

Updating Two Columns with a Subquery

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

```
UPDATE empl3
SET job_id = (SELECT job_id
              FROM employees
              WHERE employee_id = 205),
    salary = (SELECT salary
              FROM employees
              WHERE employee_id = 168)
WHERE employee_id = 114;
```

Updating Rows Based on Another Table

You can use subqueries in UPDATE statements to update rows in a table.

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

Deleting Rows Based on Another Table

You can use subqueries to delete rows from a table based on values from another table.

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

Using the WITH CHECK OPTION Keyword on DML Statements

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

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

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

Explicit Defaults

The DEFAULT keyword can be used in INSERT and UPDATE statements to identify a default column value. If no default value exists, a null value is used. The DEFAULT option saves you from hard coding the default value in your programs or querying the dictionary to find it, as was done before this feature was introduced.

Hard coding the default is a problem if the default changes because the code consequently needs changing. Accessing the dictionary is not usually done in an application program, so this is a very important feature.

DEFAULT with INSERT -

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

DEFAULT with UPDATE -

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

Overview of Multitable INSERT Statements

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

The types of multitable INSERT statements are -

- Unconditional INSERT -

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

- Conditional ALL INSERT -

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

- Conditional FIRST INSERT -

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

If the condition WHEN SAL > 25 000 is true, values are inserted into special_sal, and the other WHEN clauses are skipped for the current row.

- Pivoting INSERT -

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

MERGE Statement

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

Performs an UPDATE if the row exists, and an INSERT if it is a new row.

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

Tracking Data Changes in Data

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

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

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

Summary

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

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

--ooOoo--

Generating Reports by Grouping Related Data

Objectives

Learn how to -

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

Content

Review of Group Functions

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

Types of group functions -

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

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

Guidelines for Using Group Functions

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

Review of the GROUP BY Clause

The example is evaluated by the Oracle server as follows -

- The SELECT clause specifies that the following columns be retrieved -
  - Department ID and job ID columns from the EMPLOYEES table;
  - The sum of all the salaries and the number of employees in each group that you have specified in the GROUP BY clause;

- The GROUP BY clause specifies how the rows should be grouped in the table. The total salary and the number of employees are calculated for each job ID within each department. The rows are grouped by department ID and then grouped by job within each department.

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

Review of the HAVING Clause

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

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

- It groups rows, specifies which groups are to be displayed.
- It applies the group functions to the groups and displays the groups that match the criteria in the HAVING clause.

GROUP BY with the ROLLUP and CUBE Operators

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

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

- ROLLUP grouping produces a result set containing the regular grouped rows and the subtotal values.
- CUBE grouping produces a result set containing the rows from ROLLUP and cross-tabulation rows.

ROLLUP Operator

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

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

- ROLLUP is an extension to the GROUP BY clause.
- Use the ROLLUP operation to produce cumulative aggregates, such as subtotals;
- Subtotals and totals are produced with ROLLUP. CUBE produces totals as well, but effectively rolls up in each possible direction, producing cross-tabular data.

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

In the example -

- Total salaries for every job ID within a department for those departments whose department ID is less than 60 are displayed by the GROUP BY clause;
- The ROLLUP operator displays -
  - Total salary for each department whose department ID is less than 60;
  - Total salary for all departments whose department ID is less than 60, irrespective of the job IDs.
- In this example, totaled by both DEPARTMENT_ID and JOB_ID, totalled only by DEPARTMENT_ID, and the grand total.
- The ROLLUP operator creates subtotals that roll up from the most detailed level to a grand total, following the grouping list specified in the GROUP BY clause. First, it calculates the standard aggregate values for the groups specified in the GROUP BY clause (in the example, the sum of salaries grouped on each job within a department). Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. (In the example, the sum of salaries for each department is calculated, followed by the sum of salaries for all departments.)

CUBE Operator

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

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

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

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

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

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

- The total salary for every job within a department (for those departments whose department ID is less than 60).
- The total salary for those departments whose department ID is less than 60;
- The total salary for every job irrespective of the department;
- The total salary for those departments whose department ID is less than 60, irrespective of the job titles.

GROUPING Function

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

- Used to find the groups forming the subtotal in a row;
- Is used to differentiate stored NULL values from NULL values created by ROLLUP or CUBE;
- Returns 0 or 1. (0 - has been taken into account, 1 - has NOT been taken into account).

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

GROUPING SETS

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

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

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

This statement calculates aggregates over three groupings -

- (department_id, job_id, manager_id),
- (department_id, manager_id),
- and (job_id, manager_id)

Without this feature, multiple queries combined together with UNION ALL are required to obtain the output of the preceding SELECT statement.

Composite Columns

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

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

Here, (b, c) forms a composite column and is treated as a unit. In general, composite columns are useful in ROLLUP, CUBE, and GROUPING SETS.

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

Concatenated Groupings

Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified by listing multiple grouping sets, CUBEs, and ROLLUPs, and separating them with commas.

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

Summary

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

--ooOoo--

Managing Data in Different Time Zones

Objectives

This lesson addresses some of the datetime functions available in the Oracle database. After completing this lesson, you should be able to use the following datetime functions -

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

Content

CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP

CURRENT_DATE

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

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

CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP function returns the current date and time in the session time zone, as a value of the TIMESTAMP WITH TIME ZONE data type.

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

LOCALTIMESTAMP

The LOCALTIMESTAMP function returns the current date and time in the session time zone. LOCALTIMESTAMP returns a TIMESTAMP value.

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

EXTRACT

The EXTRACT expression extracts and returns the value of a specified datetime field from a datetime or interval value expression.

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


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

The syntax of the EXTRACT function is -

```
SELECT EXTRACT ([YEAR] [MONTH][DAY] [HOUR] [MINUTE][SECOND]
            [TIMEZONE_HOUR] [TIMEZONE_MINUTE]
            [TIMEZONE_REGION] [TIMEZONE_ABBR]
FROM [datetime_value_expression] [interval_value_expression]);
```

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

Summary

How to use the following functions -

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

--ooOoo--

Retrieving data using sub-queries

Objectives

In this lesson, you learn how to write multiple-column subqueries and subqueries in the FROM clause of a SELECT statement. You also learn how to solve problems by using scalar, correlated subqueries and the WITH clause.

- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Update and delete rows using correlated subqueries
- Use the EXISTS and NOT EXISTS operators
- Use the WITH clause

Content

Multiple-Column Subqueries

So far, you have written single-row subqueries and multiple-row subqueries where only one column is returned by the inner SELECT statement and this is used to evaluate the expression in the parent SELECT statement. If you want to compare two or more columns, you must write a compound WHERE clause using logical operators. Using multiple-column subqueries, you can combine duplicate WHERE conditions into a single WHERE clause.

Column Comparisons - Pairwise Versus Nonpairwise Comparisons

Multiple-column comparisons involving subqueries can be nonpairwise comparisons or pairwise comparisons. If you consider the example “Display the details of the employees who work in the same department, and have the same manager, as ‘Daniel’?”, you get the correct result with the following statement -

```
SELECT first_name, last_name, manager_id, department_id
FROM employees
WHERE manager_id IN (SELECT manager_id
                     FROM employees
                     WHERE first_name = ‘Daniel’)
AND department_id IN (SELECT department_id
                      FROM employees
                      WHERE first_name = ‘Daniel’);
```

There is only one “Daniel” in the EMPLOYEES table (Daniel Faviet, who is managed by employee 108 and works in department 100). However, if the subqueries return more than one row, the result might not be correct. For example, if you run the same query but substitute “John” for “Daniel,” you get an incorrect result. This is because the combination of department_id and manger_id is important. To get the correct result for this query, you need a pairwise comparison.

Pairwise Comparison Subquery (Both entries match only the specific row of the subquery)

The example compares the combination of values in the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table with the values in the MANAGER_ID column and the DEPARTMENT_ID column for the employees with the FIRST_NAME of “John.” First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for the employees with the FIRST_NAME of “John” is executed.

These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table. If the combination matches, the row is displayed. In the output, the records of the employees with the FIRST_NAME of “John” will not be displayed.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE (manager_id, department_id) IN
            (SELECT manager_id, department_id
             FROM employees
             WHERE first_name = 'John')
AND first_name <> 'John';
```

Nonpairwise Comparison Subquery (Either entry matches any row in the subquery, ie: not necessarily the same row))

The example shows a nonpairwise comparison of the columns. First, the subquery to retrieve the MANAGER_ID values for the employees with the FIRST_NAME of “John” is executed.

Similarly, the second subquery to retrieve the DEPARTMENT_ID values for the employees with the FIRST_NAME of “John” is executed. The retrieved values of the MANAGER_ID and DEPARTMENT_ID columns are compared with the MANAGER_ID and DEPARTMENT_ID columns for each row in the EMPLOYEES table. If the MANAGER_ID column of the row in the EMPLOYEES table matches with any of the values of MANAGER_ID retrieved by the inner subquery and if the DEPARTMENT_ID column of the row in the EMPLOYEES table matches with any of the values of DEPARTMENT_ID retrieved by the second subquery, the record is displayed.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE manager_id IN
      (SELECT manager_id
       FROM employees
       WHERE first_name = 'John')
AND department_id IN
      (SELECT department_id
       FROM employees
       WHERE first_name = 'John')
AND first_name <> 'John';
```

This query retrieves five rows more than the pairwise comparison (those with the combination of manager_id=100 and department_id=50, although no employee named “John” has such a combination).

Scalar Subquery Expressions

A subquery that returns exactly one column value from one row is also referred to as a scalar subquery. Multiple-column subqueries that are written to compare two or more columns, using a compound WHERE clause and logical operators, do not qualify as scalar subqueries.

The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is NULL. If the subquery returns more than one row, the Oracle server returns an error. The Oracle server has always supported the usage of a scalar subquery in a SELECT statement. You can use scalar subqueries in -

- The condition and expression part of DECODE and CASE;
- All clauses of SELECT except GROUP BY;
- The SET clause and WHERE clause of an UPDATE statement.

Scalar subqueries in CASE expressions -

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

Scalar subqueries in the ORDER BY clause -

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

Correlated Subqueries

The Oracle server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement.

Nested Subqueries Versus Correlated Subqueries

With a normal nested subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. That is, the inner query is driven by the outer query.

Nested Subquery Execution

- The inner query executes first and finds a value.
- The outer query executes once, using the value from the inner query.

Correlated Subquery Execution

- Get a candidate row (fetched by the outer query).
- Execute the inner query using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. That is, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

The Oracle server performs a correlated subquery when the subquery references a column from a table in the parent query.

You can use the ANY and ALL operators in a correlated subquery.

```
SELECT last_name, salary, department_id
FROM employees outer
WHERE salary >
      (SELECT AVG(salary)
       FROM employees
       WHERE department_id = outer.department_id);
```

The example determines which employees earn more than the average salary of their department. In this case, the correlated subquery specifically computes the average salary for each department.

Because both the outer query and inner query use the EMPLOYEES table in the FROM clause, an alias is given to EMPLOYEES in the outer SELECT statement for clarity. The alias makes the entire SELECT statement more readable. Without the alias, the query would not work properly because the inner statement would not be able to distinguish the inner table column from the outer table column.

Display details of those employees who have changed jobs at least twice -

```
SELECT e.employee_id, last_name,e.job_id
FROM employees e
WHERE 2 <= (SELECT COUNT(*)
            FROM job_history
            WHERE employee_id = e.employee_id);
```

EXISTS Operator

With nesting SELECT statements, all logical operators are valid. In addition, you can use the EXISTS operator. This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query. If the subquery returns at least one row, the operator returns TRUE. If the value does not exist, it returns FALSE. Accordingly, NOT EXISTS tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.

```
SELECT employee_id, last_name, job_id, department_id
FROM employees outer
WHERE EXISTS ( SELECT 'X'
      FROM employees
      WHERE manager_id =
            outer.employee_id);
```

The EXISTS operator ensures that the search in the inner query does not continue when at least one match is found for the manager and employee number by the condition -

WHERE manager_id = outer.employee_id.
Note that the inner SELECT query does not need to return a specific value, so a constant can be selected.

Correlated UPDATE

In the case of the UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

```
UPDATE empl6 e
SET department_name =
      (SELECT department_name
       FROM departments d
       WHERE e.department_id = d.department_id);
```

Correlated DELETE

In the case of a DELETE statement, you can use a correlated subquery to delete only those rows that also exist in another table.

```
DELETE FROM empl6 E
WHERE employee_id =
      (SELECT employee_id
       FROM emp_history
       WHERE employee_id = E.employee_id);
```

WITH Clause

Using the WITH clause, you can define a query block before using it in a query. The WITH clause (formally known as subquery_factoring_clause) enables you to reuse the same query block in a SELECT statement when it occurs more than once within a complex query. This is particularly useful when a query has many references to the same query block and there are joins and aggregations.

- It is used only with SELECT statements.
- A query name is visible to all WITH element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).
- When the query name is the same as an existing table name, the parser searches from the inside out, and the query block name takes precedence over the table name.
- The WITH clause can hold more than one query. Each query is then separated by a comma.

```
WITH
dept_costs AS (
      SELECT d.department_name, SUM(e.salary) AS dept_total
      FROM employees e JOIN departments d
      ON e.department_id = d.department_id
      GROUP BY d.department_name),
avg_cost AS (
      SELECT SUM(dept_total)/COUNT(*) AS dept_avg
      FROM dept_costs)
SELECT *
FROM dept_costs
WHERE dept_total >
      (SELECT dept_avg
       FROM avg_cost)
       ORDER BY department_name;
```

Summary

You can use multiple-column subqueries to combine multiple WHERE conditions in a single WHERE clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

You can use a subquery to define a table to be operated on by a containing query. Scalar subqueries can be used in -

- Condition and expression part of DECODE and CASE;
- All clauses of SELECT except GROUP BY;
- A SET clause and WHERE clause of the UPDATE statement.

The Oracle server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement. Using the WITH clause, you can reuse the same query when it is costly to reevaluate the query block and it occurs more than once within a complex query.

- Correlated subqueries are useful whenever a subquery must return a different result for each candidate row;
- The EXISTS operator is a Boolean operator that tests the presence of a value;
- Correlated subqueries can be used with SELECT, UPDATE, and DELETE statements;
- You can use the WITH clause to use the same query block in a SELECT statement when it occurs more than once.

--ooOoo--

Hierarchical Retrieval

Objectives

In this lesson, you learn how to use hierarchical queries to create tree-structured reports -

- Interpret the concept of a hierarchical query
- Create a tree-structured report
- Format hierarchical data
- Exclude branches from the tree structure

Content

Section omitted.

Summary

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table. The LEVEL pseudocolumn counts how far down a hierarchical tree you have travelled. You can specify the direction of the query using the CONNECT BY PRIOR clause. You can specify the starting point using the START WITH clause. You can use the WHERE and CONNECT BY clauses to prune the tree branches.

--ooOoo--

Regular Expression Support

Objectives

After completing this lesson, you should be able to use regular expression support in SQL to search, match, and replace strings in terms of regular expressions.

Content

Meta Characters

Meta characters are special characters that have a special meaning, such as a wildcard character, a repeating character, a nonmatching character, or a range of characters. You can use several predefined meta character symbols in the pattern matching.

Meta Character Syntax	Operator Name	Description
.	Any Character – Dot	Match any character.
+	One or More – Plus Quantifier	Match one or more occurrences of the preceding subexpression.
?	Zero or One – Question Mark Quantifier	Match zero or one occurrence of the preceding subexpression.
*	Zero or More – Star Quantifier	Match zero or more occurrences of the preceding subexpression.
{m} {m,} {m,n}	Interval – Exact Count	Match - • exactly *m* occurrences • at least *m* occurrences • at least *m*, but not more than *n* occurrences of the preceding subexpression
[…]	Matching Character List	Match any character in list ...
[^…]	Non-Matching Character List	Match any character not in list ...
|	Or	'a|b' matches character 'a' or 'b'.
(…)	Subexpression or Grouping	Treat expression ... as a unit.
\n	Back reference	Match the n^{th} preceding subexpression, where *n* is an integer from 1 to 9
\	Escape character	Treat the subsequent meta character in the expression as a literal.
^	Beginning of line anchor	Match the subsequent expression when it occurs at the beginning of a line.
$	End of line anchor	Match the preceding expression only when it occurs at the end of a line.

Using Meta Characters

```
Problem: Find 'abc' within a string -

Solution:         'abc'
Matches:           abc
Does not match: 'def'


Problem: To find 'a' followed by any character, followed by 'c' -
Meta Character: any character is defined by '.'

Solution:         'a.c'
Matches:           abc
Matches:           adc
Matches:           alc
Matches:           a&c
Does not match:  abb


Problem: To find one or more occurrences of 'a' -
Meta Character: Use'+' sign to match one or more of the previous characters

Solution:         'a+'
Matches:           a
Matches:           aa
Does not match:  bbb
```

You can search for nonmatching character lists too. A nonmatching character list allows you to define a set of characters for which a match is invalid. For example, to find anything but the characters “a,” “b,” or “c,” you can define the “^” to indicate a non-match.

```
Expression:       [^abc]
Matches:          abcdef
Matches:          ghi
Does not match:   abc
```

To match any letter not between “a” and “i,” you can use -

```
Expression:       [^a-i]
Matches:          hijk
Matches:          lmn
Does not match:   abcdefghi
```

Regular Expression Functions

Oracle Database 10*g* provides a set of SQL functions that you can use to search and manipulate strings using regular expressions. You can use these functions on any data type that holds character data such as CHAR, NCHAR, CLOB, NCLOB, NVARCHAR2, and VARCHAR2. A regular expression must be enclosed or wrapped between single quotation marks. Doing so ensures that the entire expression is interpreted by the SQL function and can improve the readability of your code.

REGEXP_LIKE: This function searches a character column for a pattern. Use this function in the WHERE clause of a query to return rows matching the regular expression you specify.

REGEXP_REPLACE: This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern you specify.

REGEXP_INSTR: This function searches a string for a given occurrence of a regular expression pattern. You specify which occurrence you want to find and the start position to search from.
This function returns an integer indicating the position in the string where the match is found.

REGEXP_SUBSTR: This function returns the actual substring matching the regular expression pattern you specify.

Performing Basic Searches

In this query, against the EMPLOYEES table, all employees with first names containing either Steven or Stephen are displayed.

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

Checking the Presence of a Pattern

In this example, the REGEXP_INSTR function is used to search the street address to find the location of the first nonalphabetic character, regardless of whether it is in uppercase or lowercase. The search is performed only on those addresses that do not start with a number. Note that [:<class>:] implies a character class and matches any character from within that class, and [:alpha:] matches with any alphabetic character.

```
SELECT street_address,
REGEXP_INSTR(street_address,'[^[:alpha:]]')
FROM locations
WHERE
REGEXP_INSTR(street_address,'[^[:alpha:]]')> 1;
```

Extracting Substrings

In this example, the road names are extracted from the LOCATIONS table. To do this, the contents in the STREET_ADDRESS column that are before the first space are returned using the REGEXP_SUBSTR function.

```
SELECT REGEXP_SUBSTR(street_address , ' [^ ]+ ')
"Road" FROM locations;
```

Replacing Patterns

This example examines COUNTRY_NAME. The Oracle database reformats this pattern with a space after each non-null character in the string.

```
SELECT REGEXP_REPLACE( country_name, '(.)',
      '\1 ') "REGEXP_REPLACE"
FROM countries;
```

Summary

In this lesson, you should have learned how to use regular expression support in SQL and PL/SQL to search, match, and replace strings in terms of regular expressions.

--ooOoo--