A brief Introduction

to

Oracle SQL/PL-SQL

Prepared for Database Course

Dept. of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Author

Sukarna Barua

Associate Professor

Dept. of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka-1000, Bangladesh.

Last Update: 12 November 2023

Table of Contents

           Author: Sukarna Barua

Chapter 1: Introduction

1. **Introduction**

What is SQL

Structured Query Language (SQL) is the language used to interaction with a database management system. The language defines the statements for retrieving data from the database, updating data in the database, inserting data into the database, and more similar things.

SQL is the way by which all programs and users access data in an Oracle database. SQL provides statements for a variety of tasks including:

- Querying data

- Inserting, updating, and deleting rows in a table

- Creating, replacing, altering, and dropping objects

- Controlling access to the database, and its objects

- Guaranteeing database consistency and integrity

SQL Statements

All SQL statements supported by Oracle database can be grouped in several categories as illustrated in following table:

SQL statement	Function of the statement
SELECT	Retrieves data from database
INSERT	Inserts data, updates data, and deletes data in database
UPDATE	
DELETE	
CREATE	Creates new tables in database, alters existing tables, deletes tables from database
ALTER	
DROP	
COMMIT	Saves data permanently in database, erases changes done form database
ROLLBACK	

Retrieve data — SELECT

Manipulate Data — INSERT, UPDATE, DELETE

Manipulate Table — CREATE, ALTER, DROP

Manipulate database — COMMIT, ROLLBACK

SQL Working Tools

There are two primary tools that can be used for SQL. They are:

- SQL * PLUS – This is a command line tool and is the most popular tool

- SQL Developer – This is a graphical tool used by Oracle users to interact with Oracle database.

- Navicat – This is a third party tool that is used to connect to several databases such as Oracle, MySQL, etc. This is one of the most popular GUI tools to interact with database.

The HR schema

The Human Resource (HR) schema is a part of Oracle sample database that can be installed with Oracle. This course uses the tables in HR schema for all practice and homework sessions. The schema contains the following tables:

- REGIONS – contain rows that represent a region such as America, Asia, and so on.

- COUNTRIES – contain rows for countries each of which is associated with a region

- LOCATIONS – contains the specific address of a specific office, warehouse, or a production site of a company in a particular country

- DEPARTMETNS – shows detail about a department in which an employee works. Each department may have a relationship representing the department manager in the EMPLOYEES table.

- EMPLOYEES – contain detail about each employee working for a department. Some employees may not be assigned to any department.

- JOBS – contain the job types that can be held by each employee.

- JOB_HISTORY – contain the job history of the employees. If an employee changes department within a job, or changes a job within a department, a new row is inserted in this table with the earlier job information of the employee.

The following table shows the column names for all tables in HR schema.

Table Name	Column Names
REGIONS	REGION_ID, REGION_NAME
COUNTRIES	COUNTRY_ID, COUNTRY_NAME, REGION_ID

LOCATIONS	LOCATION_ID, STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID
DEPARTMENTS	DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID
EMPLOYEES	EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, DEPARTMENT_ID
JOB_HISTORY	EMPLOYEE_ID, START_DATE, END_DATE, JOB_ID, DEPARTMENT_ID
JOBS	JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY

CHAPTER 2: Retrieve Data from Database Tables

1. **Selecting Data Using Basic SELECT Statement**

Basic SELECT statement

The SELECT statement is used to retrieve data from database tables. The very basic general form of the SELECT statement is given below:

```
SELECT Column1, Column2, …
FROM table_name ;
```

For example, the following SELECT statement retrieves some data from EMPLOYEES table.

```
SELECT EMPLOYEE_ID, LAST_NAME, SALARY
FROM EMPLOYEES ;
```

The above SELECT statement –

- Will retrieve all rows from the EMPLOYEES table

- Will retrieve values of three columns only, EMPLOYEE_ID, LAST_NAME, SALARY

If you want select values of all columns you can either specify all column names or specify a '*' as follows:

```
SELECT * FROM EMPLOYEES ;
```

Expressions instead of column names

You can also use expressions instead of direct column names in the SELECT clause as shown below. The query will multiply values of SALARY column by 12, and then will display the results.

```
SELECT EMPLOYEE_ID, LAST_NAME, SALARY*12
FROM EMPLOYEES ;
```

You can use arithmetic expressions involving addition, subtraction, multiplication, and division as shown in following SELECT statements.

```
SELECT EMPLOYEE_ID, LAST_NAME, SALARY*12
FROM EMPLOYEES ;


SELECT EMPLOYEE_ID, LAST_NAME, SALARY+12
FROM EMPLOYEES ;


SELECT EMPLOYEE_ID, LAST_NAME, (SALARY-1000)*5
FROM EMPLOYEES ;


SELECT EMPLOYEE_ID, LAST_NAME, (SALARY + SALARY*0.15) / 1000
FROM EMPLOYEES ;
```

Use of column alias

You will notice that, when you execute the first SELECT query above with expressions instead of column names, the displayed column header in the result is SALARY*12. These headers sometime become misleading and unreadable. Using alias, i.e., naming the expression, will solve the problem. You can give meaning names to expressions in this way. The following example illustrates this and gives the expression a name ANNSAL. The query results will be displayed with ANNSAL as the column header.

```
SELECT EMPLOYEE_ID, LAST_NAME, SALARY*12 ANNSAL
FROM EMPLOYEES ;
```

The column alias should be double quoted if it contains spaces as shown below:

```
SELECT EMPLOYEE_ID, LAST_NAME, SALARY*12 "ANNUAL SALARY"
FROM EMPLOYEES ;
```

You should remember the following when writing SQL statements.

- All SQL statements have a terminator, i.e., semicolon (;) at the end

- SQL Keywords are not case-sensitive, e.g., SELECT, Select, and select are all same.

- SQL statements can be entered in one or more lines to increase readability

NULL values in columns

In Oracle, a NULL value -

- <mark>Means a value in the column which is unassigned.</mark>

- <mark>NULL value means empty value, i.e., no value was given</mark>

- <mark>NULL value is not same as zero value or some other specific value, it is simply unknown</mark>

- Arithmetic expressions involving a NULL value outputs NULL

Execute the following SELECT statement and search through the values of COMMISSION_PCT column in the output. You will notice that there are many NULL, i.e., empty values there.

```
SELECT LAST_NAME, COMMISSION_PCT
FROM EMPLOYEES ;
```

The following example illustrate that when a column value is NULL, then arithmetic expression also outputs NULL. You will notice that SALCOMM column in the output is null in those rows where either SALARY value is NULL or COMMISSION_PCT value is NULL.

```
SELECT LAST_NAME, (SALARY+SALARY*COMMISSION_PCT) SALCOMM
FROM EMPLOYEES ;
```

Use of text concatenation operator (||)

The concatenation operation, || is used to join multiple text values. For example, the following query outputs employee's first name and last name joined together and separated by a space. The output column header is also given a new name FULLNAME. Although, the parenthesis is not necessary, it improves readable of the statement.

```
SELECT (FIRST_NAME || ' ' || LAST_NAME) FULLNAME, SALARY
FROM EMPLOYEES ;
```

You can also add additional texts with the column values as the following statement does.

```
SELECT ('NAME is: ' || FIRST_NAME || ' ' || LAST_NAME) FULLNAME, SALARY
FROM EMPLOYEES ;
```

Note the use of single quotes in the above statement. The single quote is used to mean text values. This is necessary and without the single quote, Oracle will think the text as a column name, which is definitely wrong in this case. So, you must use single quote around text values (not around column names). However, no quote is used for numeric values, e.g., 350, and 7777.

Use of DISTINCT keyword for removing duplicate column values

Consider the output of the following SELECT statements.

```
SELECT JOB_ID
FROM EMPLOYEES ;
```

You will notice that, the output contains similar, i.e., duplicate values since several rows (employees) have the same JOB_ID in the EMPLOYEES table. You can remove the duplicate outputs by using the DISTINCT keyword as shown below.

```
SELECT DISTINCT JOB_ID
FROM EMPLOYEES ;
```

In the above SELECT, output will contain unique values of JOB_ID therefore removing duplicate outputs. The DISTINCT Keyword works on all columns specified in the SELECT and outputs unique rows (removes duplicate rows).

```
SELECT DISTINCT DEPARTMENT_ID, JOB_ID
FROM EMPLOYEES ;
```

Use of DESCRIBE statement

You can use the DESCRIBE statement to view the structure, i.e., column names and data types of different columns of a table. The following example will output all column names, their data types, and all constraints of the EMPLOYEES table.

```
DESCRIBE EMPLOYEES ;
```

The basic data types used in Oracle are given in following table.

Data Type	Description
NUMBER	Used for storing numeric values
VARCHAR2(size)	Used for storing variable length text data, where parameter *size* specifies the maximum characters in the text
CHAR(size)	Used for storing fixed-length text data where *size* is the number of characters
DATE	Used for storing date values

Practice 2.1

    a.   Write an SQL query to retrieve all country names.

    b.   Write an SQL query to retrieve all job titles.

    c.   Write an SQL query to retrieve all MANAGER_IDs.

    d.   Write an SQL query to retrieve all city names. Remove duplicate outputs.

    e.   Write an SQL query to retrieve LOCATION_ID, ADDRESS from LOCATIONS table. The ADDRESS should print each location in the following format: STREET_ADDRESS, CITY, STATE_PROVINCE, POSTAL_CODE.

2.   SELECT Statement with WHERE clause

Use of WHERE clause in SELECT Statement

The WHERE clause is used with one or more conditions to limit rows in the output. The general syntax of the SELECT statement with WHERE clause is given below.

```
SELECT …
FROM table_name
WHERE condition ;
```

For example, the following SELECT retrieves last names and salaries of those employees only whose DEPARTMENT_ID column value is 80.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 80 ;
```

The following example shows conditions involving text and date values.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME = 'Whalen' ;


SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE HIRE_DATE =  '01-JAN-1995' ;


SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE COMMISSION_PCT IS NULL ;
```

Note that use of single quotes around text data and date data ('01-JAN-1995'). The date text given in the query is in default format. Until, you learn TO_DATE conversion function in later chapters, you will need to enter data values in this way in default format. Otherwise, Oracle may generate error messages.

Comparison operators in Oracle

The condition in the above WHERE clause uses the equal ('=') operator to compare DEPARTMENT_ID values. This is a comparison operator available in Oracle. The most commonly used comparison operators are:

Operator	Description
=	Equal to

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
BETWEEN value1 AND value	Between value1 and value2 (inclusive)
IN (value1, value2, …)	Equal to value1 or value 2 or …
LIKE	Pattern matching for text
IS NULL	Is a NULL value

!= IS ALSO VALID BUT DOESN'T FOLLOW ISO STANDARS

Range condition using BETWEEN operator

You can check whether a value is in the given range by using the BETWEEN operator. The following statements illustrate this.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE SALARY BETWEEN 5000 AND 10000 ;


SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE HIRE_DATE BETWEEN '01-JAN-1990' AND '31-DEC-1995' ;
```

Use of IN operator to match the column value against a set of values

The IN operator can be used to check whether a column value is equal to a set of values. The following examples illustrate this.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (50, 60, 70, 80, 90, 100) ;


SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME IN ('Ernst', 'Austin', 'Pataballa', 'Lorentz') ;
```

Joining multiple conditions in WHERE clause

You can join two or more conditions using AND, OR, and NOT operators. You may need to use parenthesis to enforce operation execution order.

The following example shows the use of AND, OR and NOT operators.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 80 AND SALARY > 5000 ;


SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE JOB_ID = 'SALES_REP' OR SALARY >= 10000 ;


SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE COMMISSION_PCT IS NOT NULL ;
```

Three or more conditions can be joined. In such cases, parenthesis should be used to clarify the execution order and combination of the conditions.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID <> 80 AND
(SALARY > 5000 OR COMMISSION_PCT IS NOT NULL)  ;


SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE (DEPARTMENT_ID = 80 AND SALARY > 5000) OR
COMMISSION_PCT IS NOT NULL)  ;
```

Note the difference in outputs of the above two statements. The use of parenthesis change the order of execution of the operators, therefore changes meaning and output of the SELECT statements.

Pattern matching in texts using LIKE operator

The LIKE operator is used for pattern matching in texts. Suppose, you want to retrieve those employee records whose last names contain the character, 's'. This type of condition is expressed using LIKE operators and two special symbols ('%' and '_'). For example, the following statement will retrieve all retrieve records of those employees whose last name column contain at least one 's'. → small s, not capital

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME LIKE '%s%' ;
```

Note the use of special symbol ('%') above. The '%' means zero or more characters. A '%' before 's' means zero or more characters can precede the 's', a '%' after 's' means zero or more characters can follow 's'. To understand the position of '%' and resultant effect, observe the outputs of the following queries.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME LIKE 'S%' ;

SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME LIKE '%s' ;

SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME LIKE 's' ;
```

If you execute the above three statements and observe the output, you will note that-

- The first statement retrieves those rows where LAST_NAME starts with a 'S'. Any number of any characters can follow after the 'S'. Note that text match is case-sensitive, hence 'S%' and 's%' are not same.

- The second statement retrieves those rows where LAST_NAME ends with 's'. Any number of any characters can precede the 's'. Only the last character must be 's'.

- The third statement retrieves those rows where LAST_NAME is exactly 's' (like the '=' operator)

Like the '%' special symbol, the '_' special symbol is used to match exactly one character. For example, execute the following statements and observe the outputs.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME LIKE 'a_' ;

SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME LIKE '_ _ b' ;

SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE LAST_NAME LIKE '_%' ;
```

If you execute the above three statements and observe the output, you will note that-

- The first statement retrieves those rows where LAST_NAME contains exactly two characters where the first character must be 'a', e.g., 'ab', 'ak', and 'ac'.

- The second statement retrieves those rows where LAST_NAME contains exactly three characters in which the last one must be 'b'

- The third statement retrieves those rows where LAST_NAME contains at least one character, i.e., last name cannot be empty text

NULL values in comparison

In a comparison statement, if anything is NULL, than the comparison is regarded as FALSE, and corresponding row is not retrieved. Consider the following statement.

```
SELECT LAST_NAME, SALARY, COMMISSION_PCT
FROM EMPLOYEES
WHERE COMMISSION_PCT <  0.20 ;
```

The above statement retrieves records of those employees whose COMMISSION_PCT value is less than 0.20. The query should retrieve those rows that have NULL values in COMMISSION_PCT column,

because NULL should mean a 0 value in most cases. However, Oracle would not retrieve those rows. This is because; comparison with NULL is regarded as FALSE (not matched). If you want to retrieve records with NULL values also, then you have to use IS NULL comparison operator as shown below.

```
SELECT LAST_NAME, SALARY, COMMISSION_PCT
FROM EMPLOYEES
WHERE COMMISSION_PCT < 0.20 OR COMMISSION_PCT IS NULL ;
```

A common mistake is using equality (=) operator to retrieve records with NULL values as shown below. But, the query will retrieve no rows as the equality comparison fails due to NULL value.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE COMMISSION_PCT = NULL ;
```

Practice 2.2

    a. Select names of all employees who have joined before January 01, 1998.

    b. Select all locations in the following countries: Canada, Germany, United Kingdom.

    c. Select first names of all employees who do not get any commission.

    d. Select first names of employees whose last name starts with an 'a'.

    e. Select first names of employees whose last name starts with an 's' and ends with an 'n'.

    f. Select all department names whose MANAGER_ID is 100.

    g. Select all names of employees whose job type is 'AD_PRES' and whose salary is at least 23000.

    h. Select names of all employees whose last name do not contain the character 's'.

    i. Select names and COMMISSION_PCT of all employees whose commission is at most 0.30.

    j. Select names of all employees who have joined after January 01, 1998.

    k. Select names of all employees who have joined in the year 1998.

3.   **Sorting Rows in the Output**

ORDER BY Clause

You can use the ORDER BY clause with the SELECT statement to sort the rows in the results. The general syntax is given below:

```
SELECT …
FROM table_name
WHERE …
ORDER BY Column1 [ASC | DESC], Column2 [ASC | DESC], …
```

The ORDER BY clause is used after the WHERE clause to specify sorting order. The following statement retrieves records of employees and sorts the output in descending order of SALARY values.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
ORDER BY SALARY DESC ;


SELECT LAST_NAME, SALARY, HIRE_DATE
FROM EMPLOYEES
ORDER BY HIRE_DATE ASC ;
```

Outputs can be sorted based on multiple columns. So, if the first column is equal for multiple rows, then the rows are sorted according to the second column. If the second column is also same, then rows are sorted based on third column, and so on. The following statement illustrates such queries.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
ORDER BY SALARY DESC, LAST_NAME ASC ;
```

You can use column alias in ordering results. The following examples illustrate this.

```
SELECT JOB_TITLE, (MAX_SALARY - MAX_SALARY) DIFF_SALARY
FROM JOBS
ORDER BY DIFF_SALARY DESC ;
```

→ MIN_SALARY

Practice 2.3

    a. Select names, salary, and commissions of all employees of job type 'AD_PRES'. Sort the result in ascending order of commission and then descending order of salary.

    b. Retrieve all country names in lexicographical ascending order.

4. More Practice Problems

In class.

     Author: Sukarna Barua

Chapter 3: Use of Oracle Single Row Functions

1.   **Character Functions**

Case conversion functions

The case conversion functions are used to convert case of text characters. The most commonly used Oracle functions for this purpose are given below:

Function	Description
LOWER (text)	Converts the text to all lowercase. Here, text can be column name or an expression.
UPPER (text)	Converts the text to all uppercase. Here, text can be column name or an expression.
INITCAP (text)	Converts the first character of the text to uppercase. Here, text can be column name or an expression.

The following table shows the outputs of applying case conversion functions on the text 'hello WORLD'.

Function	Output
LOWER('hello WORLD')	hello world
UPPER('hello WORLD')	HELLO WORLD
INITCAP('hello WORLD')	Hello world

The following statement shows the use case-conversion functions in SELECT statements.

```
SELECT (INITCAP(FIRST_NAME) ||  LOWER(LAST_NAME)) NAME, UPPER(JOB_ID) JOB
FROM EMPLOYEES ;
```

Note the use of three functions in the above statement. The column aliases are good ways to redefine column headers.

Case conversion functions are frequently used in WHERE clause to match specific texts. For example, if you want to retrieve records of those departments which contain the text SALE in its name, then you may write the following statement.

```
SELECT *
FROM DEPARTMENTS
WHERE UPPER(DEPARTMENT_NAME) LIKE '%SALE%' ;


SELECT *
FROM DEPARTMENTS
WHERE LOWER(DEPARTMENT_NAME) LIKE '%sale%' ;
```

Any of the above two statements will retrieve the required records successfully. However, if no functions were used, then the query may not retrieve all records due to case differences. For examples, if there were two departments named as 'Sales Office' and 'Whole sale', then comparing without functions may not be able to retrieve both records simultaneously.

Character manipulation functions

The most commonly used functions in this category are given below:

Function	Description
CONCAT (Column1, Column2)	Concatenates, i.e., joins the texts of two columns.
SUBSTR(Column, m [, n])	Extracts n characters from the Column, starting from m-th position. If n is omitted, the all characters from the starting position are extracted.
LENGTH (Column)	Outputs the length, i.e., number of characters of text.
INSTR(Column, 'text')	Returns the numeric position of the 'text' in Column if found. Otherwise, returns 0.
LPAD(Column, n, 'text')	Returns a left-padded text of n characters, padded with 'text'.
RPAD(Column, n, 'text')	Returns a right-padded text of n characters, padded with 'text'
TRIM(Column)	Trims, i.e., removes whitespace characters from left and right.
REPLACE(Column,'text1','text2')	Searches for 'text1' in Column, and if found, replaces with 'text2'.

The following table shows the output of applying these functions.

Function	Output
CONCAT('Hello', ' world')	Hello world
SUBSTR('Hello world', 7)	World
SUBSTR('Hello world', 1, 5)	Hello
INSTR('Hello world', 'world')	7
LPAD('12345', 10, '*')	*****12345
RPAD('12345', 10, '*')	12345*****
TRIM(' Hello world ')	Hello world
REPLACE('Hesso worsd', 's', 'l')	Hello world

The following statements show the use of the above functions in SELECT query.

```
SELECT CONCAT(FIRST_NAME, LAST_NAME) NAME, JOB_ID
FROM EMPLOYEES
WHERE INSTR( UPPER(JOB_ID), 'CLERK') > 0 ;


SELECT ( SUBSTR(FIRST_NAME, 1, 1) || '.' || SUBSTR(LAST_NAME, 1, 1) || '.' ) ABBR
FROM EMPLOYEES ;


SELECT INITCAP( TRIM(LAST_NAME) )  NAME, LPAD(SALARY, 10, 0)
FROM EMPLOYEES ;
```

The first query retrieves name and job id of all employees whose job id field contains the word 'CLERK'. The second query outputs the abbreviation of all employee names. The third query prints the last name and padded salary in 10 character width.

Practice 3.1

a. Print the first three characters and last three characters of all country names. Print in capital letters.

b. Print all employee full names (first name followed by a space then followed by last name). All names should be printed in width of 60 characters and left padded with '*' symbol for names less than 60 characters.

    c.   Print all job titles that contain the text 'manager'.

2.   **Number functions**

ROUND, TRUNC, and MOD functions

There are three number functions most commonly used in Oracle. They are given below.

Function	Description
ROUND (Column, n)	Rounds the value in Column up-to n decimal places after the decimal point. If n is 0, then rounding occurs in the digit before the decimal point.
TRUNC (Column, n)	Truncates the value up-to n decimal places after the decimal point. If n is 0, the truncation occurs up-to the digit before the decimal point.
MOD(m, n)	Returns the value of the remainder for m divided by n

The following table shows the results of applying numeric functions.

Function	Output
ROUND(45.923, 2)	45.92
ROUND(45.926, 2)	45.93
ROUND(45.926, 0)	46
TRUNC(45.923, 2)	45.92
TRUNC(45.926, 2)	45.92
TRUNC(45.926, 0)	45
MOD(23, 5)	3

The following statements show the use of numeric functions.

```
SELECT LAST_NAME, ROUND(SALARY/1000, 2) "SALARY (IN THOUSANDS)"
FROM EMPLOYEES
WHERE INSTR(JOB_ID, 'CLERK') > 0 ;


SELECT LAST_NAME, ROUND( (SYSDATE - HIRE_DATE)/7, 4) WEEKS_EMPLOYED
FROM EMPLOYEES
```

```
WHERE DEPARTMENT_ID = 80 ;


SELECT LAST_NAME, TRUNC(SALARY/1000, 0) || ' thousands ' ||
TRUNC( MOD(SALARY,1000)/100, 0) || ' hundreds ' || MOD(SALARY,100) || ' taka only'
FROM EMPLOYEES ;
```

Practice 3.2

    a.   Print employee last name and number of days employed. Print the second information rounded up to 2 decimal places.

    b.   Print employee last name and number of years employed. Print the second information truncated up to 3 decimal place.

3.   Date functions

Use of SYSDATE function

The SYSDATE function returns the current database server date and time. You can use this function as follows in SELECT query.

```
SELECT (SYSDATE - HIRE_DATE)/7 "WEEKS EMPLOYED"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 80 ;
```

Note the use of double quotes around "WEEKS EMPLOYED". It is necessary as the aliasing text contains spaces. The SYSDATE is used here to find the number of days an employee has worked, and then this number is divided by 7 to find the number of weeks, the employee passed in the company.

Date arithmetic

You can use arithmetic operators to subtract dates, add some numeric value with dates, etc. In the previous query, you subtracted SYSDATE from HIRE_DATE, and the result is the number of days between these two dates. The following table explains the outcomes of different arithmetic operators that can be applied on DATE type values.

Operation	Outcome

DATE + number	DATE value. Adds the number of days with the given DATE value.
DATE – number	DATE value. Subtracts the number of days with from the given DATE value.
DATE – DATE	Outcome is a numeric value specifying the number of days between these two dates.
DATE + DATE	Invalid operation

Suppose the value of HIRE_DATE column for an employee is '05-FEB-1995'. Then, the following table shows the use date arithmetic operations.

Operation	Outcome
HIRE_DATE + 7	'12-FEB-1995' which will be a DATE type value
HIRE_DATE – 4	'01-FEB-1995' which will be a DATE type value
SYSDATE – HIRE_DATE	Number of days between '01-FEB-1995' and today.

Date manipulation functions

The most commonly used date manipulation functions are given below.

Function	Outcome
MONTHS_BETWEEN(date1, date2)	Number of months between date1 and date2.
ADD_MONTHS(date1, n)	Adds n months with the date. Result is another date after addition of the months.
ROUND(date, 'MONTH')	Rounds the date to the nearest month. This results date corresponding to either 1st day of the next month or 1st day of the same month.
ROUND(date, 'YEAR')	Rounds the date to the nearest year. This results date corresponding to either 1st day of the next yearor 1st day of the same year.
TRUNC(date, 'MONTH')	Truncates the date to the start of the month.
TRUNC(date, 'YEAR')	Truncates the date to the start of the year.

If we assume SYSDATE = '20-MAR-14' then the following table illustrates some outputs of the date functions.

Expression	Outcome
ADD_MONTHS(SYSDATE, 5)	20-AUG-14
ROUND(SYSDATE, 'MONTH')	01-APR-14
ROUND(date, 'YEAR')	01-MAR-14
TRUNC(date, 'MONTH')	01-JAN-14
TRUNC(date, 'YEAR')	01-JAN-14

The following statements show the user of date manipulation functions.

```
SELECT LAST_NAME, MONTHS_BETWEEN(SYSDATE, HIRE_DATE) MON_EMPLOYED
FROM EMPLOYEES ;
```

Practice 3.3

a. For all employees, find the number of years employed. Print first names and number of years employed for each employee.

b. Suppose you need to find the number of days each employee worked during the first month of his joining. Write an SQL query to find this information for all employees.

4. **Functions for manipulating NULL values**

NVL function

The NVL function works on a column and outputs a specific value whenever the column value is NULL. The general syntax of this function is given below:

```
NVL(expr1, expr2)
```
NVL= None Value Logic, NVL(expr,replace_value)

If expr1 evaluated to NULL, the NVL function outputs expr2, otherwise, output is expr1.

The following query outputs COMMISSION_PCT value for all employees. Whenever, COMMISSION_PCT is NULL, the output shows a 0 instead of NULL.

```
SELECT LAST_NAME, NVL(COMMISSION_PCT, 0)
FROM EMPLOYEES ;
```

The NVL function can also be used in where to check if a column contains NULL value. For example, the following statement selects all employee records, whose COMMISSION_PCT value is NULL. We assume that, usually COMMISSION_PCT value is not negative.

```
SELECT *
FROM EMPLOYEES
WHERE NVL(COMMISSION_PCT, -1) = -1 ;
```

A better use of the NVL function is shown below where total annual salary is calculated for all employees. Without the NVL function, the expression SALARY*12 + SALARY*12*COMMMISSION_PCT would result in NULL whenever COMMISSION_PCT is NULL.

```
SELECT LAST_NAME,
(SALARY*12 + SALARY*12*NVL(COMMISSION_PCT, 0) ) ANNSAL
FROM EMPLOYEES
WHERE NVL(COMMISSION_PCT, -1) = -1 ;
```

Practice 3.4

    a. Print the commission_pct values of all employees whose commission is at least 20%. Use NVL function.

    b. Print the total salary of an employee for 5 years and 6 months period. Print all employee last names along with this salary information. Use NVL function assuming that salary may contain NULL values.

5. **Data Type Conversion Functions**

Oracle Automatic (Implicit) Type Conversion

Oracle can convert automatically the following whenever it requires -

- Convert a NUMBER value to VARCHAR2 value if used in text operations such as concatenation.

- Convert a VARCHAR2 value to NUMBER value in an arithmetic expression provided VARCHAR2 text represents a valid number. Otherwise Oracle would generate an error.

- Convert a DATE value to VARCHAR2 value (in default date format). This conversion is applied in most operations like comparison, concatenation, etc.

- Convert a VARCHAR2 value to a date value provided that VARCHAR2 text is in a default date format. Otherwise, automatic conversion fails. The default date format in Oracle is 'DD-MON-YY' or 'DD-MON-YYYY'.

The following table illustrates this in more detail using examples.

Operation	Description
MOD('25', 3)	Works fine. Although, '25' is a text, Oracle automatically convert the text to a number before applying the MOD
ADD_MONTHS('31-JAN-2011', 5)	Works fine. Converts the text '01-JAN-2011' automatically to a DATE value, then adds 5 months. Result will be '30-JUN-2011'
ADD_MONTHS('JAN/31/2011', 5)	Fails! The text is not in default format, so Oracle cannot automatically convert the text to DATE before ADD_MONTHS works.
CONCAT('Today is ', SYSDATE)	Here, converts the DATE value to VARCHAR2 before concatenation.
'Hello' || 123	Works fine. Oracle converts the number 123 to a text of VARCHAR2 before concatenation.
512 + '123'	Works fine. Oracle converts the text '123' to a number.
512 – 'hello'	Fails! Oracle generates an error because it cannot convert the text to a number for arithmetic operation.
SYSDATE – '01-MON-1998'	Converts the text to a DATE value. Then applies DATE arithmetic.

So, whenever, default conversion does not work, we need to use manual conversion functions.

Explicit type conversion: TO_CHAR function

The TO_CHAR function is used to convert DATE value to VARCHAR2 value. The general syntax of the function is given below:

```
TO_CHAR(date, format)
```

The format string specifies how to convert the DATE value to VARCHAR2 value. The following table illustrates the format texts that are most commonly applied to DATE value. Assume that, the value of HIRE_DATE is '31-JAN-1995'.

Expression	Output (a VARCHAR2 value)
TO_CHAR(HIRE_DATE, 'DD/MM/YYYY')	'31/01/1995'
TO_CHAR(HIRE_DATE, 'MONTH DD, YYYY')	'JANUARY 31, 1995'
TO_CHAR(HIRE_DATE, 'MONTH DD, YEAR')	'JANUARY 31, NINETEEN HUNDRED AND NINETY FIVE
TO_CHAR(HIRE_DATE, 'Ddspth Month, YEAR')	'Thirty-First January, Nineteen hundred and ninety five'
TO_CHAR(123456)	'123456'

The last example in above table shows that, TO_CHAR function can applied to numeric values also.

Explicit type conversion: TO_NUMBER function

The TO_NUMBER function converts any text of VARCHAR2 to a numeric value. This is very much essential when computing arithmetical operations on two or more columns where all columns were defined of type VARCHAR2. In such cases, we must explicitly convert them to number before operation.

Consider the following query that retrieves all locations in ascending order of POSTAL_CODE.

```
SELECT STREET_ADDRESS, POSTAL_CODE
FROM LOCATIONS
ORDER BY POSTAL_CODE ASC ;
```

However, the above query may not give correct ordering because POSTAL_CODE column is of type VARCHAR2. So, Oracle will do the ordering lexicographically ('1000' will come before '999'). To do numerical order, you can use the following query instead.

```
SELECT STREET_ADDRESS, POSTAL_CODE
FROM LOCATIONS
ORDER BY TO_NUMBER(POSTAL_CODE) ASC ;
```

Explicit type conversion: TO_DATE function

The TO_DATE function converts VARCHAR2 type text strings to DATE type value. The general syntax of the function expression is given below.

```
TO_DATE(text, format)
```

The following table illustrates the use of TO_DATE function with examples.

Expression	Output (a DATE type value)
TO_DATE('DEC 31 1995', 'MON DD YYYY')	Works fine.
TO_DATE('31/12/1995', 'DD/MM/YYYY')	Works fine.
TO_DATE('1995-12-31', 'YYY-MM-DD')	Works fine.
TO_DATE('1995-12-31', 'YYY-DD-MM')	Fails! Format does not match with the given text.

You are always encouraged to use explicit date conversions in comparison operations. Otherwise, you query may bring unexpected results. For example, the following query finds all employee last names who was hired before 1st January 1997.

```
SELECT LAST_NAME, TO_CHAR(HIRE_DATE, 'DD-MON-YYYY') HD
FROM EMPLOYEES
WHERE HIRE_DATE < TO_DATE('01-JAN-1997', 'DD-MON-YYYY')
ORDER BY HIRE_DATE ASC ;
```

Practice 3.5

    a.   Print hire dates of all employees in the following formats:

       (i) 13th February, 1998 (ii) 13 February, 1998.

6.   **More Practice Problems**

In class.

Chapter 4: Aggregate Functions

1.   **Retrieve Aggregate Information: Simple GROUP BY queries**

Group functions

In previous chapter, we have studied single-row functions. Those functions operates on one single row and outputs something based on the input columns of the row. The group functions operate on a group of rows and outputs a value such as total, average, etc. These functions report only summary information per group, rather than per row.

The group functions result in only one row per group of rows in the output. The most commonly used group functions are given below.

Function	Description
SUM (Column)	Finds the total, i.e., summation of values in Column for all rows in the group
MAX(Column)	Finds the maximum value in Column for all rows in the group
MIN(Column)	Finds the minimum value in Column for all rows in the group
AVG(Column)	Finds the average value in Column for all rows in the group
COUNT(Column)	Count the number of non-NULL values in Column for all rows in the group

The group functions given above discard NULL values during their computation.

Use of group function in SELECT statement

The group functions are used in the SELECT statement to retrieve information by groups. The groups are identified by GROUP BY clause. The general syntax is given below. These queries are called GROUP BY queries and are very important to database for generating various types of reports.

```
SELECT Column1, Column2, … , SUM(Column1), MAX(Column1), …
FROM table_name
GROUP BY Column1, Column2, …
```

In the above statement, Oracle will first create several groups based on <Column1, Column2,…>. Every unique combination of these columns will denote a group and all rows in the table that have these combinations of values will be in the group. Note that, only the columns in the GROUP BY clause can be selected along with group function expressions.

For example, suppose, you want to create a report showing the total salary paid by the company to each departments. This requires a GROUP BY query and group function as shown below.

```
SELECT DEPARTMENT_ID, SUM(SALARY)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID ;
```

For another example, suppose, you want to know the maximum salary, minimum salary and average salary the company pays in different job types. The following GROUP BY query will retrieve the required information.

```
SELECT JOB_ID, MAX(SALARY), MIN(SALARY), AVG(SALARY)
FROM EMPLOYEES
GROUP BY JOB_ID ;
```

Note that, you cannot select a column that is not present in GROUP BY clause. For example, the following query would return an error.

```
SELECT JOB_ID, JOB_TITLE, COUNT(*) TOTAL
FROM JOBS
GROUP BY JOB_ID ;
```

For the third example, suppose you want to know the number of employees working in each department. As you may recall, this requires creating a group based on DEPARTMENT_ID column, then count the number of rows in each group, and then output the information. So, you may think, the following query will output the information.

```
SELECT DEPARTMENT_ID, COUNT(LAST_NAME)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID ;
```

In the above statement, counting the LAST_NAME column is similar to counting any other columns. However, COUNT(LAST_NAME) will ignore the NULL values in LAST_NAME field, therefore not counting rows that contain NULL values in LAST_NAME field. The correct query should be the following.

```
SELECT DEPARTMENT_ID, COUNT(*)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID ;
```

In the above, COUNT(*) counts number of rows (NULL and non-NULL) and therefore will give the correct result.

Omitting GROUP BY clause in group function query

You can omit the GROUP BY clause in group function query. In such cases, the entire table will form a single group, and summary information will be calculated for whole table as one group only.

The following statements find the maximum, minimum, and average salary for the entire table, i.e., all employees of the company. The second statement finds the number of rows in EMPLOYEE table.

```
SELECT MAX(SALARY), MIN(SALARY), AVG(SALARY)
FROM EMPLOYEES ;

SELECT COUNT(*)
FROM EMPLOYEES ;
```

Use of WHERE clause in GROUP BY query

In a GROUP BY statement, the WHERE clause can be used for filtering rows before grouping is applied. The general syntax of such statements is given below.

```
SELECT Column1, Column2, … , SUM(Column1), MAX(Column1), …
FROM table_name
WHERE condition
GROUP BY Column1, Column2, …
```

The following statement finds the maximum and minimum salary for each job types for only the employees working in the department no. 80.

```
SELECT JOB_ID, MAX(SALARY), MIN(SALARY)
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 80
GROUP BY JOB_ID ;
```

Use of ORDER BY clause in GROUP BY query

In a GROUP BY statement, ORDER BY clause can be used to sort the final group results based on grouping column. The following example sorts the final results based on JOB_ID value.

```
SELECT JOB_ID, MAX(SALARY), MIN(SALARY)
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 80
GROUP BY JOB_ID
ORDER BY JOB_ID ASC ;
```

Practice 4.1

   a. For all managers, find the number of employees he/she manages. Print the MANAGER_ID and total number of such employees.

   b. For all departments, find the number of employees who get more than 30k salary. Print the DEPARTMENT_ID and total number of such employees.

   c. Find the minimum, maximum, and average salary of all departments except DEPARTMENT_ID 80. Print DEPARTMENT_ID, minimum, maximum, and average salary. Sort the results in descending order of average salary first, then maximum salary, then minimum salary. Use column alias to rename column names in output for better display.

2. DISTINCT and HAVING in GROUP BY queries

Use of DISTINCT keyword in Group functions

You can use DISTINCT keyword inside the group functions as given below. In this case, group functions consider only unique rows, and discards duplicate values. So, duplicate value is counted only once for calculation.

```
SELECT JOB_ID, MAX(DISTINCT SALARY), MIN(DISTINCT SALARY),
SUM(DISTINCT SALARY), COUNT(DISTINCT DEPARTMENT_ID)
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 80
GROUP BY JOB_ID
ORDER BY JOB_ID ASC ;
```

Note that, the use of DISTINCT does not change the result of MAX and MIN group function But, it does change the outcomes of SUM, and COUNT functions.

Use of HAVING Clause to discard groups

You can use the HAVING clause to remove some group information from the final results. Suppose, you want to retrieve the maximum and minimum salaries of each department except the department no. 80. The following GROUP BY statement will retrieve the required information. The information for group no. 80 are removed using a HAVING clause. HAVING works like WHERE except that HAVING works on groups.

```
SELECT DEPARTMENT_ID, MAX(SALARY), MIN(SALARY)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID
HAVING DEPARTMENT_ID <> 80 ;
```

Note that, the HAVING condition is applied after group information is calculated unlike WHERE condition is applied before grouping.

The following statements show some more examples of HAVING clause.

```
SELECT JOB_ID, MAX(SALARY), MIN(SALARY)
FROM EMPLOYEES
GROUP BY JOB_ID
HAVING MAX(SALARY) > 5000 ;


SELECT JOB_ID, AVG(SALARY)
FROM EMPLOYEES
GROUP BY JOB_ID
HAVING AVG(SALARY) <= 5000
ORDER BY AVG(SALARY) DESC ;
```

The first statement above outputs maximum and minimum salary for each job types. However, only those results are printed where group maximum is greater than 5000. The second query retrieves average salary of each job type. This time, only those group results are printed for which average salary is less than or equal to 5000. Results are sorted in descending order of average salary.

Practice 4.2

      a. Find for each department, the average salary of the department. Print only those DEPARTMENT_ID and average salary whose average salary is at most 50k.

3.   **Advanced GROUP BY queries**

Expression in GROUP BY clause

You can use expressions in group by clause. Expressions help you to retrieve more complex information from the table.

The following query finds the total employees hired in each year.

```
SELECT TO_CHAR(HIRE_DATE, 'YYYY') YEAR, COUNT(*) TOTAL
FROM EMPLOYEES
GROUP BY TO_CHAR(HIRE_DATE, 'YYYY')
ORDER BY YEAR ASC ;
```

The following query finds the total number of employees whose name starts with the same character. The query reports the first character and total employees.

```
SELECT SUBSTR(FIRST_NAME, 1, 1) FC, COUNT(*) TOTAL
FROM EMPLOYEES
GROUP BY SUBSTR(FIRST_NAME, 1, 1)
ORDER BY FC ASC ;
```

More than one column in GROUP BY clause

Sometimes you will need to use multiple columns in group by. For example, the following query finds the number of employees for each department and for each job type. One column would not be sufficient to find this group information.

```
SELECT DEPARTMENT_ID, JOB_ID, COUNT(*) TOTAL
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID, JOB_ID ;
```

An important reason for adding more than one column in a GROUP BY clause is to select more columns. Consider the following query that finds the number of employees for each job type.

```
SELECT JOB_ID, COUNT(*) TOTAL
FROM JOBS
GROUP BY JOB_ID ;
```

Now, if we want to print job titles along with job id in the above query, we has to put the job title column in the GROUP BY clause, because Oracle does not allow to select columns in the SELECT clause that are not present in the GROUP BY clause.

```
SELECT JOB_ID, JOB_TITLE, COUNT(*) TOTAL
FROM JOBS
GROUP BY JOB_ID, JOB_TITLE ;
```

Practice 4.3

    a. Find number of employees in each salary group. Salary groups are considered as follows. Group 1: 0k to <5K, 5k to <10k, 10k to <15k, and so on.

    b. Find the number of employees that were hired in each year in each job type. Print year, job id, and total employees hired.

4. **More Practice Problems**

In class.

Chapter 5: Query Multiple Tables – Joins

1.   **Oracle Joins to Retrieve Data from Multiple Tables**

Data from multiple tables

All queries used in previous chapters fetched data from one table only. Sometimes, we need to retrieve data from multiple tables. For example, suppose you want to create a report containing last names of employee, salary, and name of the department employee works in. The first two fields are available in EMPLOYEES table (LAST_NAME and SALARY column). But, the third field is available in DEPARTMENTS table. To know in which department an employee works, we need to join rows from EMPLOYEES table with rows from DEPARTMENT table based on the DEPARTMENT_ID field. This type of joins is done by join queries.

By join operation, rows of two or more tables are joined together to form larger rows. The rows of different tables are joined together based on some columns that may be present in both tables.

Joining two tables by USING clause

Rows from two tables can be joined together by USING clause. The general syntax of this query is given below.

```
SELECT …
FROM table1 JOIN table2 USING (Column1, Column2, …)
WHERE …
GROUP BY …
ORDER BY …
```

The following example statement joins EMPLOYEES and DEPARTENT tables based on the values of DEPARTMENT_ID column. A row from EMPLOYEES table is joined with a row from DEPARTMENTS table where both rows have the same DEPARTMENT_ID value.

```
SELECT E.LAST_NAME, D.DEPARTMENT_NAME
FROM EMPLOYEES E JOIN DEPARTMENTS D USING (DEPARTMENT_ID) ;
```

```
SELECT EMPLOYEES.LAST_NAME, EMPLOYEES.SALARY, DEPARTMENTS.DEPARTMENT_NAME
FROM EMPLOYEES JOIN DEPARTMENTS USING(DEPARTMENT_ID);
THIS IS ALSO VALID. SO, USING ALIAS IS NOT MANDATORY FOR OUTER JOINS!
```

Note the use of table aliases in the above statement. The table aliases are used in SELECT clause to choose columns from the tables. However, you cannot alias the column, i.e., DEPARTMENT_ID which is used in the USING clause. Otherwise, Oracle will generate error message. The following query clarifies this by not aliasing the DEPARTMENT_ID column in the SELECT clause.

```
SELECT E.LAST_NAME, DEPARTMENT_ID, D.DEPARTMENT_NAME
FROM EMPLOYEES E JOIN DEPARTMENTS D USING (DEPARTMENT_ID) ;
```

Joining two tables by ON clause

Two tables can be joined using the ON clause which is more general than USING. The ON clause specifies an explicit condition on which rows will be joined together. In case of USING, rows are joined based on same values (an equality condition) in join column. But, in case of ON clause, the condition can be specified explicitly allowing equality and non-equality conditions for join.

The following statement shows the same join as before but using ON clause.

```
SELECT E.LAST_NAME,  E.DEPARTMENT_ID,  D.DEPARTMENT_NAME
FROM EMPLOYEES E JOIN DEPARTMENTS D
ON (E.DEPARTMENT_ID = D.DEPARTMENT_ID) ;
```

The more general nature of the ON clause will be clear if you observer the following statements.

Self-join: joining a table with itself using ON clause

A table can be joined with itself using ON clause which is called self-join. This is very useful in many cases. Suppose you want to know the last name of manager for each employee. Each row in the EMPLOYEES table has a MANAGER_ID column which is actually an EMPLOYEE_ID of the same table since the manager is also an employee. To retrieve the last name of the manager for each employee, we need to join two copies of EMPLOYEES table based on MANAGER_ID field.

The following statement shows how the join happens.

```
SELECT E1.LAST_NAME EMPLOYEE, E2.LAST_NAME MANAGER
FROM EMPLOYEES E1 JOIN EMPLOYEES E2
ON (E1.MANAGER_ID = E2.EMPLOYEE_ID) ;
```

Joins using non-quality condition

Tables can be joined by non-equality condition by ON clause. The following statement shows such a join operation. Assume that there is table named JOB_GRADES that has LOWEST_SAL, HIGHEST_SAL, and GRADE_LEVEL columns. We want to retrieve an employee's grade based on his salary. In this case, we need to join EMPLOYEES table with JOB_GRADES table. However, a row from the EMPLOYEES table should join with that row from the JOB_GRADES table whose LOWEST_SAL and HIGHEST_SAL contains the SALARY value of the employee.

```
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)  ;
```

Self-join allows many interesting queries. Suppose you want to create a report showing last name of an employee and a number which is the number of employees getting higher salary than the employee. This type of results can be calculated using joins and groupings. The following statement computes this.

```
SELECT E1.LAST_NAME, COUNT(*) HIGHSAL
FROM EMPLOYEES E1 JOIN EMPLOYEES E2
ON (E1.SALARY < E2.SALARY)
GROUP BY E1.EMPLOYEE_ID, E1.LAST_NAME
ORDER BY E1.LAST_NAME ASC ;
```

Observe that in the above query, the E1.LAST_NAME column was not required to be put in the GROUP BY clause unless it was selected in SELECT clause.

Joining more than two tables   IN ORACLE SQL, JOINING OCCURS FROM LEFT TO RIGHT.

Three or more tables can also be joined using ON clause. The following statement shows such a query that joins three tables.

```
SELECT E.LAST_NAME, D.DEPARTMENT_NAME, L.CITY
FROM EMPLOYEES E
JOIN DEPARTMENTS D
ON (E.DEPARTMENT_ID = D.DEPARTMENT_ID)
JOIN LOCATIONS L
ON (D.LOCATION_ID = L.LOCATION_ID) ;
```

Oracle left outer join and right outer join

The general JOIN operation keeps rows that are found matched in both tables based on join condition. In some cases, we need to keep the rows of first table (or second table or both) in the output that does not meet join criteria. The normal JOIN operation would not keep such rows. In such cases, we need to use LEFT OUTER JOIN and RIGHT OUTER JOIN syntax.

The LEFT OUTER JOIN keeps rows of the left table that did not have any match with rows of right table based on join criteria.

The RIGHT OUTER JOIN keeps rows of the right table that did not have any match with rows of the left table based on join criteria.

The application of such syntax is illustrated in the following query. The query finds the number of employees managed by each employee. If an employee does not mange anyone, then the output displays 0.

```
SELECT E1.LAST_NAME, COUNT(E2.EMPLOYEE_ID) TOTAL
FROM EMPLOYEES E1 LEFT OUTER JOIN EMPLOYEES E2
ON (E1.EMPLOYEE_ID = E2.MANAGER_ID)
GROUP BY E1.EMPLOYEE_ID, E1.LAST_NAME
ORDER BY TOTAL ASC ;
```

If you replace COUNT(E2.EMPLOYEE_ID) by COUNT(*) in the above query, then the query would not output correct results! Find the reason yourself! Morever, E1.LAST_NAME does not any effect on grouping. It was included in the grouping clause so that it can be selected in the output.

Practice 5.1

  a. For each employee print last name, salary, and job title.

  b. For each department, print department name and country name it is situated in.

  c. For each country, finds total number of departments situated in the country.

  d. For each employee, finds the number of job switches of the employee.

  e. For each department and job types, find the total number of employees working. Print department names, job titles, and total employees working.

              Author: Sukarna Barua

f. For each employee, finds the total number of employees those were hired before him/her. Print employee last name and total employees.

g. For each employee, finds the total number of employees those were hired before him/her and those were hired after him/her. Print employee last name, total employees hired before him, and total employees hired after him.

`Out of the scope of this chapter`

h. Find the employees having salaries greater than at least three other employees

i. For each employee, find his rank, i.e., position with respect to salary. The highest salaried employee should get rank 1 and lowest salaried employee should get the last rank. Employees with same salary should get same rank value. Print employee last names and his/he rank.

j. Finds the names of employees and their salaries for the top three highest salaried employees. The number of employees in your output should be more than three if there are employees with same salary.

Chapter 6: Query Multiple Tables – Sub-query

1.   **Retrieving Records using Sub-query**

What is a sub-query

A sub-query is a SELECT statement which is used inside another SELECT statement. The sub-query can be place in FROM clause, WHERE clause, HAVING clause, etc. The use of sub-query in the WHERE clause makes SELECT statements easier and powerful for retrieving information.

There are special comparison operators when sub-queries are used in WHERE clause. They are ANY, ALL. The general syntax of sub-query in a WHERE clause is given below.

```
SELECT Column1, Column2, …
FROM table_name
WHERE sub-query-with-condition
```

The sub-query executes before the main query and results of sub-query are used in main query.

Use of sub-query in WHERE clause

Suppose you want to know the last names and salaries of all employees whose salary is greater than Abel's salary. The following sub-query solution will retrieve the information efficiently.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE SALARY >
(
SELECT SALARY
FROM EMPLOYEES
WHERE LAST_NAME = 'Abel'
) ;
```

Sub-query can result in one row or multiple rows. In the above statement, sub-query is first executed, and SALARY value of Abel is retrieved. Then, this value is used in the main query.

The following is another example where the main query retrieves information of those employees whose JOB_ID is same as the employee numbered 141.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE JOB_ID =
(
SELECT JOB_ID
FROM EMPLOYEES
WHERE EMPLOYEE_ID = 141
) ;
```

Use of multiple sub-queries in single statement

The following statement shows the use of multiple sub-queries in single statement. The statement retrieves records of those employees whose JOB_ID is same as employee numbered 114 and whose SALARY is greater than Abel's SALARY.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE JOB_ID =
(
SELECT JOB_ID
FROM EMPLOYEES
WHERE EMPLOYEE_ID = 141
)
AND SALARY >
(
SELECT SALARY
FROM EMPLOYEES
WHERE EMPLOYEE_ID = 141
)  ;
```

Use of group functions in sub-query

Suppose you want to retrieve the name of the employee who gets highest salary among all employees. The following statement retrieves the information using sub-query.

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE  SALARY =
(
SELECT MAX(SALARY)
FROM EMPLOYEES
)  ;
```

Sub-query returns more than one row

In case sub-query returns more than one row, usual comparison operators must be used with ANY or ALL keyword. Their meanings are

- ANY – If used with comparison operation, the outcome will be true if operator evaluates to true for any of the sub-query values
- ALL – if used with comparison operator, the outcome will be true only if operator evaluates to true for all values returned by the sub-query

To understand the use of ANY and ALL, examine the following statement. The query retrieves those employee records (working in other than 'IT_PROG' department) whose SALARY is less than at least one employee of 'IT_PROG'.

```
SELECT LAST_NAME, JOB_ID, SALARY
FROM EMPLOYEES
WHERE  JOB_ID <> 'IT_PROG'
AND SALARY < ANY
(
SELECT SALARY
FROM EMPLOYEES
WHERE JOB_ID = 'IT_PROG'
)  ;
```

In the query below, ALL is used instead of ANY. This query retrieves those employee records whose SALARY is less than all employees of 'IT_PROG'.

```
SELECT LAST_NAME, JOB_ID, SALARY
FROM EMPLOYEES
WHERE  JOB_ID <> 'IT_PROG'
```

```
AND SALARY < ALL
(
SELECT SALARY
FROM EMPLOYEES
WHERE JOB_ID = 'IT_PROG'
)  ;
```

Practice 6.1

a. Find the last names of all employees that work in the SALES department.

b. Find the last names and salaries of those employees who get higher salary than at least one employee of SALES department.

c. Find the last names and salaries of those employees whose salary is higher than all employees of SALES department.

d. Find the last names and salaries of those employees whose salary is within $\pm$ 5k of the average salary of SALES department.

2. **Advanced Sub-query**

Correlated Sub-query

In a correlated sub-query, we use row references of the main query to in the sub-query. Suppose, you need to retrieve those employees whose salary is higher than at least three other employees. To write this query using sub-query, we need to do the following:

```
SELECT *
FROM EMPLOYEES E1
WHERE 3 <= (
SELECT COUNT(*)
FROM EMPLOYEES E2
WHERE E2.SALARY < E1.SALARY
)  ;
```

In the above query, the sub-query in the WHERE clause retrieve those rows form EMPLOYEES table who earn less salary than the salary of the current employee. The current is determined by the execution of this

query by Oracle. Oracle retrieves all records of the main query first. Then for each row (say *X*) of this main query (outer query), Oracle executes the sub-query by setting E1.SAL value to the salary value of row *X*. Then sub-query results are retrieved, conditions are evaluated, and then X goes to final output if condition becomes true. If main query returns *N* rows, then the inner sub-query will be executed *N* times.

Correlated Sub-query with EXISTS clause

The EXISTS operator checks whether any row is found. A sub-query is used as an operand of EXISTS operation. The operator would then return true if some rows were retrieved by the sub-query. Otherwise, it would return false.

The following query illustrate the behavior of EXISTS operator. The query retrieves those departments which have at least one employee having JOB_ID as 'IT_PROG'.

```
SELECT DEPARTMENT_NAME
FROM DEPARTMENTS D
WHERE  EXISTS
(
SELECT *
FROM EMPLOYEES E
WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID AND JOB_ID = 'IT_PROG'
) ;
```

The following query uses NOT EXISTS to find those employees whose earns the maximum salary in his/her department.

```
SELECT LAST_NAME, SALARY, DEPARTMENT_ID
FROM EMPLOYEES E1
WHERE  NOT EXISTS
(
SELECT *
FROM EMPLOYEES E2
WHERE E2.DEPARTMENT_ID = E1.DEPARTMENT_ID AND
E2.SALARY > E1.SALARY
)  ;
```

Sub-query in SELECT Clause

You can use sub-query in SELECT Clause. For example, you could have re-written the previous query by writing a sub-query in the SELECT clause to display names of departments instead of department id. The following query shows that.

```
SELECT LAST_NAME, SALARY, (SELECT DEPARTMENT_NAME FROM DEPARTMENTS D
                                    WHERE  D.DEPARTMENT_ID  =  E1.DEPARTMENT_ID)
DEPARTMENT
FROM EMPLOYEES E1
WHERE  NOT EXISTS
(
SELECT *
FROM EMPLOYEES E2
WHERE E2.DEPARTMENT_ID = E1.DEPARTMENT_ID AND
E2.SALARY > E1.SALARY
)  ;
```

Sub-query in FROM clause

We can also sue sub-query in FROM clause. If a sub-query is placed in a FROM clause, the sub-query results will form a temporary table. The rest of the query will work the same as before such as join, grouping, etc.

To illustrate this, suppose we need to show the last name and salary of each employee along with the minimum salary and maximum salary of his/her department. This query can easily be solved using a self-join as shown below.

```
SELECT E1.LAST_NAME, E1.SALARY, MIN(E2.SALARY), MAX(E2.SALARY)
FROM EMPLOYEES E1 JOIN EMPLOYEES E2
ON (E1.DEPARTMENT_ID = E2.DEPARTMENT_ID)
GROUP BY E1.DEPARTMENT_ID, E1.LAST_NAME, E1.SALARY
ORDER BY E1.SALARY ;
```

The above query can be solve by placing a sub-query in the FROM clause as shown below.

```
SELECT E.LAST_NAME, E.SALARY, D.MINSAL, D.MAXSAL
FROM EMPLOYEES E,
```

```
      (
      SELECT DEPARTMENT_ID AS DEPT, MIN(SALARY) MINSAL, MAX(SALARY) MAXSAL
      FROM EMPLOYEES
      GROUP BY DEPARTMENT_ID
      ) D
WHERE (E.DEPARTMENT_ID = D.DEPT)
ORDER BY E.SALARY ;
```

The sub-query in the FROM clause above creates temporary table whose schema is D(DEPT, MINSAL, MAXSAL). Then Oracle joins E with D and rest of the query executes as usual.

Practice 6.2

a. Find those employees whose salary is higher than at least three other employees. Print last names and salary of each employee. *You cannot use join in the main query*. Use sub-query in WHERE clause only. You can use join in the sub-queries.

b. Find those departments whose average salary is greater than the minimum salary of all other departments. Print department names. Use sub-query. You can use join in the sub-queries.

c. Find those department names which have the highest number of employees in service. Print department names. Use sub-query. You can use join in the sub-queries.

d. Find those employees who worked in more than one department in the company. Print employee last names. *You cannot use join in the main query*. Use sub-query. You can use join in the sub-queries.

e. For each employee, find the minimum and maximum salary of his/her department. Print employee last name, minimum salary, and maximum salary. Do not use sub-query in WHERE clause. Use sub-query in FROM clause.

f. For each job type, find the employee who gets the highest salary. Print job title and last name of the employee. Assume that there is one and only one such employee for every job type.

Chapter 7: Set operations

1.   **Performing Set Operations**

Set operators

The general syntax of set operations is given below.

```
SELECT Column1, Column2, … , ColumnN
FROM table_name
…
SET-OPERATOR
(
SELECT Column1, Column2, … , ColumnN
FROM table_name
…
)
```

The following must be met for the sub-query to be successful-

- Number of columns must be same

- The data type of each column in the second query must match the data type of its corresponding column in the first query.

The following set-operators are available in Oracle.

Set operator	Description
UNION	Combines results of two queries eliminating duplicate rows
UNION ALL	Combines results of two queries keeping duplicate values
INTERSECT	Finds the intersection of results of two queries
MINUS	Rows in the first query that are not present in the second query

UNION and UNION ALL operators

The following statement shows the use of UNION operator. The query does not keep duplicate values.

```
SELECT EMPLOYEE_ID, JOB_ID
FROM EMPLOYEES
UNION
(
SELECT EMPLOYEE_ID, JOB_ID
FROM JOB_HISTORY
) ;
```

The UNION ALL operator combines results of two queries and keeps duplicate values.

```
SELECT EMPLOYEE_ID, JOB_ID
FROM EMPLOYEES
UNION ALL
(
SELECT EMPLOYEE_ID, JOB_ID
FROM JOB_HISTORY
) ;
```

INTERSECT operator

The intersect operator performs the set intersection. Rows that are common in both queries are retrieved in the output. The following statement finds the employees whose current job title is same as one of their previous job titles.

```
SELECT EMPLOYEE_ID, JOB_ID
FROM EMPLOYEES
INTERSECT
(
SELECT EMPLOYEE_ID, JOB_ID
FROM JOB_HISTORY
) ;
```

MINUS operator

The MINUS operator performs set minus operation. Rows of first query that are not in the second query will be retrieved for output. The following query finds the employees who have not changed their jobs even once.

```
SELECT EMPLOYEE_ID, JOB_ID
FROM EMPLOYEES
MINUS
(
SELECT EMPLOYEE_ID, JOB_ID
FROM JOB_HISTORY
) ;
```

Use of conversion functions in set operations

In case of set operators, it is mandatory that data types of the columns in both queries must match. In that case, we may need to use type conversion functions to explicitly match data types. For example, assume, there is an EMPLOYEES2 table in which JOB_ID was defined as a numeric value. Now, if you want to find the employee records (LAST_NAME, JOB_ID, SALARY of both tables (EMPLOYEES and EMPLOYEEES2), you may issue the following statement.

```
SELECT LAST_NAME, JOB_ID, SALARY
FROM EMPLOYEES
UNION ALL
(
SELECT LAST_NAME, JOB_ID, SALARY
FROM EMPLOYEES2
) ;
```

But, the above statement would generate an error message as the JOB_ID fields of the two tables have different data types. So, in order to match the data types explicitly, you will require converting the data type of the second query to VARCHAR2 as follows.

```
SELECT LAST_NAME, JOB_ID, SALARY
FROM EMPLOYEES
UNION ALL
```

```
(
SELECT LAST_NAME, TO_CHAR(JOB_ID), SALARY
FROM EMPLOYEES2
) ;
```

Practice 7.1

     a. Find EMPLOYEE_ID of those employees who are not managers. Use minus operator to perform this.

     b. Find last names of those employees who are not managers. Use minus operator to perform this.

     c. Find the LOCATION_ID of those locations having no departments.

2.   **More Practice Problems**

In class.

Chapter 8: Data Manipulation Language (DML)

DML Statements

DML statements are used to:

- Adding new rows to a table – INSERT statements

- Changing data in a table – UPDATE statements

- Removing rows from a table – DELETE statements

- Transactions controls in database – COMMIT, ROLLBACK statements

1. Inserting Data into Table

INSERT statement

The INSERT statement is used to insert new row in a table. INSERT statement allows you to insert one or more rows to the table. The general syntax of the INSERT statement is given below:

```
INSERT INTO table_name (column1, column2, column3, …)
VALUES (value1, value2, value3, ...) ;
```

For example, the following statement inserts a new row in the DEPARTMENTS table.

```
INSERT INTO DEPARTMENTS (DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID)
VALUES (179, 'Public Relations', 100, 2600) ;
```

Inserting rows without column names unspecified

You can insert values in a table without column names unspecified in the insert statement. However, in this case, you have to ensure the following:

- Values are given for all columns in the table

- Values are given in the default order of the columns as defined in the table creation statements.

For example, the following statement would successfully insert a row in the DEPARTMENTS table.

```
INSERT INTO DEPARTMENTS VALUES (189, 'PRINTING_STATIONARY', 100, 2600) ;
```

However, the following insert statement would not work and will generate an Oracle error message. The reason is that, the DEPARTMENTS table has four columns, but in the statement only values are specified for two columns only.

```
INSERT INTO DEPARTMENTS VALUES (170, 'Public Relations') ;
```

The following insert statement will also fail. Even though, all values are specified, they are not in correct order as defined in the original table definition.

```
INSERT INTO DEPARTMENTS VALUES (170, 100, 'Public Relations', 1700) ;
```

Inserting rows with NULL values for some columns

You can insert rows in a table with NULL values in some columns. In this case, you have to specify the column names of the table explicitly as shown in the following query.

```
INSERT INTO DEPARTMENTS (DEPARTMENT_ID, DEPARTMENT_NAME)
VALUES (279, 'Purchasing') ;
```

In the above INSERT statement, rows will successfully be inserted. Since, values for MANAGER_ID and LOCATION_ID columns were not specified; they will be filled with NULL. Note that, this will work only if these columns satisfy the following conditions:

- None of these columns is part of a primary key of the table

- None of these columns has NOT NULL constraints

Inserting Date type values

If the column is of DATE type, then you may use the TO_DATE conversion function to convert a text to DATE type value. The following examples show such a conversion to insert a new employee in the EMPLOYEES table.

```
INSERT INTO EMPLOYEES VALUES (279, 'Den', 'Raphealy', 'Drapheal', '515.127.4515',
TO_DATE ( 'FEB 13, 1999', 'MON DD, YYYY'), 'SA_REP', 11000, 0.2, 100, 60) ;
```

However, if you specify the text in default date format, i.e, 'DD-MON-YYYY', then TO_DATE function is not required as shown in the following statement. Oracle will automatically convert the text to DATE type value. **However, it is a good practice to always use the TO_DATE function to explicitly convert to DATE values.**

```
INSERT INTO EMPLOYEES VALUES (279, 'Den', 'Raphealy', 'Drapheal', '515.127.4515',
'13-FEB-1999', 'SA_REP', 11000, 0.2, 100, 60) ;
```

Inserting rows from another table

You can insert several rows from a table directly into another table. The following statement copies all rows from EMPLOYEES table into another table named EMPLOYEES2. Note that, VALUES keyword is not required in the statement.

```
INSERT INTO Employees2
Select * from employees ;
```

You can also use sub-query and where condition to copy some specific rows as shown in the following statement.

```
INSERT INTO SALES_EMPLOYEES (ID, NAME, SALARY, COMMISSION_PCT)
SELECT EMPLOYEE_ID, LAST_NAME, SALARY, COMMISSION_PCT
FROM EMPLOYEES
WHERE JOB_ID LIKE '%REP%' ;
```

Some Common Mistakes of INSERT statements

The oracle server automatically enforces all data types, data ranges, and data integrity constraints. Common errors that may occur during INSERT statements are following:

- Value missing for a column which has NOT NULL constraint

- Duplicate value violating UNIQUE, or PRIMARY KEY constraint

- Any value violating a CHECK constraint

- Data type mismatch or value too large to fit in the column

- Referential integrity violation for FOREIGN KEY constraint.

2. **Changing Data in a Table**

UPDATE Statement

The UPDATE statement is used to change data in a table. The general syntax of the UPDATE statement is given below. The statement can be used to change data in one or more columns. The WHERE clause can be used to change data for some particular rows and is optional.

```
UPDATE table_name
SET Column1 = value1, Column2 = value2, …
WHERE condition ;
```

The following example changes DEPARTMENT_ID of the employee numbered 113.

```
UPDATE EMPLOYEES
SET DEPARTMENT_ID = 50
WHERE EMPLOYEE_ID = 113 ;
```

To change the DEPARTMENT_ID of all employees, i.e., all rows in the table, omit the WHERE clause as shown below:

```
UPDATE EMPLOYEES
SET DEPARTMENT_ID = 110 ;
```

You can update multiple column values as shown below.

```
UPDATE EMPLOYEES
SET JOB_ID = 'IT_PROG', COMMISSION_PCT = NULL
WHERE EMPLOYEE_ID = 114 ;
```

Use of sub-query in UPDATE statement

You can use a sub-query in the UPDATE statement to fetch date from other tables. For example, the following statement updates job and salary of employee numbered 113 to match those of employee number 205.

```
UPDATE EMPLOYEES SET
JOB_ID = (SELECT JOB_ID FROM EMPLOYEES WHERE EMPLOYEE_ID = 205),
SALARY = (SELECT SALARY FROM EMPLOYEES WHERE EMPLOYEE_ID = 205)
WHERE EMPLOYEE_ID = 113 ;
```

Practice 8.2

    a.   Update COMMISSION_PCT value to 0 for those employees who have NULL in that column.

    b.   Update salary of all employees to the maximum salary of the department in which he/she works.

    c.   Update COMMISSION_PCT to N times for each employee where N is the number of employees he/she manages. When $N = 0$, keep the old value of COMMISSION_PCT column.

    d.   Update the hiring dates of all employees to the first day of the same year. Do not change this for those employees who joined on or after year 2000.

3.   Deleting Rows from a Table

DELETE Statement

The DELETE statement is used to remove rows from a table. The general syntax of the DELETE statement is given below. The WHERE condition is required to delete only some specified values. If the WHERE condition is omitted, then all rows are deleted from the table.

```
DELTE FROM table_name
WHERE condition ;
```

The following statement removes the Finance department row from the DEPARTMENTS table. *Actually, it will not remove any row due to violation of constraints!*

```
DELETE FROM DEPARTMENTS
WHERE DEPARTMENT_NAME = 'Finance' ;
```

The following statement removes two rows from the DEPARTMENT table. *Actually, it will not remove any row due to violation of constraints!*

```
DELETE FROM DEPARTMENTS
WHERE DEPARTMENT_ID IN (30, 40) ;
```

Note that, the following statement will remove all rows from the DEPARTMENTS table as there is no WHERE condition. *Actually, it will not remove any row due to violation of constraints!*

```
DELETE FROM DEPARTMENTS ;
```

Use of sub-query in the DELETE statement

You can use sub-query to delete rows from a table based on information from another table. The following statement removes all employees from the EMPLOYEES table who are working in the Finance department. *Actually, it will not remove any row due to violation of constraints!*

```
DELETE FROM EMPLOYEES
WHERE DEPARTMENT_ID =
(
SELECT DEPARTMENT_ID FROM DEPARTMENTS
WHERE UPPER(DEPARTMENT_NAME) LIKE '%FINANCE%'
)
```

Note that use of UPPER function in the above query. The UPPER function ensures that, the statement will work even in department names are stored in lower case or upper case letters.

Practice 8.3

      a. Delete those employees who earn less than 5k.

      b. Delete those locations having no departments.

      c. Delete those employees from the EMPLOYEES table who joined before the year 1997.

                                                              Author: Sukarna Barua

4.   **Database Transaction Controls Using COMMIT, ROLLBACK**

Database Transactions

A transaction means one or more SQL statements which together make a unit of work. All SQL statements should successfully execute or fail together. In Oracle database, a transaction is automatically started with the first DML statement use executes. The already-started transactions will end whenever one of the following events occurs:

- A COMMIT or ROLBACK statement is issued

- A DDL or DCL statement is issued (automatic COMMIT)

- The user exits SQL*DEVELOPER or SQL*PLUS (automatic COMMIT)! Do not forget this in your entire life!

- The system crashes (automatic ROLLBACK) or SQL*PLUS stopped unexpectedly (automatic ROLLBACK).

After one transaction ends, another transaction will start with the execution of next DML statement.

COMMIT statement

The COMMIT statement saves results of DML operations of the current transaction permanently in database. It also ends the ongoing transaction.

The following statements show the use of COMMIT to store data permanently to database. One row is deleted from the EMPLOYEES table and a row is added to the DEPARTMENTS table. Finally, COMMIT saves this changed permanently into database.

```
DELTE FROM EMPLOYEES
WHERE EMPLOYEE_ID = 99999 ;


INSERT INTO DEPARTMENTS
VALUES (290, 'Corporate Tax', NULL, 1700) ;


COMMIT ;
```

ROLLBACK Statement

The ROLLBACK statement undoes the results of all DML operations executed in the current transaction. It also ends the current transaction. The state of the tables will be restored in the previous values before the current transaction started.

The following statements when executed will not store the new row in the DEPARTMENTS table and will not remove the row from the EMPLOYEES table. The ROLLBACK statement will undo all the changes done by the DELETE and INSERT statements.

```
DELTE FROM EMPLOYEES
WHERE EMPLOYEE_ID = 99999 ;

INSERT INTO DEPARTMENTS
VALUES (290, 'Corporate Tax', NULL, 1700) ;

ROLLBACK ;
```

5.   **Practice Problems**

In class.

Chapter 9: Data Definition Language (DDL) Statements

DDL Statements

Most commonly used DDL statements are for:

- Creating tables in database – CREATE TABLE

- Specifying constraints in tables – NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK

- Deleting tables from database – DROP TABLE

1.   **Creating Tables**

CEATE TABLE statement

The CREATE TABLE statement is used to create new tables in database. The general syntax of the statement is given below:

```
CREATE TABLE [schema_name.]table_name
(
Column1 Datatype1,
Column2 Datatype2,
…
) ;
```

For example, the following example creates a new table in the database:

```
CREATE TABLE PERSON
(
NID VARCHAR2(15),
NAME VARCHAR2(50),
BDATE DATE
) ;
```

The above statement will create a table named PERSON in database. The table PERSON will have three columns which are:

- NID – Data type is VARCHAR2, maximum length 15 characters.

- NAME – Data type if VARCHAR2, maximum length 50 characters

- BDATE – Date type value

Naming rules

Table names and column names have to satisfy the following

- Must begin with a letter

- Must be 1-30 characters long

- Must contain only A-Z, a-z, 0-9, _, $, and #

- Must not duplicate the name of another object of the same user

- Must not be an Oracle keyword

Use the following commands to delete a table entirely from database/

```
DROP TABLE PERSON ;
```

DEFAULT option in CREATE TABLE

You can specify a DEFAULT option in column definition to specify default value for the column as given below:

```
CREATE TABLE PERSON
(
NID VARCHAR2(15),
NAME VARCHAR2(50),
BDATE DATE DEFAULT SYSDATE
) ;
```

The PERSON table created above will have a default value, i.e., SYSDATE for the BDATE column. So, if user does not specify any value for this parameter, then the default value will be used for the column. For example, the following INSERT statement does not specify value for the BDATE column. The value of SYSDATE will be used by default for the BDATE column.

```
INSERT INTO PERSON(NID, NAME) VALUES ('10010010010001', 'Sakib') ;
```

Data Types of Columns

When you are creating tables, you have to specify data type for the columns. The following commonly used data types are available in Oracle:

Data Types	Description
VARCHAR2(size)	Used to store variable length text data. A maximum size must be specified. The minimum size is 1 and maximum size is 4000.
CHAR(size)	Used to store fixed length text data. Minimum size is 1 and maximum size is 2000.
NUMBER(p, s)	Used to store numeric values. You can optionally specify precision (p) and scale (s) parameter values. Precision is the total number of decimal digits and scale is the number to the right of the decimal point.
DATE	Used to store date values.
CLOB	Used to store variable length text data (Up-to 4GB)
BLOB	Used to store binary data (Up-to 4GB)

Creating tables using a sub-query

You can create a table using a sub-query. In this case, table is created automatically with column definitions matched with the results of sub-query. The rows that are returned by the sub-query are also get inserted in the newly created table. The general syntax of this statement is given below:

```
CREATE TABLE table_name [(Column1, Column2, …)]
AS sub-query
```

For example, the following statement creates a table DEPT80 with a sub-query.

```
CREATE TABLE DEPT80
AS
SELECT EMPLOYEE_ID, LAST_NAME, SALARY*12 ANNSAL
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 80 ;
```

The DEPT80 table created above –

- Will contain three columns named as EMPLOYEE_ID, LAST_NAME, and ANNSAL.

- The data types of the three columns are inherited from the EMPLOYEES table. So, the data types of the three columns reflect the same as in the EMPLOYEES table.

- The newly created table will inherit all constraints that are defined on the EMPLOYEE_ID, LAST_NAME columns in the EMPLOYEE table.

- The newly created table will contain all data rows of employees whose DEPARTMENT_ID column value is 80.

- Note that, the column alias ANNSAL for the expression SALARY*12 is necessary here. Otherwise, Oracle will generate an error message.

2.   **Specifying Constraints in Tables**

Constraints

Constraints are used in table definitions to specify rules so that invalid data is not entered in database. Usually, constraints do the following:

- Enforce rules on the data. Whenever, a row is inserted, updated, or deleted, these rules are checked against the data. Constraints must be satisfied for the operation to be successful.

- Constraints prevent the deletion of important rows from a table, which may have dependencies in other tables

The commonly used Oracle constraints are following:

Constraint	Description
NOT NULL	Specify that the column can not contain NULL values
UNIQUE	The column values in different rows must be unique, however, NULL values are allowed for multiple rows.
PRIMARY KEY	This constraint is used for a column that uniquely identifies each row of a table. Column values must be unique and cannot be NULL. So, this is equivalent to NOT NULL and UNIQUE constraints together.
FOREGIN KEY	Used to create referential integrity checks. Values in a column of one table must have similar values in a column of another table.
CHECK	This specifies a condition which must be true for all rows.

Defining constraints

Constraints can be defined in column-level and table-level. The general syntax of column-level constraint definition is as follows:

```
CREATE TABLE [schema_name.]table_name
(
…,
Column datatype [CONSTRAINT constraint_name] constraint_type,
…
) ;
```

The general syntax of table-level constraint definition is given below:

```
CREATE TABLE [schema_name.]table_name
(
Column1 Datatype1,
Column2 Datatype2,
…,
[CONSTRAINT constraint_name] constraint_type (Column1, Column2, …),
…
) ;
```

The following statements show column-level constraints:

```
CREATE TABLE PERSON
(
NID VARCHAR2(15) CONSTRAINT PERSON_PK PRIMARY KEY,
NAME VARCHAR2(50) NOT NULL,
BDATE DATE DEFAULT SYSDATE
) ;
```

The above statement –

- Creates PERSON table with two column-level constraints

Author: Sukarna Barua

- NID column is the PRIMARY KEY of the table. So, NID will uniquely specify each row of the table. Moreover, NID cannot be NULL for any row inserted in the table.

- The PRIMARY KEY constraint of NID column has been given a name, i.e., PERSON_PK

- NAME column cannot have NULL values. This constraint is not given any name.

The following statement creates the same constraints, but this time, constraints are defined in table-level with no difference.

```
DROP TABLE PERSON;

CREATE TABLE PERSON
(
NID VARCHAR2(15),
NAME VARCHAR2(50) NOT NULL UNIQUE,
BDATE DATE DEFAULT SYSDATE,
CONSTRAINT PERSON_PK PRIMARY KEY (NID)
) ;
```

In some cases, two columns together may uniquely identify each row of a table. So, PRIMARY KEY is composed of two column values rather than a single column. This type of constraints cannot be defined using column-level constraints. They must be defined using table-level constraints as shown below:

```
DROP TABLE PERSON;

CREATE TABLE PERSON
(
COUNTRYID CHAR(3),
PERSONID VARCHAR2(15),
NAME VARCHAR2(50) NOT NULL UNIQUE,
BDATE DATE DEFAULT SYSDATE,
CONSTRAINT PERSON_PK PRIMARY KEY (COUNTRYID, PERSONID)
) ;
```

In the above PERSON table, the PRIMARY KEY is composed of two columns, COUNTRYID, and PERSONID.

FOREIGN KEY constraint

A FOREIGN KEY constraint establishes a relationship between columns in one table with another column (which is a PRIMARKY KEY) of another table. Consider the PERSON table created below:

```
DROP TABLE PERSON;

CREATE TABLE PERSON
(
NID VARCHAR2(15) CONSTRAINT PERSON_PK PRIMARY KEY,
NAME VARCHAR2(50) NOT NULL,
BDATE DATE DEFAULT SYSDATE
) ;
```

Now, consider the following ADDRESS table, which stores the address of each person. This person must be a valid row of the PERSON table.

```
CREATE TABLE PERSON_ADDRESS
(
PID VARCHAR2(15) CONSTRAINT PERSON_ADDRESS_PK PRIMARY KEY,
ADDR_LINE1 VARCHAR2(50) NOT NULL,
ADDR_LINE2 VARCHAR2(50),
CITY VARCHAR2(50) NOT NULL,
DISTRICT VARCHAR2(50) NOT NULL,
CONSTRAINT PERSON_ADDRESS_FK FOREIGN KEY(PID) REFERENCES PERSON(NID)
)  ;
```

The PID column in the PERSON_ADDRESS table is defined as the PRIMARY KEY of the table. Moreover, this column is also specified as a FOREIGN KEY constraint which is linked (by REFERENCES Keyword) to NID column of PERSON table. This link will ensure that if a value is inserted in the PID column of PERSON_ADDRESS table, the same value must also be present in the NID column of a row in the PERSON table. Otherwise, the insertion will fail. The PERSON table will be called parent table and PERSON_ADDRESS table will be called a child table.

The FOREIGN KEY constraints create a problem during deletion of rows from the parent table. If a row is to be deleted from the parent table, but the child table have some rows which are dependent (because of the

FOREIGN KEY constraint) on that row, then Oracle will not allow deletion of the row. All dependent rows in the child table must be deleted manually before the deletion of a row in the parent table.

However, there are two solutions available in Oracle. The FOREIGN KEY constraints can have following optional keywords at the end of the constraint definition –

- ON DELETE CASCADE – When a row is deleted from the parent table, the dependent rows are also deleted from the child table automatically by Oracle.

- ON DELETE SET NULL – When a row is deleted from the parent table, the dependent rows are set to NULL values in the child table automatically by Oracle.

CHECK constraints

The CHECK constraints define one or more conditions that must be satisfied by the column values of a row. These constraints can be defined as column-level or table-level. The general syntax of the CHECK constraint is given below:

```
CHECK ( condition )
```

For example, the following table definition contains a CHECK constraint in column-level which ensure that, the SALARY value cannot be negative.

```
DROP TABLE EMPLOYEE2;

CREATE TABLE EMPLOYEE2
(
EID VARCHAR2(15) CONSTRAINT EMPLOYEE2_PK PRIMARY KEY,
SALARY NUMBER CONSTRAINT EMPLOYEE_SAL_MIN CHECK (SALARY > 0)
) ;
```

The above CHECK constraint –

- Is defined in column-level

- Constraint has been given a name, i.e., EMPLOYEE_SAL_MIN

- Condition of the constraint is: SALARY > 0

3.   Deleting Tables from Database

DROP TABLE statement

The DROP table statement is used for removing tables from database. The general syntax of the statement is given below:

```
DROP TABLE table_name [PURGE] ;
```

For example, the following statement removes the EMPLOYEES2 table from the database.

```
DROP TABLE EMPLOYEES2 ;
```

Note that, the table structure is deleted and all rows are deleted. But, the space used by the table is not released. To release the storage used by the table, you need to specify PURGE option at the end of the DROP TABLE statements as shown below:

```
DROP TABLE EMPLOYEES2 PURGE ;
```

4.   Add or Remove Table Columns

You can add new column to an existing table using the ALTER TABLE SQL command. For example, suppose we want to add a new column BDATE to our EMPLOYEES table whose type will be of date. The following command will add the column.

```
ALTER TABLE EMPLOYEES ADD BDATE DATE ;
```
To delete a column from a table, use the ALTER TABLE DROP command like below.

```
ALTER TABLE EMPLOYEES DROP COLUMN BDATE ;
```

5.   More Practice Problems

In class.

Chapter 10: Creating Views, Sequences, and Indexes

Database objects

The database contains different kinds of objects, among which most common are:

- Tables – stores data, created with CREATE TABLE statement.

- Views – stores sub set of data from tables, created with CREATE VIEW statement.

- Sequence – Generate numeric values, created with CREATE SEQUENCE statement

- Index – Improves performance of database, created with CREATE INDEX statement

1. **Creating Views**

What is a View

A view –

- Is defined by a sub-query, i.e., a SELECT statement

- Is a logical table based on one or more tables or views

- Contains no data of its own, and contain no data at the physical level.

- When user executes query on the view, the view sub-query gets executed and view data are fetched dynamically

CREATE VIEW statement

A view is created by the CREATE VIEW statement. The general syntax of the statement is given below:

```
CREATE [OR REPLACE] VIEW view_name
AS sub-query ;
```

For example, the following statement creates a view named VIEW80.

```
CREATE OR REPLACE VIEW DEPTV80
AS
SELECT EMPLOYEE_ID, LAST_NAME, SALARY*12 ANNSAL
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 80 ;
```

The above created view DEPTV80 –

- Contain three columns EMPLOYEE_ID, LAST_NAME, ANNSAL

- The view does not contain any physical records at the time of creation.

- The sub-query is not executed. It will be executed later when view data are required by another query on the view.

- OR REPLACE clause is optional and if it is specified, then the view is created even though a view with the same name already exists. The old view is deleted.

- The column alias ANNSAL is necessary here, otherwise Oracle will generate an error message.

You can issue SQL SELECT statements on the created view to retrieve data as follows:

```
SELECT * FROM DEPTV80 ;


SELECT EMPLOYEE_ID, ANNSAL FROM DEPTV80 ;
```

The following shows another example of CREATE VIEW statement.

```
CREATE OR REPLACE VIEW DEPT_SUMMARY
AS
SELECT D.DEPARTMENT_NAME DEPT, MIN(E.SALARY) MINSAL, MAX(E.SALARY) MAXSAL,
AVG(E.SALARY) AVGSAL
FROM EMPLOYEES E  JOIN DEPARTMENTS D
ON (E.DEPARTMENT_ID = D.DEPARTMENT_ID)
GROUP BY D.DEPARTMENT_ID, D.DEPARTMENT_NAME ;
```

The above created view –

- Will have four columns, DEPT, MINSAL, MAXSAL, and AVGSAL.

- The column aliases in the sub-query are necessary; otherwise Oracle will generate error messages.

- Since, the view sub-query contains JOIN and GROUP BY clause, DEPT_SUMMARY will be a complex view. In complex view, no DML operations are allowed unlike the simple views where DML operations are allowed.

Advantages of view over tables

There are some advantages of creating views over tables which are –

- Views give an option to restrict data access to a table.

- Views can make complex query easier.

- Views give different view of the same table to different users of the database.

- Views do not contain any record of its own which saves a lot of space of the database than copying a table multiple times.

Removing views from database

You can remove views from the database by DROP VIEW statement. The following example removes the DEPTV80 view from the database. Note that, no data is deleted from database since view does not contain any physical data.

```
DROP VIEW DEPTV80 ;
```

2. Creating Sequences

What is a Sequence

A sequence is a database objects that create numeric (integer) values. The sequence is usually used to create values for a table column, e.g., PID column in the PERSON table defined in previous chapters.

CREATE SEQUENCE statement

To create a sequence CREATE SEQUENCE statement is used. The general syntax of this statement is given below:

```
CREATE SEQUENCE sequence_name
[INCREMENT BY n]
[START WITH n]
[MAXVALUE n]
[MINVALUE n]
[CYCLE | NOCYCLE]
```

Remember the PERSON table created in previous chapter with the following statement.

```
CREATE TABLE PERSON
(
```

```
NID VARCHAR2(15) CONSTRAINT PERSON_PK PRIMARY KEY,
NAME VARCHAR2(50) NOT NULL,
BDATE DATE DEFAULT SYSDATE
) ;
```

Now, the following statement creates a sequence named PERSON_NID_SEQ to be used for inserting rows in PERSON table.

```
CREATE SEQUENCE PERSON_NID_SEQ
INCREMENT BY 1
START WITH 100000000000001
MAXVALUE 999999999999999
NOCYCLE ;
```

The above created sequence –

- Will generate numeric values starting with 100000000000001

- The next value is found by adding 1 with previous value

- The maximum value will be 999999999999999

- The NOCYLE option ensures that after the maximum value is generated, the sequence will stop generating value. If you want the sequence to generate values again from the starting value, use the CYCLE option instead of NOCYCLE option.

Using the sequence

The created sequence PERSON_NID_SEQ will be used for inserting data in the PERSON table. The sequence will provide sequential numbers for the NID column of the PERSON table. The NEXTVAL is used to retrieve next sequential number from the sequence as shown by the following INSERT statements:

```
INSERT INTO PERSON
VALUES (PERSON_NID_SEQ.NEXTVAL, 'Ahsan', TO_DATE('12-DEC-1980','DD-MON-YYYY') )
;
```

You can view the current value of the sequence by the following query:

```
SELECT PERSON_NID_SEQ.CURRVAL
```

```
FROM DUAL ;
```

Note that use of DUAL table here.

3. **Creating Indexes**

What is an Index

An index –

- Is created on a column of a database table

- Provides fast access to the table data using the column on which index is defined

- Is used by the Oracle server to retrieve query results efficiently.

- Reduces disk input outputs for retrieving query results.

- Are automatically managed by the Oracle server after user creates the index.

CREATE INDEX statement

An index is created automatically on those columns of the table which have PRIMARY KEY or UNIQUE constraints. To create index on other columns, CREATE INDEX statement is used. The general syntax of this statement is given below:

```
CREATE INDEX index_name
ON table_name (Column1, Column2, … ) ;
```

For example, the following statement creates an index on the NAME column of the PERSON table defined above. This will enable faster access of data based on NAME column. That means, if the NAME column is used in WHERE clause of a query, then Oracle can retrieve query results more quickly than if no index was present.

```
CREATE INDEX PERSON_NAME_IDX
ON PERSON (NAME) ;
```

You can drop an index using DROP INDEX statement as shown below:

```
DROP INDEX PERSON_NAME_IDX ;
```

When to create indexes?

In general, you should create an index on a column only when one or more of the following conditions are satisfied –

- The column contains wide range of values

- The column contains large number of NULL values

- The column is frequently used in WHERE clause condition

- The table is large and most queries are expected to retrieve only less than 2% to 4% rows.

4. **More Practice Problems**

In class.

Chapter 11: Advanced Expression in Queries

1. **Decode**

What is a decode function?

Decode is an Oracle function that works like an if-then-else statement. It can be used in an expression to return conditional results. The general structure of the Decode function is shown below.

```
DECODE(EXPR,MATCH_EXPR,RESULT_EXPR,DEFAULT_EXPR)
```

In the above,

- EXPR is the first expression which is to be matched

- MATCH_EXPR is the second expression which is matched with EXPR1

- RESULT_EXPR is the return value expression of the DECODE function if EXPR1 matches with EXPR2(EXPR1=EXPR2)

- DEFAULT_EXPR is the value returned by the DECODE function if EXPR1 does not match with EXPR2. If no DEFAULT value is specified, then NULL is returned by ORACLE.

Note that the data type of all RESULT_EXPRs and DEFAULT_EXPR must be the same. Otherwise Oracle will throw an error.

Use of DECODE in SELECT clause

Assume we want to find the out the JOB_IDs which are Managers. Now, the JOBS table contain the necessary information. To retrieve the JOB_IDs which are managers, we can issue the following query:

```
SELECT JOB_ID, DECODE(JOB_TITLE,'Marketing Manager','Manager','Not a manager')
FROM JOBS;
```

The above query will show the text 'Manager' or 'Not a manager' if the corresponding JOB_ID is a manager. However, the above query is not correct as the second expression in DECODE is only comparing with 'Marketing Manager'. There are other types of managers in the table which would not be shown as

'Manager'. To correctly show 'Manager' for all JOB_IDs which are some types of managers, we can issue the following query:

```
SELECT JOB_ID, DECODE(INSTR(UPPER(JOB_TITLE),'MANAGER'),0,'Not a manager',
'Manager')
FROM JOBS
```

Note the use of UPPER and INSTR functions:

- UPPER is used to convert to uppercase before comparison as the table might contain texts in different cases

- INSTR returns 0 if the second string 'MANAGER' is not found in the column JOB_TITLE

- DEOCDE compares the result of the first expression with 0 and returns 'Not a manager' if equal; otherwise returns 'Manager'.

Note that the value of EXPR1 and EXPR2 in DECODE cannot be a Boolean., i.e., value cannot be the result of a relational or logical expression. It may be changed in future versions of Oracle.

DECODE is not limited to a single if-then-else statement. The more general syntax is below:

```
DECODE(EXPR, MATCH_EXPR1, RESULT_EXPR1, MATCH_EXPR2, RESULT_EXPR2,…,
DEFAULT_EXPR) --DEFAULT_EXPR is optional
```

The above expression works as:

- If EXPR = MATCH_EXPR1, returns RESULT_EXPR1

- IF EXPR = MATCH_EXPR2, returns RESULT_EXPR2,

- And so on, and finally DEFAULT_EXPR if does not match with any expressions.

Suppose we want to grade the employees based on their salaries as below:

- Salary < 5000, Grade = C

- Salary >= 5000 and Salary < 10000, Grade = B

- Salary >= 10000 and Salary < 15000, Grade = A

- Salary >= 15000, Grade = A+

The following query shows the salary grades as required using DECODE expression:

```
SELECT FIRST_NAME, SALARY, DECODE(FLOOR(SALARY/5000),0,'C',1,'B',2,'A','A+')
"SALARY GRADE"
FROM EMPLOYEES
ORDER BY SALARY DESC
```

Note the use FLOOR to convert the floating-point result of the expression SALARY/5000 into an integer value before comparing.

DECODE can also be used in WHERE clause as well also its most use-cases is in the SELECT clause.

2. CASE

What is a CASE expression?

Oracle CASE expression is similar to the DECODE function. It is an if-then-else statement that returns specific values based on the result of a condition. The general syntax of the CASE expression can be written in two ways which are shown below:

```
CASE
      WHEN COMPARISON_EXPR1 THEN RESULT_EXPR1
      WHEN COMPARISON_EXPR2 THEN RESULT_EXPR2

      …
      ELSE DEFAULT_EXPR
END


Or


CASE EXPR
      WHEN CONDITION_EXPR1 THEN RESULT_EXPR1
      WHEN CONDITION_EXPR2 THEN RESULT_EXPR2

      …
      ELSE DEFAULT_EXPR
END
```

In the firs CASE expression, COMPARISON_EXPR1 should be a relationship or logical expression result while the latter CASE expression compares the main expression EXPR with CONDITION_EXPRs similar to how DECODE compares.

Use of CASE in SELECT clause

Let's solve the salary grade calculation example shown previously for DECODE using CASE expression. The employees will be categories in different grades based on the salaries. The below query uses CASE expression to solve the problem:

```
SELECT FIRST_NAME, SALARY,
CASE
        WHEN SALARY < 5000 THEN 'C'
        WHEN SALARY < 10000 THEN 'B'
        WHEN SALARY < 15000 THEN 'A'
        ELSE 'A+'
END "SALARY GRADE"
FROM EMPLOYEES
ORDER BY SALARY DESC
```

Not that we use the "SALARY < 5000" as the first comparison and thus don't need to add the comparison "SALARY>=5000" in the second expression. Oracle evaluates the second comparison when the first comparison results in false.

Use of CASE in aggregation query

CASE expression can be used in aggregation queries to retrieve and display complex results. Suppose we want to display the total number of employees in each salary grade as obtained by the previous case query. We need to group all employees based on the salary grades and the count the total using aggregation query. There are other ways to calculate this without using CASE. However, we will show here how we can use the CASE expression to find this.

Examine the following query that uses CASE in GROUP BY expression:

```
SELECT (CASE
        WHEN SALARY < 5000 THEN 'C'
        WHEN SALARY < 10000 THEN 'B'
        WHEN SALARY < 15000 THEN 'A'
```

```
       ELSE) 'A+'
END "SALARY GRADE", COUNT(*)
FROM EMPLOYEES
GROUP BY (CASE
       WHEN SALARY < 5000 THEN 'C'
       WHEN SALARY < 10000 THEN 'B'
       WHEN SALARY < 15000 THEN 'A'
       ELSE) 'A+'
END
ORDER BY "SALARY GRADE" ASC
```

The GROUP BY expression uses the "SALARY GRADE" as computed by the CASE expression to group the employee records. We are not permitted to use alias "SALARY GRADE" in the GROUP BE expression. The parenthesis to enclose the CASE expression is not mandatory and is used only for aesthetic purpose.

Now, suppose you want to compute the number of employees of different salary grades for each department individually. You want to display the total number in each salary grade beside the department ID of the department. The following query displays the required information using CASE expression:

```
SELECT DEPARTMENT_ID,
SUM(CASE WHEN SALARY > 15000 THEN 1 ELSE 0 END) "A+",
SUM(CASE WHEN SALARY >= 10000 AND SALARY < 15000 THEN 1 ELSE 0 END) "A",
SUM(CASE WHEN SALARY >= 5000 AND SALARY < 10000 THEN 1 ELSE 0 END) "B",
SUM(CASE WHEN SALARY < 5000 THEN 1 ELSE 0 END) "C"
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID
ORDER BY DEPARTMENT_ID ASC
```

Note that we need to use four CASE expressions to calculate the number of employees in different salary grades. Each expression is within an aggregate SUM function that sums either 1 or 0 depending on whether the corresponding row's salary is within the range of the salary grade.

3.   **With clause**

What is WITH clause?

Oracle WITH clause can be used to separately write a subquery and then reference it in the main query. Previously we saw how we can write a subquery in the FROM clause to create a temporary table and then join it with other tables. The WITH clause is somewhat similar to that expression. An example syntax of WITH clause is shown below:

```
WITH TEMPTABLE AS (A SUBQUERY)
SELECT …
FROM TEMPTABLE
WHERE …
```

In the above expression, the subquery inside the WITH clause creates a temporary table which can then reference in the main query as like a table in the database. Note that Oracle may or may not actually create any temporary table or view for the WITH clause subquery. For our understanding, we can just assume it as if a temporary table is created in the database.

Use of WITH clause

Suppose we want to find for each employee the number of employees working in his/her department. This means we want to display employee name and a number which is the total employee working the employee's department. Using a WITH clause, we can write the following query to retrieve the information from the database.

```
WITH EMPCOUNTBYDEPT AS
(
      SELECT DEPARTMENT_ID, COUNT(*) EMPCOUNT
      FROM EMPLOYEES
      GROUP BY DEPARTMENT_ID
)
SELECT E1.FIRST_NAME, E2.EMPCOUNT
FROM EMPLOYEES E1 JOIN EMPCOUNTBYDEPT E2 ON (E1.DEPARTMENT_ID = E2.DEPARTMENT_ID)
ORDER BY E1.EMPLOYEE_ID ASC
```

In the above query, EMPTCOUNTBYDEPT is referenced in the main query to retrieve the number of employees working in each department as if it was a table in the database.

Note that the same query can also be written using the subquery in the FROM clause as shown below.

```
SELECT E1.FIRST_NAME, E2.EMPCOUNT
FROM EMPLOYEES E1 JOIN
(
       SELECT DEPARTMENT_ID, COUNT(*) EMPCOUNT
       FROM EMPLOYEES
       GROUP BY DEPARTMENT_ID
) E2 ON (E1.DEPARTMENT_ID = E2.DEPARTMENT_ID)
ORDER BY E1.EMPLOYEE_ID ASC
```

Things get complicated if we want to display more information. Suppose, we want to display total number of employees working in the employee's department and total number of employees working in the employees' manager's department. So, we need to fetch the manager of each employee and find the total number of employees working in the department of the manager. The following query solves this using WITH clause.

```
WITH EMPCOUNTBYDEPT AS
(
       SELECT DEPARTMENT_ID, COUNT(*) EMPCOUNT
       FROM EMPLOYEES
       GROUP BY DEPARTMENT_ID
)
SELECT E1.FIRST_NAME, E3.EMPCOUNT "Emp Count Employee Dept", E4.EMPCOUNT "Emp
Count Maneger Dept"
FROM EMPLOYEES E1 JOIN EMPLOYEES E2 ON (E1.MANAGER_ID = E2.EMPLOYEE_ID)
JOIN EMPCOUNTBYDEPT E3 ON (E1.DEPARTMENT_ID = E3.DEPARTMENT_ID)
JOIN EMPCOUNTBYDEPT E4 ON (E2.DEPARTMENT_ID = E4.DEPARTMENT_ID)
ORDER BY E1.EMPLOYEE_ID ASC
```

The first join retrieves the manager's information. The second join retrieves the number of employees working the employee's department and the third join retrieves the number of employees working in the manager's department.

Practice 11.3

    a. Calculate the number of employees in different salary grades for each department using COUNT aggregation function instead of SUM.

    b. Calculate the number of employees in different salary grades for each department using DECODE instead of CASE.

    c. Write the query to show total employees working in the employee's department and in the employee's manager's department without using WITH clause. You can use subqueries in the FROM clause.

Chapter 12: Introduction to PL/SQL

1.   **Anonymous PL/SQL blocks**

General structure of Anonymous blocks

The general structure of PL/SQL block is given below.

```
DECLARE
      <declarative section>
      --variables are declared here
BEGIN
      <executable section>
      --processing statements are written here
EXCEPTION
      <exception code>
      --exception handling codes are written here if any
END ;
```

The different sections of PL/SQL block are summarized below:

- Declarative (after DECLARE) – Contains declaration of variables, constants, etc. This is an optional section.

- Executable (after BEGIN) – Contains SELECT statements to retrieve data from tables and contain PL/SQL statements to manipulate those data. This is a mandatory section.

- Exception (after EXCEPTION) – This is an optional section. Contains exception handling codes.

The following statement shows an anonymous block that outputs 'Hello world'.

```
BEGIN
      DBMS_OUTPUT.PUT_LINE('Hello World') ;
END ;
/
```

Note the following:

- Every statement in a block ends with a semi-colon (;)

- DBMS_OUT.PUT_LINE ( ) is a function that is used to print messages from a PL/SQL block.

Execute an anonymous block

To execute the anonymous block written above, you need to copy paste the entire code into SQL*PLUS. Do not forget to copy the forward slash (/) also. At first, you may not see any output. To view PL/SQL outputs in the SQL*PLUS, you need to issue following command before executing the script above.

```
SET SERVEROUTPUT ON
```

Use of variables in PL/SQL

For programming, you will need to declare variables to store temporary values. The following script declares a variable named ENAME. The SQL query in the executive section retrieves full name of the employee numbered 100. Then, output the result.

```
DECLARE
      ENAME VARCHAR2(100) ;
BEGIN
      SELECT (FIRST_NAME || LAST_NAME) INTO ENAME
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = 100 ;
      DBMS_OUTPUT.PUT_LINE('The name is : ' || ENAME) ;
END ;
/
```

Note that use INTO keyword to store value from SELECT statement into PL/SQL variable (ENAME).

Use of SQL functions in PL/SQL

You can use SQL functions in PL/SQL statements. The following statement outputs the number of months employee 100 worked in the company.

```
DECLARE
      JDATE DATE ;
      MONTHS NUMBER ;
      RMONTHS NUMBER ;
BEGIN
      SELECT HIRE_DATE INTO JDATE
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = 100 ;
```

```
        MONTHS   :=  MONTHS_BETWEEN(SYSDATE, JDATE) ;
        RMONTHS  :=  ROUND(MONTHS, 0) ;
        DBMS_OUTPUT.PUT_LINE('The employee worked ' || RMONTHS || ' months.') ;
END ;
/
```

Note that, the assignment operator in PL/SQL is colon-equal (:=). The equal (=) operator is the equality comparison operator and cannot be used as an assignment operator. *Do not give space between the colon (:) and equal (=) of the colon-equal (:=) operator, otherwise it will generate an error.*

View errors of a PL/SQL block

In case, your PL/SQL block contains a syntactic error, Oracle will generate and show an error-code rather than showing the errors. To view the errors along with their line numbers, you need to run the following command:

```
SHOW ERRORS ;
```

Oracle will not compile the erroneous blocks. You must correct the errors and re-run the corrected code.

Use of IF-ELSE conditional statement

The IF-ELSE statement processes conditional statements. The general structure of IF-ELSE statement is given below.

```
IF condition THEN
      Statements ;
[ ELSIF condition THEN
      Statements ; ]
[ ELSE
      Statements ; ]
END IF ;
```

The following example shows the use of IF-ELSE in PL/SQL. The script finds whether employee numbered 100 worked for more than 10 years in the company.

```
DECLARE
      JDATE DATE ;
```

```
      YEARS NUMBER ;
BEGIN
      SELECT HIRE_DATE INTO JDATE
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = 100 ;
      YEARS :=  (MONTHS_BETWEEN(SYSDATE, JDATE) / 12) ;
      IF YEARS >= 10 THEN
            DBMS_OUTPUT.PUT_LINE('The employee worked 10 years or more') ;
      ELSE
            DBMS_OUTPUT.PUT_LINE('The employee worked less than 10 years') ;
      END IF ;
END ;
/
```

The following example shows another use of IF-ELSE statement to find the grade level of employee numbered 100.

```
DECLARE
      ESALARY NUMBER ;
BEGIN
      SELECT SALARY INTO ESALARY
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = 100 ;
      IF ESALARY < 1000 THEN
            DBMS_OUTPUT.PUT_LINE('Job grade is D') ;
      ELSIF ESALARY >= 1000 AND ESALARY < 2000 THEN
            DBMS_OUTPUT.PUT_LINE('Job grade is C') ;
      ELSIF ESALARY >= 2000 AND ESALARY < 3000 THEN
            DBMS_OUTPUT.PUT_LINE('Job grade is B') ;
      ELSIF ESALARY >= 3000 AND ESALARY < 5000 THEN
            DBMS_OUTPUT.PUT_LINE('Job grade is A') ;
      ELSE
            DBMS_OUTPUT.PUT_LINE('Job grade is A+') ;
      END IF ;
END ;
/
```

Do not forget to use THEN after each ELSIF clause. This is a common mistake everyone does.

Comments in PL/SQL blocks

You can put single-line comments and multi-line comments in a PL/SQL block. The syntax of both comments is given below.

```
--This is a single-line comment

/*
This is a multi-line
Comment.
*/
```

2.   Exception Handling in PL/SQL block

Handling exceptions

An Oracle statement can throw exceptions. For example, in our previous two blocks, the first SELECT statement can throw exception if not data is found in database for the given employee id.

When an Oracle statement throws an exception, the PL/SQL block immediately *exits* discarding later statements in the block. To examine this behavior, just run the above two blocks by modifying the employee id from 100 to 10000. Since, there is no employee numbered 10000 in the EMPLOYEES table, the SELECT statement would throw an exception and PL/SQL block would exit.

Sometimes, we need to show meaningful messages to the user when such an exception occurs. In this case, we will need somehow to handle the exception ourselves and generate those messages. In Oracle PL/SQL, we can handle an exception by writing an EXCEPTION section before the END statement of the block. The following example shows how to do this.

```
DECLARE
     JDATE DATE ;
     YEARS NUMBER ;
BEGIN
     --first retrieve hire_date and store the value into JDATE variable
     SELECT HIRE_DATE INTO JDATE
     FROM EMPLOYEES
     WHERE EMPLOYEE_ID = 10000 ;
     --calculate years from the hire_date field
     YEARS :=  (MONTHS_BETWEEN(SYSDATE, JDATE) / 12) ;
```

```
     IF YEARS >= 10 THEN
          DBMS_OUTPUT.PUT_LINE('The employee worked 10 years or more') ;
     ELSE
          DBMS_OUTPUT.PUT_LINE('The employee worked less than 10 years') ;
     END IF ;
EXCEPTION
     --handle exceptions one by one here
     WHEN NO_DATA_FOUND THEN
          DBMS_OUTPUT.PUT_LINE('Employee is not present in database.') ;
     WHEN OTHERS THEN
          DBMS_OUTPUT.PUT_LINE('I dont know what happened!') ;
END ;
/
```

The general syntax of handling exception codes is following:

```
DECLARE
     ...
BEGIN
     ...
EXCEPTION
     WHEN <EXCEPTION_NAME1> THEN
          Statement1 ;
     WHEN <EXCEPTION_NAME2> THEN
          Statement2 ;
     ...
     WHEN OTHERS THEN
          Default Statement ;
END ;
/
```

Note the following:

- <EXCEPTION_NAME1>, <EXCEPTION_NAME2> can be Oracle pre-defined exception names or user defined names.

- The default exception name OTHERS is optional.

- In our example above, we used two pre-defined exception names: NO_DATA_FOUND and OTHERS. There are many pre-defined exception names in Oracle. You must have the knowledge

of these exception names as Oracle throws such exceptions for several types of errors while processing a statement.

Oracle pre-defined exception names

The following table shows some Oracle pre-defined exception names.

Name of Exception	When Oracle Throws It?
NO_DATA_FOUND	When a SELECT INTO statement cannot retrieve a row.
TOO_MANY_ROWS	When a SELECT INTO statement retrieves more than one row.
DUP_VAL_ON_INDEX	When duplicate values are attempted to be stored in a table column with unique index.
NVALID_NUMBER	When the conversion of a character string into a number fails because the string does not represent a valid number.
VALUE_ERROR	When an arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	When an attempt is made to divide a number by zero.

Practice 12.2

   a. Extend the above block by handing the following exception: TOO_MANY_ROWS. Print an appropriate message when such an exception occurs.

   b. Write an example PL/SQL block that inserts a new arbitrary row to the COUNTRIES table. The block should handle the exception DUP_VAL_ON_INDEX and OTHERS. Run the block for different COUNTRY_ID and observe the cases when above exception occurs.

3.   **Loops in PL/SQL block**

There are several types of loop statements in PL/SQL such as FOR loops, WHITE loops, and unconditional loops. These are described below.

Use of loops in PL/SQL

The general structure of writing loops using FOR is given below.

```
FOR variable in lowest val..highest val
LOOP
     Statement1 ;
```

```
      Statement2 ;
      …
END LOOP ;
```

The following for loop prints the numbers from 1 to 100.

```
DECLARE
BEGIN
      FOR i in 1..100
      LOOP
            DBMS_OUTPUT.PUT_LINE(i);
      END LOOP ;
End ;
/
```

The general structure of WHILE loop is given below.

```
WHILE Condition
LOOP
      Statement1 ;
      Statement2 ;
      …
END LOOP ;
```

The following example prints the numbers from 1 to 100 using WHILE loop.

```
DECLARE
      i number;
BEGIN
      i := 1;
      WHILE i<=100
      LOOP
            DBMS_OUTPUT.PUT_LINE(i);
            i := i + 1;
      END LOOP ;
End ;
/
```

There is a type of loop called unconditional-loop in Oracle PL/SQL. The following example shows how to write the unconditional loop to print the numbers from 1 to 100.

```
DECLARE
      i number;
BEGIN
      --this is an unconditional loop, must have EXIT WHEN inside loop
      i := 1;
      LOOP
            DBMS_OUTPUT.PUT_LINE(i);
            i := i + 1;
            EXIT WHEN  (i > 100) ;
      END LOOP ;
End ;
/
```

Note the statement EXIT WHEN to exit the unconditional loop. This is a must in an unconditional loop otherwise the loop will run for-over. Also note that EXIT WHEN statement can also be used inside a WHILE loop or FOR loop to exit at any time.

Use of Cursor FOR loops in PL/SQL

The cursor FOR loop is special FOR loop in Oracle PL/SQL. The loop variable of this type of loop iterates over the rows of a SQL query. So, it is widely used when loop is to be done over the rows of a SELECT statement.

The general structure of writing a loop (a FOR loop) in PL/SQL is given below.

```
FOR variable in (SELECT statement)
LOOP
      Statement1 ;
      Statement2 ;
      …
END LOOP ;
```

The following example augments our previous code of finding employees working for more than 10 years. This time, the script counts the number of employees who worked 10 years or more in the company. Then, it displays the count.

```
DECLARE
```

```
      YEARS NUMBER ;
      COUNTER NUMBER ;
BEGIN
      COUNTER := 0 ;
      --the following for loop will iterate over all rows of the SELECT results
      FOR R IN (SELECT HIRE_DATE FROM EMPLOYEES )
      LOOP
            --variable R is used to retrieve columns
            YEARS := (MONTHS_BETWEEN(SYSDATE, R.HIRE_DATE) / 12) ;
            IF YEARS >= 10 THEN
                  COUNTER := COUNTER + 1 ;
            END IF ;
      END  LOOP ;
      DBMS_OUTPUT.PUT_LINE('Number of employees worked 10 years or more: ' ||
COUNTER) ;
END ;
/
```

In the above script, a special variable R is used which is called a *cursor-variable*. Details of cursor are out of scope of this book. In summary –

- A cursor variable fetches rows of a SELECT statement one be one

- The FOR LOOP ends automatically after the last row have been fetched by the cursor variable R

- The columns of a row are accessed by *variable_name.column_name* format (R.HIRE_DATE)

- Only those columns are available that are retrieved in the SELECT statement (R.LAST_NAME is invalid as LAST_NAME is not retrieved in SELECT)

Updating table from PL/SQL block

The following PL/SQL block increases salary of each employee X by 15% who have worked in the company for 10 years or more.

```
DECLARE
      YEARS NUMBER ;
      COUNTER NUMBER ;
      OLD_SAL NUMBER;
      NEW_SAL NUMBER;
BEGIN
```

```
      COUNTER := 0 ;
      FOR R IN (SELECT EMPLOYEE_ID, SALARY, HIRE_DATE FROM EMPLOYEES )
      LOOP
            OLD_SAL := R.SALARY ;
            YEARS := (MONTHS_BETWEEN(SYSDATE, R.HIRE_DATE) / 12) ;
            IF YEARS >= 10 THEN
                  UPDATE EMPLOYEES SET SALARY = SALARY * 1.15
                  WHERE EMPLOYEE_ID = R.EMPLOYEE_ID ;
            END IF ;
            SELECT SALARY INTO NEW_SAL FROM EMPLOYEES
            WHERE EMPLOYEE_ID = R.EMPLOYEE_ID ;
            DBMS_OUTPUT.PUT_LINE('Employee id:' || R.EMPLOYEE_ID || ' Salary: '
|| OLD_SAL || ' -> ' || NEW_SAL) ;
      END  LOOP ;
      COMMIT;
END ;
/
```

Practice 12.3

    a.   Write a PL/SQL block that will print 'Happy Anniversary X' for each employee X whose hiring date is today. Use cursor FOR loop for the task.

4.   PL/SQL Procedures

PL/SQL procedures and functions (to be discussed in later section) are like PL/SQL blocks except the following:

- Unlike PL/SQL (anonymous) blocks, a PL/SQL procedure or function has a name and zero or more parameters. The parameters are used to give inputs to the procedure or function. The function can even return values.

- A PL/SQL (anonymous) block is to be saved in a separate file by the user, if he wishes to run it later. However, a PL/SQL procedure or function can be saved in the database, so it need not be saved in a separate file by the user.

- When a procedure or function code is run in the SQL command line, the procedure or function codes are not executed. The codes are just stored in the database. To execute the procedure, we will

need to call it from the command line using EXEC PROC_NAME statement. To execute the function, we need to call it in a select statement or from inside a PL/SQL block.

Wring a procedure

The general syntax of writing a PL/SQL procedure is given below.

```
CREATE OR REPLACE PROCEDURE <PROC_NAME> ( <PARAM1> [IN/OUT] <DATATYPE1>, ...) IS
      Declaration1 ;
      Declaration2 ;
      ... ;
BEGIN
      Statement1 ;
      Statement2 ;
      ... ;
END ;
/
```

Note the following regarding procedure parameters:

- Parameters are optional so a procedure can have zero parameters.

- Parameters can be of two types: IN and OUT. IN parameters receive input and OUT parameters generate output (like returning values from procedure).

Let us re-write our PL/SQL block to find whether employee numbered 100 has worked10 years or more. Let us call this procedure IS_SENIOR_EMPLOYEE. The following code shows the procedure.

```
CREATE OR REPLACE PROCEDURE IS_SENIOR_EMPLOYEE IS
      JDATE DATE ;
      YEARS NUMBER ;
BEGIN
      SELECT HIRE_DATE INTO JDATE
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = 100 ;
      YEARS :=  (MONTHS_BETWEEN(SYSDATE, JDATE) / 12) ;
      IF YEARS >= 10 THEN
            DBMS_OUTPUT.PUT_LINE('The employee worked 10 years or more') ;
      ELSE
            DBMS_OUTPUT.PUT_LINE('The employee worked less than 10 years') ;
```

```
        END IF ;
END ;
/
```

If you run the above code, then the procedure is saved in the database. Later you can execute the procedure form the SQL*PLUS command line as follows.

```
EXEC  IS_SENIOR_EMPLOYEE ;
```

You can also execute the procedure inside a PL/SQL block, inside another procedure or function as shown below.

```
DECLARE
BEGIN
      IS_SENIOR_EMPLOYEE ;
END ;
```

Use of procedure parameters

Procedure parameters can be utilized to send inputs to the procedure. Let us re-write our previous procedure IS_SENIOR_EMPLOYEE so that the employee id is not kept fixed in the code rather we can send the employee id as a parameter to the procedure during run-time. The modified version of the procedure is given below.

```
CREATE OR REPLACE PROCEDURE IS_SENIOR_EMPLOYEE(EID IN VARCHAR2) IS
      JDATE DATE ;
      YEARS NUMBER ;
BEGIN
      SELECT HIRE_DATE INTO JDATE
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = EID ;
      YEARS :=  (MONTHS_BETWEEN(SYSDATE, JDATE) / 12) ;
      IF YEARS >= 10 THEN
            DBMS_OUTPUT.PUT_LINE('The employee worked 10 years or more') ;
      ELSE
            DBMS_OUTPUT.PUT_LINE('The employee worked less than 10 years') ;
      END IF ;
```

```
END ;
/
```

The greatest advantage of this modified parameterized version is that we can now run the same procedure for several employees as shown below.

```
DECLARE
BEGIN
      IS_SENIOR_EMPLOYEE(100) ;
      IS_SENIOR_EMPLOYEE(105) ;
END ;
```

Handling exception in procedure

We can handle exceptions in a PL/SQL procedure like we did in PL/SQL (anonymous) block. In all practical tasks, handling exception is a must.

Although, we did not handle any exception in the above blocks, a good programmer should handle all required exceptions and show meaning messages to the user. The following code illustrates this.

```
CREATE OR REPLACE PROCEDURE IS_SENIOR_EMPLOYEE(EID IN VARCHAR2) IS
      JDATE DATE ;
      YEARS NUMBER ;
BEGIN
      SELECT HIRE_DATE INTO JDATE
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = EID ;
      YEARS :=  (MONTHS_BETWEEN(SYSDATE, JDATE) / 12) ;
      IF YEARS >= 10 THEN
            DBMS_OUTPUT.PUT_LINE('The employee worked 10 years or more') ;
      ELSE
            DBMS_OUTPUT.PUT_LINE('The employee worked less than 10 years') ;
      END IF ;
EXCEPTION
       WHEN NO_DATA_FOUND THEN
             DBMS_OUTPUT.PUT_LINE('No employee found.') ;
       WHEN TOO_MANY_ROWS THEN
             DBMS_OUTPUT.PUT_LINE('More than one employee found.') ;
```

```
            WHEN OTHERS THEN
                    DBMS_OUTPUT.PUT_LINE('Some unknown error occurred.') ;
END ;
/
```

After handling exceptions as above, run the following block to understand the effect.

```
DECLARE
BEGIN
      IS_SENIOR_EMPLOYEE(10000) ;
      IS_SENIOR_EMPLOYEE(105) ;
END ;
```

Generate output from a procedure

Let us re-write above procedure so that it stores the resulting message in an output variable rather than printing the message itself. The following code shows how to do this.

```
CREATE OR REPLACE PROCEDURE IS_SENIOR_EMPLOYEE(EID IN VARCHAR2, MSG OUT VARCHAR2)
IS
      JDATE DATE ;
      YEARS NUMBER ;
BEGIN
      SELECT HIRE_DATE INTO JDATE
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = EID ;
      YEARS :=  (MONTHS_BETWEEN(SYSDATE, JDATE) / 12) ;
      IF YEARS >= 10 THEN
            MSG := 'The employee worked 10 years or more' ;
      ELSE
            MSG := 'The employee worked less than 10 years' ;
      END IF ;
EXCEPTION
      WHEN NO_DATA_FOUND THEN
            MSG := 'No employee found.' ;
      WHEN TOO_MANY_ROWS THEN
            MSG := 'More than one employee found.' ;
      WHEN OTHERS THEN
            MSG := 'Some unknown error occurred.' ;
```

```
END ;
/
```

The following code now tests the re-written procedure. It uses a new variable MESSAGE to retrieve the output of the procedure. Then it prints the MESSAGE.

```
DECLARE
      MESSAGE VARCHAR2(100) ;
BEGIN
      IS_SENIOR_EMPLOYEE(10000, MESSAGE) ;
      DBMS_OUTPUT.PUT_LINE(MESSAGE) ;
      IS_SENIOR_EMPLOYEE(105, MESSAGE) ;
      DBMS_OUTPUT.PUT_LINE(MESSAGE) ;
END ;
```

5.   **PL/SQL Functions**

A PL/SQL function is like a PL/SQL procedure except that it must return a value. The general syntax of a PL/SQL function is slightly different than a PL/SQL procedure. It only has an RETURN <DATATYPE> clause at the end of the CREATE OR REPLACE statement.

```
CREATE OR REPLACE FUNCTION <FUNC_NAME> ( <PARAM1> [IN/OUT] <DATATYPE1>, ...)
RETURN <DATATYPE> IS
      Declaration1 ;
      Declaration2 ;
      ... ;
BEGIN
      Statement1 ;
      Statement2 ;
      ... ;
END ;
/
```

The effect of the RETURN statement is that a value is return after successful execution of the function. That value will be used in the calling section.

Let us re-write the same procedure used before as a function. In this task, the MSG variable used in the procedure to output the message will be done by return statement. The following code shows the function how to do this.

```
CREATE OR REPLACE FUNCTION GET_SENIOR_EMPLOYEE(EID IN VARCHAR2)
RETURN VARCHAR2 IS
      JDATE DATE ;
      YEARS NUMBER ;
      MSG VARCHAR2(100) ;
BEGIN
      SELECT HIRE_DATE INTO JDATE
      FROM EMPLOYEES
      WHERE EMPLOYEE_ID = EID ;
      YEARS :=  (MONTHS_BETWEEN(SYSDATE, JDATE) / 12) ;
      IF YEARS >= 10 THEN
          MSG := 'The employee worked 10 years or more' ;
      ELSE
          MSG := 'The employee worked less than 10 years' ;
      END IF ;
      RETURN MSG ; --return the message
EXCEPTION
      --you must return value from this section also
      WHEN NO_DATA_FOUND THEN
          RETURN 'No employee found.' ;
      WHEN TOO_MANY_ROWS THEN
          RETURN 'More than one employee found.' ;
      WHEN OTHERS THEN
          RETURN 'Some unknown error occurred.' ;
END ;
/
```

 Note the following:

- A function must return a value of desired value from the BEGIN section

- A function must also return a value of desired type from the EXCEPTION section.

The following code now tests our function.

```
DECLARE
      MESSAGE VARCHAR2(100) ;
```

```
BEGIN
      MESSAGE := GET_SENIOR_EMPLOYEE(10000) ;
      DBMS_OUTPUT.PUT_LINE(MESSAGE) ;
      MESSAGE := GET_SENIOR_EMPLOYEE(105) ;
      DBMS_OUTPUT.PUT_LINE(MESSAGE) ;
END ;
```

Why function when procedure can output value?

We have seen that a procedure can output values. So why should one write a function? The most important application of writing a function is that it can be used in a SELECT query unlike a procedure which can't be used. The beautiful effect of a function can then be best understood like other SQL functions we have seen and used before in various queries.

Let us write the following SQL query and see the output.

```
SELECT LAST_NAME, GET_SENIOR_EMPLOYEE(EMPLOYEE_ID)
FROM EMPLOYEES ;
```

Nested PL/SQL blocks

PL/SQL blocks can be nested. This means we can write a PL/SQL block inside a PL/SQL block. We will go into too much details of nesting. We will just illustrate the nesting by re-writing the function we created above.

Let us re-write the above function as follows. In this case, before retrieving the HIRE_DATE of the employee, we will first check whether such an employee occurs in database. After it is found that such an employee occurs, we will retrieve its HIRE_DATE and continue later processing. The modified function is shown below.

```
CREATE OR REPLACE FUNCTION GET_SENIOR_EMPLOYEE(EID IN VARCHAR2)
RETURN VARCHAR2 IS
      ECOUNT NUMBER;
      JDATE DATE ;
      YEARS NUMBER ;
      MSG VARCHAR2(100) ;
BEGIN
      --Inner PL/SQL block
```

```
        BEGIN
                SELECT COUNT(*) INTO ECOUNT
                FROM EMPLOYEES
                WHERE EMPLOYEE_ID = EID ;
        END ;
        IF ECOUNT = 0 THEN
                MSG := 'No employee found.' ;
        ELSIF ECOUNT > 1 THEN
                MSG := 'More than one employee found.' ;
        ELSE
                SELECT HIRE_DATE INTO JDATE
                FROM EMPLOYEES
                WHERE EMPLOYEE_ID = EID ;
                YEARS :=  (MONTHS_BETWEEN(SYSDATE, JDATE) / 12) ;
                IF YEARS >= 10 THEN
                        MSG := 'The employee worked 10 years or more' ;
                ELSE
                        MSG := 'The employee worked less than 10 years' ;
                END IF ;
        END IF ;
        RETURN MSG ;
END ;
/
```

Note the following:

- Inner PL/SQL blocks are like a general block that can have EXCEPTION section as well if required.

- In the modified function, exception handling is not required. Because this time number of employees is first checked before retrieving employee's hiring date. If no such data is found, then the first SELECT COUNT(*) INTO ECOUNT query retrieves 0 into ECOUNT variable. So, no exception occurs and therefore exception handling is also not required.

6.   **Pl/SQL Triggers**

A trigger is a PL/SQL stored block. It is like a function or a procedure. However, it is different than a function or a procedure. To run a function or a procedure, it is explicitly called from a code. However, a trigger is automatically run by the Oracle. It is not called from any code directly.

A trigger is automatically run by Oracle -

- After/Before a database insertion operation

- After/Before a database deletion operation

- After/Before a database update operation

- After DDL operations, etc.

Trigger is very useful when certain tasks are needed to be done after or before a DML operation. For example, suppose we need to ensure that after every deletion of an employee records from the EMPLOYEES table, the records of the deleted employee should go to a backup table. In such cases, a trigger can be written which will perform the required task easily. Since, trigger will be automatically called after the deletion operation; we need not to manually handle the writing backup records.

The general syntax of a trigger is like a procedure or a function except the header line.

```
CREATE OR REPLACE TRIGGER <TRIGGER_NAME>
(BEFORE | AFTER) (INSERT | UPDATE | DELETE)
[OF <COLUMN_NAME>]
ON <TABLE_NAME>
[FOR EACH ROW]
[WHEN <CONDITION>]
DECLARE
      Declaration1 ;
      Declaration2 ;
      ... ;
BEGIN
      Statement1 ;
      Statement2 ;
      ... ;
EXCEPTION
      Exception handing codes ;
END ;
/
```

Let's first create the following table STUDENTS.

```
CREATE TABLE STUDENTS(
STUDENT_NAME VARCHAR2(250),
CGPA NUMBER
) ;
```

Let's understand the trigger by writing our first trigger HELLO_WORLD trigger. This trigger will print "Hello World" whenever an insertion statement is run on the table STUDENTS.

The following shows the trigger codes.

```
CREATE OR REPLACE TRIGGER HELLO_WORLD
AFTER INSERT
ON STUDENTS
DECLARE
BEGIN
      DBMS_OUTPUT.PUT_LINE('Hello World');
END ;
/
```

Now, insert some rows (as shown below) and see what output comes!

```
INSERT INTO STUDENTS VALUES ('Fahim Hasan', 3.71);
INSERT INTO STUDENTS VALUES ('Ahmed Nahiyan', 3.80);
```

The trigger will run twice for the two insert statements and it will ouput "Hello World".

Although, HELLO_WORLD trigger does not show the actual power of triggers, it demonstrates at least the basic properties of a trigger which are following:

- HELLO_WORLD trigger will run automatically by Oracle when an insert statement is run on the table STUDENTS.

- It will run after the execution of insert statement.

Let's try several modification of the above trigger so that it runs in various settings. The following codes show five variations of the HELLO_WORLD trigger.

```
--This trigger will run before insert on STUDENTS table

CREATE OR REPLACE TRIGGER HELLO_WORLD2
```

```
BEFORE INSERT
ON STUDENTS
DECLARE
BEGIN
      DBMS_OUTPUT.PUT_LINE('Hello World2');
END ;
/


--This trigger will run after an insert or a delete statement on STUDENTS table

CREATE OR REPLACE TRIGGER HELLO_WORLD3
AFTER INSERT OR DELETE
ON STUDENTS
DECLARE
BEGIN
      DBMS_OUTPUT.PUT_LINE('Hello World3');
END ;
/


--The following trigger will run after an update statement on STUDENTS table.
--Added with that, this trigger will only run when update is performed on the
--CGPA column.

CREATE OR REPLACE TRIGGER HELLO_WORLD4
AFTER UPDATE
OF CGPA
ON STUDENTS
DECLARE
BEGIN
      DBMS_OUTPUT.PUT_LINE('Hello World4');
END ;
/


--The following trigger will run after an update operation on the STUDENTS table.
--Added with that, the trigger will run once for each row. The previous rows will
--run once for the whole statement, where this trigger will run N times if N rows
--are affected by the SQL statement.

CREATE OR REPLACE TRIGGER HELLO_WORLD5
AFTER UPDATE
OF CGPA
```

```
ON STUDENTS
FOR EACH ROW
DECLARE
BEGIN
      DBMS_OUTPUT.PUT_LINE('Hello World5');
END ;
/
```

Read the above five trigger codes and understand them. After you properly understand all five triggers, let's now determine which trigger will be called by Oracle for the following five SQL statements.

```
INSERT INTO STUDENTS VALUES ('Shakib Ahmed', 3.63);
--This will run HELLO_WORLD, HELLO_WORLD2, HELLO_WORLD3


DELETE FROM STUDENTS WHERE CGPA < 3.65 ;
--This will run HELLO_WOLRD3


UPDATE STUDENTS SET CGPA = CGPA + 0.01 WHERE STUDENT_NAME LIKE '%Shakib%';
--This will run HELLO_WORLD4, but will not run HELLO_WORLD5!!! Why? Because
--HELLO_WORLD5 is declared with FOR EACH ROW clause. This means trigger should be
--run for each row affected. Since, the above statement does not update any row
--(as the previous DELETE operation already deleted that row from the table)
--it will not run HELLO_WORLD5!


UPDATE STUDENTS SET STUDENT_NAME = 'Fahim Ahmed'
WHERE STUDENT_NAME = 'Fahim Hasan' ;
--This will not run any trigger. Although HELLO_TRIGGER4 is declared to be run
--after an update operation, the trigger will not run because the update is done
--on the column STUDENT_NAME rather than CGPA


UPDATE STUDENTS SET CGPA = CGPA + 0.01 ;
--This will run both HELLO_WORLD4 and HELLO_WORLD5 trigger. However, HELLO_WORLD5
--will run twice! Why? Because two rows will be affected by the SQL statement!
```

After analyzing the above five triggers, you should now understand the following:

- A trigger can run be before or after a DML operation.

- The DML operation can be insert, update, delete or a combination of the three.

- The trigger can be created such that it is run only once after an SQL statement or it is run once per row affected by an SQL statement.

Classification of triggers

Trigger can be classified in following ways:

- *BEFORE trigger vs. AFTER trigger* – A BEFORE trigger is executed before a statement and an AFTER trigger is executed after the statement.

- *ROW LEVEL trigger vs. STATEMENT LEVEL trigger* – A ROW LEVEL trigger is created when FOR EACH ROW clause is specified in the trigger definition. In this case, the trigger will be called after or before each row that is affected by the operation. A STATEMENT LEVEL trigger is executed only once for each statement. So, if you issue a DML statement that affect N rows, then a STATEMENT LEVEL trigger will be executed once while a ROW LEVEL trigger will be executed N times. A STATEMENT LEVEL trigger must get executed once while a ROW LEVEL trigger may not get executed at all if the DML operation does not affect any row!!!

Problem Example 1

Write a PL/SQL trigger LOG_CGPA_UPDATE. This trigger will log all updates done on the CGPA column of STUDENTS table. The trigger will save current user's name, current system date and time in a log table named LOG_TABLE_CGPA_UPDATE.

Solution

The trigger required will be a STATEMENT LEVEL trigger. Because, it is required to run only per SQL statement which will serve our purpose.

Let's first create the LOG_TABLE_CGPA_UPDATE table as follows.

```
CREATE TABLE LOG_TABLE_CGPA_UPDATE
(
USERNAME VARCHAR2(25),
DATETIME DATE
) ;
```

The trigger code is shown below.

```
CREATE OR REPLACE TRIGGER LOG_CGPA_UPDATE
AFTER UPDATE
OF CGPA
ON STUDENTS
DECLARE
      USERNAME VARCHAR2(25);
BEGIN
      USERNAME := USER; --USER is a function that returns current username
      INSERT INTO LOG_TABLE_CGPA_UPDATE VALUES (USERNAME, SYSDATE);
END ;
/
```

After the trigger is created, issue the following update commands and then finally view the rows inserted into LOG_TABLE_CGPA_UPDATE table.

```
--First update
UPDATE STUDENTS SET CGPA = CGPA + 0.01 ;


--Another update
UPDATE STUDENTS SET CGPA = CGPA - 0.01 ;


--View the rows inserted by the trigger
SELECT * FROM LOG_TABLE_CGPA_UPDATE
```

Problem Example 2

Write a PL/SQL trigger BACKUP_DELETED_STUDENTS. This trigger will save all records that are deleted from the STUDENTS table into a backup table named STUDENTS_DELETED. The trigger will save student's record along with current user's name and current system date and time.

<u>**Solution**</u>

The trigger required will be a ROW LEVEL trigger. Because, it is required to run per row affected. Each row that will be deleted will need to be saved in the backup table.

Let's first create the backup table STUDENTS_DELETED.

```
CREATE TABLE STUDENTS_DELETED(
STUDENT_NAME VARCHAR2(25),
CGPA NUMBER,
USERNAME VARCHAR2(25),
DATETIME DATE
) ;
```

The following code shows the trigger definition required to perform the specified task.

```
CREATE OR REPLACE TRIGGER BACKUP_DELETED_STUDENTS
BEFORE DELETE
ON STUDENTS
FOR EACH ROW
DECLARE
      V_NAME VARCHAR2(25);
      V_USERNAME VARCHAR2(25);
      V_CGPA NUMBER;
      V_DATETIME DATE;
BEGIN
      V_NAME := :OLD.STUDENT_NAME ;
      V_CGPA := :OLD.CGPA ;
      V_USERNAME := USER ;
      V_DATETIME := SYSDATE ;
      INSERT INTO STUDENTS_DELETED VALUES (V_NAME,V_CGPA,V_USERNAME,V_DATETIME);
END ;
/
```

Note the use of :OLD special reference. This reference is used to access the column values of currently affected row by the DELETE operation. Like the :OLD reference, there is a :NEW reference that can be used to retrieve the column values of the new row that will result after completion of the operation.

After the trigger is compiled and stored successfully, issue the following command and view the rows inserted by the trigger.

```
--Delete the two rows
DELETE FROM STUDENTS WHERE CGPA < 3.85 ;


--View the rows inserted by the trigger
```

```
SELECT * FROM STUDENTS_DELETED ;
```

References :OLD vs. :NEW for a ROW LEVEL trigger

Note the following regarding :OLD and :NEW references-

- They are valid only for ROW LEVEL triggers! You are not allowed to use these in a STATEMET LEVEL trigger.

- For an update statement, :OLD is used to retrieve old values of columns while :NEW is used to retrieve new values of columns.

- For an insert statement, :OLD retrieves NULL values for all columns while :NEW can be used to retrieve column values of the row

- For a delete statement :NEW retrieves NULL values while :OLD can be used to retrieve column values of the row

- :NEW reference can be used to change column values of a row before it is going to be inserted in the table. *This will require a BEFORE trigger.*

To understand the :OLD and :NEW reference more deeply, write the following trigger and issue the SQL commands listed after the trigger.

```
CREATE OR REPLACE TRIGGER OLD_NEW_TEST
BEFORE INSERT OR UPDATE OR DELETE
ON STUDENTS
FOR EACH ROW
DECLARE
BEGIN
      DBMS_OUTPUT.PUT_LINE(':OLD.CGPA = ' || :OLD.CGPA) ;
      DBMS_OUTPUT.PUT_LINE(':NEW.CGPA = ' || :NEW.CGPA) ;
END ;
/

--Issue the following SQL statements and view the dbms outputs
INSERT INTO STUDENTS VALUES ('SOUMIK SARKAR', 3.85);


UPDATE STUDENTS SET CGPA = CGPA + 0.02 ;


DELETE FROM STUDENTS WHERE CGPA < 3.90;

```

Problem Example 3

Write a PL/SQL trigger CORRECT_STUDENT_NAME. This trigger will be used to correct the text case of the student names when it is going to be inserted. So, this will be a BEFORE INSERT trigger. The trigger will change the case of the student's name to INITCAP format if it was not given by the user in the INSERT statement. The trigger will thus ensure that all names stored in the STUDENTS table will be in a consistent format.

```
CREATE OR REPLACE TRIGGER CORRECT_STUDENT_NAME
BEFORE INSERT
ON STUDENTS
FOR EACH ROW
DECLARE
BEGIN
      :NEW.STUDENT_NAME := INITCAP(:NEW.STUDENT_NAME) ;
END ;
/


--Issue the following SQL statements and then view the rows of STUDENTS table
INSERT INTO STUDENTS VALUES ('SHAkil ahMED', 3.80);


INSERT INTO STUDENTS VALUES ('masum billah', 3.60);
```

Note that the above trigger must be a BEFORE INSERT trigger. If you write an AFTER INSERT trigger, then it will not work. *Because in an AFTER INSERT trigger, you are not allowed to change values of the :NEW row.*

Drop a trigger from database

```
DROP TRIGGER <TRIGGER_NAME> ;
```

So, to drop the trigger OLD_NEW_TEST from the database, issue the following command.

```
DROP TRIGGER OLD_NEW_TEST ;
```

Practice 12.6

a. In Oracle, there is a function TO_NUMBER that converts a VARCHAR2 value to a numeric value. If the input to this function is not a valid number, then this function throws an exception. This is a problem in a SQL query because the whole query would not produce any result if one row generates an exception. So, your job is to write a PL/SQL function ISNUMBER that receives an input VARCHAR2 value and checks whether the input can be converted to a valid number. If the input can be converted to a valid number than ISNUMBER should return 'YES', otherwise ISNUMBER should return 'NO'.

b. Write a trigger HELLO_WORLD6 that will run after a deletion operation on the STUDENTS table. The trigger should be a ROW LEVEL trigger.

c. Write down a PL/SQL trigger on STUDENTS table. The trigger will ensure that whenever a new row is inserted in the STUDENTS table, the name of the student contains only alphabetic characters. Name your trigger INVALID_NAME. If the name is valid, then insertion should be allowed. However, if the name is invalid, then insertion should be denied. To deny insertion, you can throw an exception from the trigger that would halt the insertion operation.

d. Write a trigger that will save a student records in a table named LOW_CGPA_STUDENTS which contain only one column to store student's names. The trigger will work before an update operation or an insert operation. Whenever the update operation results in a CGPA value less than 2.0, the trigger will be fired and the trigger will save the students name in the LOW_CGPA_STUDENTS table. Similarly, when an insert operation inserts a new row with CGPA less than 2.0, the corresponding row must be saved in the LOW_CGPA_STUDENTS table.