Simplified

e- Book

SARVA EDUCATION

ORACLE

SARVA EDUCATION ^{SM} *- An I.T & Skill Advancement Training Programme, Initiated by* ***SITED***$^{®}$***-India***

An ISO 9001:2015 Certified Organization

ORACLE

CHAPTER – 1

BASIC OF ORACLE

INTRODUCTION

Oracle is an **RDBMS**. RDBMS stands for **Relational Database Management System.** An **RDBMS** is a computer program, which provides the user the facility to store and retrieve data, in a manner consistent with a defined model, called the Relational model.

SQL (Structured Query Language) is a computer language aimed to store, manipulate, and query data stored in relational databases. The first incarnation of SQL appeared in 1974, when a group in IBM developed the first prototype of a relational database. The first commercial relational database was released by Relational Software (later becoming Oracle).

Standards for SQL exist. However, the SQL that can be used on each one of the major **RDBMS** today is in different flavors. This is due to two reasons: 1) the SQL command standard is fairly complex, and it is not practical to implement the entire standard, and 2) each database vendor needs a way to differentiate its product from others.

SQL * Plus

It is an extension to the standard SQL and it has an on-fine command interpreter. The users can create program files and generate formatted reports. It is an interactive tool, an Oracle client provided by the company, Oracle Corporation.

OBJECTIVES

At the end of this unit, you will be able to,

* Understand RDBMS.
* CREATE a table and INSERT data into the table.
* Issue UPDATE, DELETE and SELECT commands.
* Enforce Referential Integrity constraint.
* Retrieve data from two or more tables by joining the tables with a WHERE clause.
* Use Number, Aggregate, Character and Conversion functions.

INTRODUCTION TO RDBMS

RDBMS stands for Relational Database Management System. An RDBM is a computer program, which provides the user the facility to store and retrieve data in a manner consistent with a defined model, called the Relational model.

The Relational model has three components**:**

1. Structural component to build the model -

* Relation (table) *****Attributes (columns) *****Tuples (rows)

2. Integrity component, which defines the operational rules of the data, model.

3. Manipulation component, which is the component to manipulate the structure.

Data Structure

The fundamental rule of the Relational model is that data is seen as tables. A table is formally referred to as a relation. Every relation has the same structure - i.e. the same number of columns and rows. Each column which represents a attribute of the relation must be unique.

Oracle, being an RDBMS, stores the data in tables. Many tables can be created and related to each other. All the tables form the Database.

E.g. Table with 4 columns.

ROLL NO	NAME	COURSE	JOIN DATE
1	Sachin	CCC01	01-AUG-1990
2	Kapil Dev	CCC02	20-JUL-1988
3	Azharuddi	CCC01	30-JUN-1988
4	Sunil	CCC02	1G-JAN-1970

Data Integrity

The Relational model also provides data integrity, which means that the data is accepted based on certain rules and therefore, data is valid. **Example**. Oracle will ensure that roll numbers of the students in the file is unique. Data integrity is maintained using a set of rules referred to as integrity constraints.

Data Manipulation

In order to create tables and manipulate the data within tables, Oracle provides the database access language called SQL. SQL stands for Structured Query Language.

Structured Query Language is a English-like language which is used to store and retrieve data from a database. So, it is a database language to help users to extract information from a database easily. This language is nonprocedural. SQL was introduced by IBM. SQL was standardized by ANSI (American National Standards Institute) and ISO (International Standards Organization) in 1986.

Oracle supports distributed databases which means that many physically separate databases can appear to the user as a single database. The physical distance becomes transparent to the user. This is also referred to as Client/Server architecture. Clients are nodes connected to the database server. Oracle also ha: E-Mail interface which allows communication between users and applications.

Oracle is the current version of Oracle. The first version was introduced in the year 1982 and was called Oracle 2. To develop applications, Oracle offers the following tools and features.

TOOLS

- **SQL* Plus**
- **SQL * Report Writer**
- **SQL * Forms**
- **Export/lmport/Utiitty**

SQL* Plus

SQL*Plus is Oracle's implementation of SQL. It is an extension to the standard SQL. SQL*Plus comprises of PL/SQL which is procedural. Procedural means that PL/SQL has all the programming structures of a procedural language like selection, Iteration and decision making.

SQL *Plus comprises of SQL, SQL*PLUS and PL/SQL

SQL * Forms

SQL * Forms is a tool for creating forms based applications. It has provision to create default data entry forms. SQL * Forms can be used to query a database, Update, delete or add to a database.

SQL * Report Writer:

SOL *RoportWriter is a menu-driven formatting tool to create simple to sophisticated reports.

Export / Import utility:

Files may be imported into a database or exported using this utility.

FEATURES:-

. Declarative	. Triggers	. Alerts
. Database Integrity	. Procedures	. Security

Declarative Database Integrity:

Declarative Database Integrity guarantees that the data in the database is within a predefined set of rules. The data can be maintained reliably without any programming. Database integrity is achieved through integrity constraints.

Integrity constraints are rules (hot are applicable on the columns of the table, which prevents invalid data entry into the tables.
Integrity constraint can be enforced through Entity integrity arid Referential integrity.
Entity Integrity ensures that the data may be, UNIQUE or take a DEFAULT value or be limited to a CHECK.
Referential Integrity is enforcing a parent/child relationship between tables such that child records cannot be appended without a parent record and parent records cannot be deleted when child records exist.

Triggers:

Triggers are blocks of PL/SQL code written by users for a specific table. This code is triggered (or fired or executed) when any INSERT, DELETE or UPDATE operation is performed on that table.
Triggers help in performing complex security checking and in performing related operations for an event (INSERT, UPDATE or DELETE).

Procedures:

Procedures are SQL and PL/SQL code compiled and stored in the database.
The procedures can be called into execution. Useful for batch processing like weekly backup etc.

Alerts:

Alerts are informative messages that are application specific and generated by certain transactions. Example, inform users of some critical change to a database.

Security:

Oracle performs record (row) locking or table locking when two users are updating the same data. The row locking is an automatic feature.
The use of the database is secured by privileges that are granted to the user by the Database Administrator (DBA;
Each user is given a login name and certain privileges by the DBA. The login name is an account of the user or the scheme. Without a login name the user cannot access the database.

Operating System Support:

Tie Oracle Server supports many operating systems like UNIX, VMS, OS/2, MS-DCS (Windows 3.1, Windows 95, and Widows NT).

SQL*PLUS

SQL *Plus is an extension to the standard SQL and it has an on-line command interpreter. The users can create program files and generate formatted reports, it is an interactive tool, an Oracle client provided by the company, Oracle Corporation.

SQL *Plus is used by application developers to,

Create a database

- o Modify the database
- o Generate reports

SQL *Plus is used by end-users to,

- Query and retrieve data from the database.

SQL *Pius is used by database Administrators to.

- Create users
- Specify rights and privileges to users.
- Monitor the database.

To Load SQL

- Select SQL* PLUS from Oracle document window.
- Supply the username an: the password. If both are correct, the SQL prompt appears.
- SQL>

This is called the command line.

To Exit from SQL

Type exit at the SQL command line.

SQL *Plus Commands

SQL commands fall into 3 categories -

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- DCL (Data Control Language)

Data Definition Language – DDL

DDL is used to create and remove database objects.

Example: CREATE, ALTER, DROP

Data Manipulation Language -DML

DML is used to manipulate the data in the database.

Example: INSERT, DELETE, UPDATE, SELECT

Data Control Language • DCL

DCL is used to control the kind of data access to the database, and also for Transaction control.

Example: GRANT and REVOKE are used for Data access control. COMMIT and ROLLBACK are Transaction control commands

Example: GRANT and REVOKE are used for Data access control. COMMIT and ROLLBACK are Transaction control commands

- SQL commands are terminated by a semicolon (;)
- SQL commands are stored In the SQL buffer (the last SQL buffer the last command is stored).
- SQL commands are executed by pressing /.

Some SQL Buffer Commands

In SQL* Plus, whenever we type a command, that command is stored in the memory. This memory is referred to the buffer. When we type the second command, the first command is lost.

SQL> LIST or L (To display the command in the buffer)

- If the command is a single fine command, that line itself Is the current fine.
- In a mufti-fine command the last fine is the current line.
- The current fine is indicated by a * .

To make any line the current line.
Type the fine number and press enter. SQL> 2
Now, the second fine of the command becomes the current line.
To add more lines to the existing command. SQL> INPUT or I
To delete the current line from the buffer SQL> DEL
To append to the current line SQL> APPEND command or A command
Example, SQL> a)
will add a closing bracket at the end of the current fine.

To make changes to the current line

SQL>CHANGE/old-value/new-value or C/oId-value/new-value

Suppose the command has been typed as,

SELECT EMPN,EMP_NAME FROM EMP;

SQL>C/EMPN/EMP_NO (NOT case sensitive)

will change the first occurrence of EMPN to EMP_NO

To execute the current command in the SQL buffer

Type the / and press enter.

SQL>/

To Save the command in a file for future use: SQL>SAVE filename

Example, SQL>SAVE cust

Will write the command to a file with extension SQL-cust.sql.
To OVERWRITE the command file with extension SQL :SQL>SAVE filename REPLACE.
To READ the commands in the SQL file into the buffer :SQL>GET, filename
To load and execute the specified file of SQL*pIus commands SQL>START filename

CHAPTER – 2

SQL (ORACLE) STATEMENTS

INTRODUCTION

The most commonly used SQL command is SELECT statement. The SQL SELECT statement is used to query or retrieve data from a table in the database. A query may retrieve information from specified columns or from all of the columns in the table. To create a simple SQL SELECT Statement, you must specify the column(s) name and the table name. The whole query is called SQL SELECT Statement.

Syntax of SQL SELECT Statement:

```
SELECT column_list FROM table-name
[WHERE Clause]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause];
```

- *table-name* is the name of the table from which the information is retrieved.
- *column_list* includes one or more columns from which data is retrieved.
- The code within the brackets is optional.

Database table student_details;

id	first_name	last_name	age	subject	games
100	Rahul	Sharma	10	Science	Cricket
101	Anjali	Bhagwat	12	Maths	Football
102	Stephen	Fleming	09	Science	Cricket
103	Shekar	Gowda	18	Maths	Badminton
104	Priya	Chandra	15	Economics	Chess

NOTE: These database tables are used here for better explanation of SQL commands. In reality, the tables can have different columns and different data.

For example, consider the table student_details. To select the first name of all the students the query would be like:

```
SELECT first_name FROM student_details;
```

NOTE: The commands are not case sensitive. The above SELECT statement can also be written as "select first_name from students_details;"

You can also retrieve data from more than one column. **For example,** to select first name and last name of all the students.

```
SELECT first_name, last_name FROM student_details;
```

You can also use clauses like WHERE, GROUP BY, HAVING, ORDER BY with SELECT statement. We will discuss these commands in coming chapters.

NOTE: In a SQL SELECT statement only SELECT and FROM statements are mandatory. Other clauses like WHERE, ORDER BY, GROUP BY, HAVING are optional.

How to use expressions in SQL SELECT Statement?

Expressions combine many arithmetic operators, they can be used in SELECT, WHERE and ORDER BY Clauses of the SQL SELECT Statement.

Here we will explain how to use expressions in the SQL SELECT Statement.

About using expressions in WHERE and ORDER BY clause, they will be explained in their respective sections.

The operators are evaluated in a specific order of precedence, when more than one arithmetic operator is used in an expression.

The order of evaluation is: parentheses, division, multiplication, addition, and subtraction. The evaluation is performed from the left to the right of the expression.

For example: If we want to display the first and last name of an employee combined together, the SQL Select Statement would be like

```
SELECT first_name + ' ' + last_name FROM employee;
```

Output:

```
first_name + ' ' + last_name
--------------------------------
      Rahul Sharma
      Anjali Bhagwat
      Stephen Fleming
      Shekar Gowda
      Priya Chandra
```

You can also provide aliases as below.

```
SELECT first_name + ' ' + last_name AS emp_name
FROM employee;
```

Output:

```
emp_name
-------------
Rahul Sharma
Anjali Bhagwat
Stephen Fleming
Shekar Gowda
Priya Chandra
```

SQL INSERT Statement

The INSERT Statement is used to add new rows of data to a table.

We can insert data to a table in two ways,

1) Inserting the data directly to a table.

Syntax for SQL INSERT is:

```
INSERT INTO TABLE_NAME
[ (col1, col2, col3,...colN)]
VALUES (value1, value2, value3,...valueN);
```

col1, col2,...colN -- the names of the columns in the table into which you want to insert data.

While inserting a row, if you are adding value for all the columns of the table you need not specify the column(s) name in the sql query. But you need to make sure the order of the values is in the same order as the columns in the table. The sql insert query will be as follows

```
INSERT INTO TABLE_NAME
VALUES (value1, value2, value3,...valueN);
```

For Example: If you want to insert a row to the employee table, the query would be like,

```
INSERT INTO employee (id, name, dept, age,
salary location) VALUES (105, 'Srinath',
'Aeronautics', 27, 33000);
```

NOTE: When adding a row, only the characters or date values should be enclosed with single quotes.

If you are inserting data to all the columns, the column names can be omitted. The above insert statement can also be written as,

```
INSERT INTO employee
VALUES (105, 'Srinath', 'Aeronautics', 27,
33000);
```

Inserting data to a table through a select statement.

Syntax for SQL INSERT is:

```
INSERT INTO table_name
[(column1, column2, ... columnN)]
SELECT column1, column2, ...columnN
FROM table_name [WHERE condition];
```

For Example: To insert a row into the employee table from a temporary table, the sql insert query would be like,

```
INSERT INTO employee (id, name, dept, age, salary
location) SELECT emp_id, emp_name, dept, age,
salary, location
FROM temp_employee;
```

If you are inserting data to all the columns, the above insert statement can also be written as,

```
INSERT INTO employee
SELECT * FROM temp_employee;
```

NOTE: We have assumed the temp_employee table has columns emp_id, emp_name, dept, age, salary, location in the above given order and the same datatype.

IMPORTANT NOTE:

1) When adding a new row, you should ensure the datatype of the value and the column matches

2) You follow the integrity constraints, if any, defined for the table.

SQL UPDATE Statement

The UPDATE Statement is used to modify the existing rows in a table.

The Syntax for SQL UPDATE Command is:

```
UPDATE table_name
SET column_name1 = value1,
column_name2 = value2, ...
[WHERE condition]
```

- table_name - the table name which has to be updated.
- column_name1, column_name2.. - the columns that gets changed.
- value1, value2... - are the new values.

NOTE: In the Update statement, WHERE clause identifies the rows that get affected. If you do not include the WHERE clause, column values for all the rows get affected.

For Example: To update the location of an employee, the sql update query would be like,

```
UPDATE employee
SET location ='Mysore'
WHERE id = 101;
```

To change the salaries of all the employees, the query would be,

```
UPDATE employee
SET salary = salary + (salary * 0.2);
```

SQL DELETE Statement

The DELETE Statement is used to delete rows from a table.

The Syntax of a SQL DELETE statement is:

```
DELETE FROM table_name [WHERE condition];
```

table_name -- the table name which has to be updated.

NOTE: The WHERE clause in the sql delete command is optional and it identifies the rows in the column that gets deleted. If you do not include the WHERE clause all the rows in the table is deleted, so be careful while writing a DELETE query without WHERE clause.
For Example: To delete an employee with id 100 from the employee table, the sql delete query would be like,

```
DELETE FROM employee WHERE id = 100;
```

To delete all the rows from the employee table, the query would be like,

```
DELETE FROM employee;
```

SQL TRUNCATE Statement

The SQL TRUNCATE command is used to delete all the rows from the table and free the space containing the table.

Syntax to TRUNCATE a table:

```
TRUNCATE TABLE table_name;
```

For Example: To delete all the rows from employee table, the query would be like,

```
TRUNCATE TABLE employee;
```

Difference between DELETE and TRUNCATE Statements:

DELETE Statement: This command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.
TRUNCATE statement: This command is used to delete all the rows from the table and free the space containing the table.

SQL DROP Statement:

The SQL DROP command is used to remove an object from the database. If you drop a table, all the rows in the table is deleted and the table structure is removed from the database. Once a table is dropped we cannot get it back, so be careful while using DROP command. When a table is dropped all the references to the table will not be valid.

Syntax to drop a sql table structure:

```
DROP TABLE table_name;
```

For Example: To drop the table employee, the query would be like

```
DROP TABLE employee;
```

Difference between DROP and TRUNCATE Statement:

If a table is dropped, all the relationships with other tables will no longer be valid, the integrity constraints will be dropped, grant or access privileges on the table will also be dropped, if want use the table again it has to be recreated with the integrity constraints, access privileges and the relationships with other tables should be established again. But, if a table is truncated, the table structure remains the same, therefore any of the above problems will not exist.

SQL CREATE TABLE Statement

The CREATE TABLE Statement is used to create tables to store data. Integrity Constraints like primary key, unique key, foreign key can be defined for the columns while creating the table. The integrity constraints can be defined at column level or table level. The implementation and the syntax of the CREATE Statements differs for different RDBMS.

The Syntax for the CREATE TABLE Statement is:

```
CREATE TABLE table_name
(column_name1 datatype,
column_name2 datatype,
... column_nameN datatype
);
```

- ***table_name*** - is the name of the table.
- ***column_name1, column_name2....*** - is the name of the columns
- ***datatype*** - is the datatype for the column like char, date, number etc.

For Example: If you want to create the employee table, the statement would be like,

```
CREATE TABLE employee
( id number(5),
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

In Oracle database, the datatype for an integer column is represented as "number". In Sybase it is represented as "int".

Oracle provides another way of creating a table.

```
CREATE TABLE temp_employee
SELECT * FROM employee
```

In the above statement, temp_employee table is created with the same number of columns and datatype as employee table.

SQL ALTER TABLE Statement

The SQL ALTER TABLE command is used to modify the definition (structure) of a table by modifying the definition of its columns. The ALTER command is used to perform the following functions.

1) Add, drop, modify table columns
2) Add and drop constraints
3) Enable and Disable constraints

Syntax to add a column

```
ALTER TABLE table_name ADD column_name
datatype;
```

For Example: To add a column "experience" to the employee table, the query would be like

```
ALTER TABLE employee ADD experience number(3);
```

Syntax to drop a column

```
ALTER TABLE table_name DROP column_name;
```

For Example: To drop the column "location" from the employee table, the query would be like

```
ALTER TABLE employee DROP location;
```

Syntax to modify a column

```
ALTER TABLE table_name MODIFY column_name
datatype;
```

For Example: To modify the column salary in the employee table, the query would be like

```
ALTER TABLE employee MODIFY salary number(15,2);
```

SQL RENAME Command

The SQL RENAME command is used to change the name of the table or a database object.

If you change the object's name any reference to the old name will be affected. You have to manually change the old name to the new name in every reference.

Syntax to rename a table

```
RENAME old_table_name To new_table_name;
```

For Example: To change the name of the table employee to my_employee, the query would be like

```
RENAME employee TO my_emloyee;
```

SQL DELETE Statement

The DELETE Statement is used to delete rows from a table.

The Syntax of a SQL DELETE statement is:

```
DELETE FROM table_name [WHERE condition];
```

- table_name -- the table name which has to be updated.

NOTE: The WHERE clause in the sql delete command is optional and it identifies the rows in the column that gets deleted. If you do not include the WHERE clause all the rows in the table is deleted, so be careful while writing a DELETE query without WHERE clause.

For Example: To delete an employee with id 100 from the employee table, the sql delete query would be like,

```
DELETE FROM employee WHERE id = 100;
```

To delete all the rows from the employee table, the query would be like,

```
DELETE FROM employee;
```

SQL TRUNCATE Statement

The SQL TRUNCATE command is used to delete all the rows from the table and free the space containing the table.

Syntax to TRUNCATE a table:

```
TRUNCATE TABLE table_name;
```

For Example: To delete all the rows from employee table, the query would be like,

```
TRUNCATE TABLE employee;
```

Difference between DELETE and TRUNCATE Statements:

DELETE Statement: This command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.
TRUNCATE statement: This command is used to delete all the rows from the table and free the space containing the table.

SQL DROP Statement:

The SQL DROP command is used to remove an object from the database. If you drop a table, all the rows in the table is deleted and the table structure is removed from the database. Once a table is dropped we cannot get it back, so be careful while using DROP command. When a table is dropped all the references to the table will not be valid.

Syntax to drop a sql table structure:

```
DROP TABLE table_name;
```

For Example: To drop the table employee, the query would be like

```
DROP TABLE employee;
```

Difference between DROP and TRUNCATE Statement:

If a table is dropped, all the relationships with other tables will no longer be valid, the integrity constraints will be dropped, grant or access privileges on the table will also be dropped, if want use the table again it has to be recreated with the integrity constraints, access privileges and the relationships with other tables should be established again. But, if a table is truncated, the table structure remains the same, therefore any of the above problems will not exist.

CHAPTER – 3

SQL (ORACLE) CLAUSES

SQL WHERE Clause

The WHERE Clause is used when you want to retrieve specific information from a table excluding other irrelevant data. For example, when you want to see the information about students in class 10th only then you do need the information about the students in other class. Retrieving information about all the students would increase the processing time for the query.

So SQL offers a feature called WHERE clause, which we can use to restrict the data that is retrieved. The condition you provide in the WHERE clause filters the rows retrieved from the table and gives you only those rows which you expected to see. WHERE clause can be used along with SELECT, DELETE, UPDATE statements.

Syntax of SQL WHERE Clause:

```
WHERE {column or expression} comparison-
operator value

Syntax for a WHERE clause with Select
statement is:

SELECT column_list FROM table-name
WHERE condition;
```

- *column or expression* - Is the column of a table or a expression
- *comparison-operator* - operators like = < > etc.
- *value* - Any user value or a column name for comparison

For Example: To find the name of a student with id 100, the query would be like:

```
SELECT first_name, last_name FROM
student_details
WHERE id = 100;
```

Comparison Operators and Logical Operators are used in WHERE Clause. These operators are discussed in the next chapter.

NOTE: Aliases defined for the columns in the SELECT statement cannot be used in the WHERE clause to set conditions. Only aliases created for tables can be used to reference the columns in the table.

How to use expressions in the WHERE Clause?

Expressions can also be used in the WHERE clause of the SELECT statement.

For example: Lets consider the employee table. If you want to display employee name, current salary, and a 20% increase in the salary for only those products where the percentage increase in salary is greater than 30000, the SELECT statement can be written as shown below

```
SELECT name, salary, salary*1.2 AS new_salary FROM
employee
WHERE salary*1.2 > 30000;
```

Output:

```
name        salary      new_salary
----------- ----------  ----------------
Hrithik     35000       37000
Harsha      35000       37000
Priya       30000       360000
```

NOTE: Aliases defined in the SELECT Statement can be used in WHERE Clause.

SQL ORDER BY

The ORDER BY clause is used in a SELECT statement to sort results either in ascending or descending order. Oracle sorts query results in ascending order by default.

Syntax for using SQL ORDER BY clause to sort data is:

```
SELECT column-list
FROM table_name [WHERE condition]
[ORDER BY column1 [, column2, .. columnN] [DESC]];
```

Database table "employee";

id	name	dept	age	salary	location
100	Ramesh	Electrical	24	25000	Bangalore
101	Hrithik	Electronics	28	35000	Bangalore
102	Harsha	Aeronautics	28	35000	Mysore
103	Soumya	Electronics	22	20000	Bangalore
104	Priya	InfoTech	25	30000	Mangalore

For Example: **If you want to sort the employee table by salary of the employee, the sql query would be.**

```
SELECT name, salary FROM employee ORDER BY salary;
```

The output would be like:

```
   name        salary
----------   ----------
  Soumya       20000
  Ramesh       25000
  Priya        30000
  Hrithik      35000
  Harsha       35000
```

The query first sorts the result according to name and then displays it. You can also use more than one column in the ORDER BY clause. If you want to sort the employee table by the name and salary, the query would be like,

```
SELECT name, salary FROM employee ORDER BY name,
salary;
```

The output would be like:

```
    name           salary
-------------   -------------
   Soumya          20000
   Ramesh          25000
   Priya           30000
   Harsha          35000
   Hrithik         35000
```

NOTE: The columns specified in ORDER BY clause should be one of the columns selected in the SELECT column list. You can represent the columns in the ORDER BY clause by specifying the position of a column in the SELECT list, instead of writing the column name. The above query can also be written as given below,

```
SELECT name, salary FROM employee ORDER BY 1,
2;
```

By default, the ORDER BY Clause sorts data in ascending order. If you want to sort the data in descending order, you must explicitly specify it as shown below.

```
SELECT name, salary
FROM employee
ORDER BY name, salary DESC;
```

The above query sorts only the column 'salary' in descending order and the column 'name' by ascending order. If you want to select both name and salary in descending order, the query would be as given below.

```
SELECT name, salary
FROM employee
ORDER BY name DESC, salary DESC;
```

How to use expressions in the ORDER BY Clause?

Expressions in the ORDER BY clause of a SELECT statement.

For example: If you want to display employee name, current salary, and a 20% increase in the salary for only those employees for whom the percentage increase in salary is greater than 30000 and in descending order of the increased price, the SELECT statement can be written as shown below

```
SELECT name, salary, salary*1.2 AS new_salary
FROM employee
WHERE salary*1.2 > 30000
ORDER BY new_salary DESC;
```

The output for the above query is as follows.

```
   name        salary      new_salary
----------   ----------   -------------
  Hrithik      35000        37000
  Harsha       35000        37000
  Priya        30000        36000
```

NOTE: Aliases defined in the SELECT Statement can be used in ORDER BY Clause.

SQL GROUP BY Clause

The SQL GROUP BY Clause is used along with the group functions to retrieve data grouped according to one or more columns.

For Example: If you want to know the total amount of salary spent on each department, the query would be:

```
SELECT dept, SUM (salary)
FROM employee
GROUP BY dept;
```

The output would be like:

```
      dept              salary
----------------   --------------
   Electrical          25000
   Electronics         55000
   Aeronautics         35000
   InfoTech            30000
```

NOTE: The group by clause should contain all the columns in the select list expect those used along with the group functions.

```
SELECT location, dept, SUM (salary)
FROM employee
GROUP BY location, dept;
```

The output would be like:

```
location        dept          salary
------------  --------------  ----------
Bangalore     Electrical      25000
Bangalore     Electronics     55000
Mysore        Aeronautics     35000
Mangalore     InfoTech        30000
```

SQL HAVING Clause

Having clause is used to filter data based on the group functions. This is similar to WHERE condition but is used with group functions. Group functions cannot be used in WHERE Clause but can be used in HAVING clause.

For Example: If you want to select the department that has total salary paid for its employees more than 25000, the sql query would be like;

```
SELECT dept, SUM (salary)
FROM employee
GROUP BY dept
HAVING SUM (salary) > 25000
```

The output would be like:

```
dept            salary
------------    ------------
Electronics     55000
Aeronautics     35000
InfoTech        30000
```

When WHERE, GROUP BY and HAVING clauses are used together in a SELECT statement, the WHERE clause is processed first, then the rows that are returned after the WHERE clause is executed are grouped based on the GROUP BY clause. Finally, any conditions on the group functions in the HAVING clauses are applied to the grouped rows before the final output is displayed.

CHAPTER – 4

SQL (ORACLE) OPERATORS

SQL LOGICAL OPERATORS

There are three Logical Operators namely, AND, OR, and NOT. These operators compare two conditions at a time to determine whether a row can be selected for the output. When retrieving data using a SELECT statement, you can use logical operators in the WHERE clause, which allows you to combine more than one condition.

Logical Operators	Description
OR	For the row to be selected at least one of the conditions must be true.
AND	For a row to be selected all the specified conditions must be true.
NOT	For a row to be selected the specified condition must be false.

"OR" Logical Operator:

If you want to select rows that satisfy at least one of the given conditions, you can use the logical operator, OR. **For example:** if you want to find the names of students who are studying either Maths or Science, the query would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE subject = 'Maths' OR subject = 'Science'
```

The output would be something like,

```
first_name     last_name     subject
-------------  ------------- ----------
Anajali        Bhagwat       Maths
Shekar         Gowda         Maths
Rahul          Sharma        Science
Stephen        Fleming       Science
```

The following table describes how logical "OR" operator selects a row.

Column1 Satisfied?	Column2 Satisfied?	Row Selected
YES	YES	YES
YES	NO	YES
NO	YES	YES
NO	NO	NO

"AND" Logical Operator:

If you want to select rows that must satisfy all the given conditions, you can use the logical operator, AND.

For Example: To find the names of the students between the age 10 to 15 years, the query would be like:

```
SELECT first_name, last_name, age
FROM student_details
WHERE age >= 10 AND age <= 15;
```

The output would be something like,

```
first_name     last_name     age
-------------  ------------- ------
Rahul          Sharma        10
Anajali        Bhagwat       12
Shekar         Gowda         15
```

The following table describes how logical "AND" operator selects a row.

Column1 Satisfied?	Column2 Satisfied?	Row Selected
YES	YES	YES
YES	NO	NO
NO	YES	NO
NO	NO	NO

"NOT" Logical Operator:

If you want to find rows that do not satisfy a condition, you can use the logical operator, NOT. NOT results in the reverse of a condition. That is, if a condition is satisfied, then the row is not returned.

For example: If you want to find out the names of the students who do not play football, the query would be like:

```
SELECT first_name, last_name, games
FROM student_details
WHERE NOT games = 'Football'
```

The output would be something like,

```
first_name      last_name       games
--------------- --------------- -----------
Rahul           Sharma          Cricket
Stephen         Fleming         Cricket
Shekar          Gowda           Badminton
Priya           Chandra         Chess
```

The following table describes how logical "NOT" operator selects a row.

Column1 Satisfied?	NOT Column1 Satisfied?	Row Selected
YES	NO	NO
NO	YES	YES

NESTED Logical Operators:

You can use multiple logical operators in an SQL statement. When you combine the logical operators in a SELECT statement, the order in which the statement is processed is

1) NOT
2) AND
3) OR

For example: If you want to select the names of the students who age is between 10 and 15 years, or those who do not play football, the

```
SELECT statement would be
SELECT first_name, last_name, age, games
FROM student_details
WHERE age >= 10 AND age <= 15
OR NOT games = 'Football'
```

The output would be something like,

```
first_name      last_name      age       games
-------------   -------------  --------  ------------
Rahul           Sharma         10        Cricket
Priya           Chandra        15        Chess
```

In this case, the filter works as follows:

Condition 1: All the students you do not play football are selected.
Condition 2: All the students whose are aged between 10 and 15 are selected.
Condition 3: Finally the result is, the rows which satisfy at least one of the above conditions is returned.
NOTE: The order in which you phrase the condition is important, if the order changes you are likely to get a different result.

SQL COMPARISON OPERATORS:

Comparison operators are used to compare the column data with specific values in a condition. Comparison Operators are also used along with the SELECT statement to filter data based on specific conditions.

The below table describes each comparison operator.

Comparison Operators	Description
=	equal to
<>, !=	is not equal to
<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to

SQL Comparison Keywords

There are other comparison keywords available in sql which are used to enhance the search capabilities of a sql query. They are "IN", "BETWEEN...AND", "IS NULL", "LIKE".

Comparison Operators	Description
LIKE	Column value is similar to specified character(s).
IN	Column value is equal to any one of a specified set of values.
BETWEEN...AND	Column value is between two values, including the end values specified in the range.
IS NULL	Column value does not exist.

SQL LIKE Operator

The LIKE operator is used to list all rows in a table whose column values match a specified pattern. It is useful when you want to search rows to match a specific pattern, or when you do not know the entire value. For this purpose we use a wildcard character '%'. **For example:** To select all the students whose name begins with 'S'

```
SELECT first_name, last_name
FROM student_details
WHERE first_name LIKE 'S%';
```

The output would be similar to:

```
first_name      last_name
-------------   -------------
Stephen         Fleming
Shekar          Gowda
```

The above select statement searches for all the rows where the first letter of the column first_name is 'S' and rest of the letters in the name can be any character. There is another wildcard character you can use with LIKE operator. It is the underscore character, ' _ ' . In a search string, the underscore signifies a single character.

For example: to display all the names with 'a' second character,

```
SELECT first_name, last_name
FROM student_details
WHERE first_name LIKE '_a%';
```

The output would be similar to:

```
first_name      last_name
-------------   -------------
Rahul           Sharma
```

NOTE: Each underscore act as a placeholder for only one character. So you can use more than one underscore. Eg: ' __i% '-this has two underscores towards the left, 'S__ j%' - this has two underscores between character 'S' and 'i'.

SQL BETWEEN ... AND Operator

The operator BETWEEN and AND, are used to compare data for a range of values. **For Example:** to find the names of the students between age 10 to 15 years, the query would be like,

```
SELECT first_name, last_name, age
FROM student_details
WHERE age BETWEEN 10 AND 15;
```

The output would be similar to:

```
first_name      last_name       age
-------------   -------------   ------
Rahul           Sharma          10
Anajali         Bhagwat         12
Shekar          Gowda           15
```

SQL IN Operator:

The IN operator is used when you want to compare a column with more than one value. It is similar to an OR condition.

For example: If you want to find the names of students who are studying either Maths or Science, the query would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE subject IN ('Maths', 'Science');
```

The output would be similar to:

```
first_name      last_name       subject
-------------   -------------   ----------
Anajali         Bhagwat         Maths
Shekar          Gowda           Maths
Rahul           Sharma          Science
Stephen         Fleming         Science
```

You can include more subjects in the list like ('maths','science','history')

NOTE: The data used to compare is case sensitive.

SQL IS NULL Operator

A column value is NULL if it does not exist. The IS NULL operator is used to display all the rows for columns that do not have a value.

For Example: If you want to find the names of students who do not participate in any games, the query would be as given below

```
SELECT first_name, last_name
FROM student_details
WHERE games IS NULL
```

There would be no output as we have every student participate in a game in the table student_details, else the names of the students who do not participate in any games would be displayed

CHAPTER – 5

SQL INTEGRITY CONSTRAINTS

SQL Integrity Constraints

Integrity Constraints are used to apply business rules for the database tables.

The constraints available in SQL are **Foreign Key, Not Null, Unique, Check.**
Constraints can be defined in two ways:-

1) The constraints can be specified immediately after the column definition. This is called column-level definition.
2) The constraints can be specified after all the columns are defined. This is called table-level definition.

1) SQL Primary key:

This constraint defines a column or combination of columns which uniquely identifies each row in the table.

Syntax to define a Primary key at column level:

```
column name datatype [CONSTRAINT
constraint_name] PRIMARY KEY
```

Syntax to define a Primary key at table level:

```
[CONSTRAINT constraint_name] PRIMARY KEY
(column_name1,column_name2,..)
```

- **column_name1, column_name2** are the names of the columns which define the primary Key.
- The syntax within the bracket i.e. [CONSTRAINT constraint_name] is optional. **For Example:** To create an employee table with Primary Key constraint, the query would be like.

Primary Key at column level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

or

```
CREATE TABLE employee
( id number(5) CONSTRAINT emp_id_pk PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

Primary Key at column level:

```
CREATE TABLE employee
( id number(5),
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10),
CONSTRAINT emp_id_pk PRIMARY KEY (id)
);
```

Primary Key at table level:

```
CREATE TABLE employee
( id number(5), NOT NULL,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10),
ALTER TABLE employee ADD CONSTRAINT PK_EMPLOYEE_ID PRIMARY
KEY (id)
);
```

2) SQL Foreign key or Referential Integrity :

This constraint identifies any column referencing the PRIMARY KEY in another table. It establishes a relationship between two columns in the same table or between different tables. For a column to be defined as a Foreign Key, it should be a defined as a Primary Key in the table which it is referring. One or more columns can be defined as Foreign key.

Syntax to define a Foreign key at column level:

```
[CONSTRAINT constraint_name] REFERENCES
Referenced_Table_name(column_name)
```

Syntax to define a Foreign key at table level:

```
[CONSTRAINT constraint_name] FOREIGN
KEY(column_name) REFERENCES
referenced_table_name(column_name);
```

For Example:

1) Lets use the "product" table and "order_items".

Foreign Key at column level:

```
CREATE TABLE product
( product_id number(5) CONSTRAINT pd_id_pk
PRIMARY KEY,
product_name char(20),
supplier_name char(20),
unit_price number(10)
);
```

```
CREATE TABLE order_items
( order_id number(5) CONSTRAINT od_id_pk
PRIMARY KEY,
product_id number(5) CONSTRAINT pd_id_fk
REFERENCES, product(product_id),
product_name char(20),
supplier_name char(20),
unit_price number(10)
);
```

Foreign Key at table level:

```
CREATE TABLE order_items
( order_id number(5) ,
product_id number(5),
product_name char(20),
supplier_name char(20),
unit_price number(10)
CONSTRAINT od_id_pk PRIMARY KEY(order_id),
CONSTRAINT pd_id_fk FOREIGN KEY(product_id)
REFERENCES product(product_id)
);
```

2) If the employee table has a 'mgr_id' i.e, manager id as a foreign key which references primary key 'id' within the same table, the query would be like,

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
mgr_id number(5) REFERENCES employee(id),
salary number(10),
location char(10)
);
```

3) SQL Not Null Constraint:

This constraint ensures all rows in the table contain a definite value for the column which is specified as not null. Which means a null value is not allowed.

Syntax to define a Not Null constraint:

```
[CONSTRAINT constraint name] NOT NULL
```

For Example: To create a employee table with Null value, the query would be like

```
CREATE TABLE employee
( id number(5),
name char(20) CONSTRAINT nm_nn NOT NULL,
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

4) SQL Unique Key:

This constraint ensures that a column or a group of columns in each row have a distinct value. A column(s) can have a null value but the values cannot be duplicated.

Syntax to define a Unique key at column level:

```
[CONSTRAINT constraint_name] UNIQUE
```

Syntax to define a Unique key at table level:

```
[CONSTRAINT constraint_name]
UNIQUE(column_name)
```

For Example: To create an employee table with Unique key, the query would be like,

Unique Key at column level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10) UNIQUE
);
```

or

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10) CONSTRAINT loc_un UNIQUE
);
```

Unique Key at table level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10),
CONSTRAINT loc_un UNIQUE(location)
);
```

5) SQL Check Constraint:

This constraint defines a business rule on a column. All the rows must satisfy this rule. The constraint can be applied for a single column or a group of columns.

Syntax to define a Check constraint:

```
[CONSTRAINT constraint_name] CHECK (condition)
```

For Example: In the employee table to select the gender of a person, the query would be like

Check Constraint at column level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
gender char(1) CHECK (gender in ('M','F')),
salary number(10),
location char(10)
);
```

Check Constraint at table level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
gender char(1),
salary number(10),
location char(10),
CONSTRAINT gender_ck CHECK (gender in ('M','F'))
);
```

CHAPTER – 6

SQL OTHER NECESSARY TOPICS

SQL Commands:

SQL commands are instructions used to communicate with the database to perform specific task that work with data. SQL commands can be used not only for searching the database but also to perform various other functions like, for example, you can create tables, add data to tables, or modify data, drop the table, set permissions for users. SQL commands are grouped into four major categories depending on their functionality:

- **Data Definition Language (DDL)** - These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.
- **Data Manipulation Language (DML)** - These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.
- **Transaction Control Language (TCL)** - These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.
- **Data Control Language (DCL)** - These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

SQL Alias

SQL Aliases are defined for columns and tables. Basically aliases are created to make the column selected more readable.

For Example: To select the first name of all the students, the query would be like:

Aliases for columns:

```
SELECT first_name AS Name FROM student_details;
         or
SELECT first_name Name FROM student_details;
```

In the above query, the column first_name is given a alias as 'name'. So when the result is displayed the column name appears as 'Name' instead of 'first_name'.

Output:

```
    Name
 -------------
Rahul Sharma
Anjali Bhagwat
Stephen Fleming
Shekar Gowda
Priya Chandra
```

Aliases for tables:

```
SELECT s.first_name FROM student_details s;
```

In the above query, alias 's' is defined for the table student_details and the column first_name is selected from the table.

Aliases is more useful when

- There are more than one tables involved in a query,
- Functions are used in the query,
- The column names are big or not readable,
- More than one columns are combined together

SQL GROUP Functions

Group functions are built-in SQL functions that operate on groups of rows and return one value for the entire group. These functions are: **COUNT, MAX, MIN, AVG, SUM, DISTINCT**
SQL COUNT (): This function returns the number of rows in the table that satisfies the condition specified in the WHERE condition. If the WHERE condition is not specified, then the query returns the total number of rows in the table. **For Example:** If you want the number of employees in a particular department, the query would be:

```
SELECT COUNT (*) FROM employee
```

```
WHERE dept = 'Electronics';
```

The output would be '2' rows.

If you want the total number of employees in all the department, the query would take the form:

```
SELECT COUNT (*) FROM employee;
```

The output would be '5' rows.

SQL DISTINCT(): This function is used to select the distinct rows.
For Example: If you want to select all distinct department names from employee table, the query would be:

```
SELECT DISTINCT dept FROM employee;
```

To get the count of employees with unique name, the query would be:

```
SELECT COUNT (DISTINCT name) FROM employee;
```

SQL MAX(): This function is used to get the maximum value from a column. To get the maximum salary drawn by an employee, the query would be:

```
SELECT MAX (salary) FROM employee;
```

SQL MIN(): This function is used to get the minimum value from a column.
To get the minimum salary drawn by an employee, he query would be:

```
SELECT MIN (salary) FROM employee;
```

SQL AVG(): This function is used to get the average value of a numeric column.

To get the average salary, the query would be

```
SELECT AVG (salary) FROM employee;
```

SQL SUM(): This function is used to get the sum of a numeric column

To get the total salary given out to the employees,

```
SELECT SUM (salary) FROM employee;
```

SQL Joins

SQL Joins are used to relate information in different tables. A Join condition is a part of the sql query that retrieves rows from two or more tables. A SQL Join condition is used in the SQL WHERE Clause of select, update, delete statements.

The Syntax for joining two tables is:

```
SELECT col1, col2, col3...
FROM table_name1, table_name2
WHERE table_name1.col2 = table_name2.col1;
```

If a sql join condition is omitted or if it is invalid the join operation will result in a Cartesian product. The Cartesian product returns a number of rows equal to the product of all rows in all the tables being joined. For example, if the first table has 20 rows and the second table has 10 rows, the result will be 20 * 10, or 200 rows. This query takes a long time to execute.

Lets use the below two tables to explain the sql join conditions.

Database table "product";

product_id	product_name	supplier_name	unit_price
100	Camera	Nikon	300
101	Television	Onida	100
102	Refrigerator	Vediocon	150
103	Ipod	Apple	75
104	Mobile	Nokia	50

Database table "order_items";

order_id	product_id	total_units	customer
5100	104	30	Infosys
5101	102	5	Satyam
5102	103	25	Wipro
5103	101	10	TCS

SQL Joins can be classified into Equi join and Non Equi join.

1) SQL Equi joins

It is a simple sql join condition which uses the equal sign as the comparison operator. Two types of equi joins are SQL Outer join and SQL Inner join.

For example: You can get the information about a customer who purchased a product and the quantity of product.

2) SQL Non equi joins

It is a sql join condition which makes use of some comparison operator other than the equal sign like >, <, >=, <=

1) SQL Equi Joins:

An equi-join is further classified into three categories:
a) SQL Inner Join
b) SQL Outer Join

C) SQL Self Join**:**

a) SQL Inner Join:

All the rows returned by the sql query satisfy the sql join condition specified.

For example: If you want to display the product information for each order the query will be as given below. Since you are retrieving the data from two tables, you need to identify the common column between these two tables, which is the product_id.

The query for this type of sql joins would be like,

```
SELECT order_id, product_name, unit_price,
supplier_name, total_units
FROM product, order_items
WHERE order_items.product_id = product.product_id;
```

The columns must be referenced by the table name in the join condition, because product_id is a column in both the tables and needs a way to be identified. This avoids ambiguity in using the columns in the SQL SELECT statement.

The number of join conditions is (n-1), if there are more than two tables joined in a query where 'n' is the number of tables involved. The rule must be true to avoid Cartesian product.

We can also use aliases to reference the column name, then the above query would be like,

```
SELECT o.order_id, p.product_name, p.unit_price,
p.supplier_name, o.total_units
FROM product p, order_items o
WHERE o.product_id = p.product_id;
```

b) SQL Outer Join:

This sql join condition returns all rows from both tables which satisfy the join condition along with rows which do not satisfy the join condition from one of the tables. The sql outer join operator in Oracle is (+) and is used on one side of the join condition only.

The syntax differs for different RDBMS implementation. Few of them represent the join conditions as "sql left outer join", "sql right outer join".

If you want to display all the product data along with order items data, with null values displayed for order items if a product has no order item, the sql query for outer join would be as shown below:

```
SELECT p.product_id, p.product_name,
o.order_id, o.total_units
FROM order_items o, product p
WHERE o.product_id (+) =
p.product_id;
```

The output would be like,

```
product_id  product_name   order_id   total_units
-------------  -------------  -------------  -------------
   100           Camera
   101         Television       5103          10
   102        Refrigerator      5101           5
   103            Ipod          5102          25
   104           Mobile         5100          30
```

NOTE: If the (+) operator is used in the left side of the join condition it is equivalent to left outer join. If used on the right side of the join condition it is equivalent to right outer join.

C) SQL Self Join:

A Self Join is a type of sql join which is used to join a table to itself, particularly when the table has a FOREIGN KEY that references its own PRIMARY KEY. It is necessary to ensure that the join statement defines an alias for both copies of the table to avoid column ambiguity.

The below query is an example of a self join,

```
SELECT a.sales_person_id, a.name, a.manager_id,
b.sales_person_id, b.name
FROM sales_person a, sales_person b
WHERE a.manager_id = b.sales_person_id;
```

2) SQL Non Equi Join:

A Non Equi Join is a SQL Join whose condition is established using all comparison operators except the equal (=) operator. Like >=, <=, <, >

For example: If you want to find the names of students who are not studying either Economics, the sql query would be like, (lets use student_details table defined earlier.)

```
SELECT first_name, last_name, subject
FROM student_details
WHERE subject != 'Economics'
```

The output would be something like,

```
first_name   last_name    subject
-------------  -------------  -------------
  Anajali      Bhagwat      Maths
  Shekar        Gowda       Maths
   Rahul       Sharma      Science
  Stephen      Fleming     Science
```

SQL Views

A VIEW is a virtual table, through which a selective portion of the data from one or more tables can be seen. Views do not contain data of their own. They are used to restrict access to the database or to hide data complexity. A view is stored as a SELECT statement in the database. DML operations on a view like INSERT, UPDATE, DELETE affects the data in the original table upon which the view is based.

The Syntax to create a sql view is

```
CREATE VIEW view_name
AS
SELECT column_list
FROM table_name [WHERE condition];
```

- ***view_name*** is the name of the VIEW.
- The SELECT statement is used to define the columns and rows that you want to display in the view.

For Example: to create a view on the product table the sql query would be like

```
CREATE VIEW view_product
AS
SELECT product_id, product_name
FROM product;
```

SQL Subquery

Subquery or Inner query or Nested query is a query in a query. A subquery is usually added in the WHERE Clause of the sql statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value.

Subqueries are an alternate way of returning data from multiple tables.

Subqueries can be used with the following sql statements along with the comparison operators like =, <, >, >=, <= etc.

- SELECT
- INSERT
- UPDATE
- DELETE

For Example:

1) Usually, a subquery should return only one record, but sometimes it can also return multiple records when used with operators like IN, NOT IN in the where clause. The query would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE games NOT IN ('Cricket', 'Football');
```

The output would be similar to:

```
first_name     last_name      subject
-------------  -------------  ----------
  Shekar          Gowda       Badminton
   Priya         Chandra        Chess
```

2) Lets consider the student_details table which we have used earlier. If you know the name of the students who are studying science subject, you can get their id's by using this query below,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN ('Rahul', 'Stephen');
```

but, if you do not know their names, then to get their id's you need to write the query in this manner,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN (SELECT first_name
FROM student_details
WHERE subject= 'Science');
```

Output:

```
    id         first_name
  --------    -------------
   100           Rahul
   102           Stephen
```

In the above sql statement, first the inner query is processed first and then the outer query is processed.

3) Subquery can be used with INSERT statement to add rows of data from one or more tables to another table. Lets try to group all the students who study Maths in a table 'maths_group'.

```
INSERT INTO maths_group(id, name)
SELECT id, first_name || ' ' || last_name
FROM student_details WHERE subject= 'Maths'
```

4) A subquery can be used in the SELECT statement as follows. Lets use the product and order_items table defined in the sql_joins section.

```
select p.product_name, p.supplier_name, (select
order_id from order_items where product_id = 101)
as order_id from product p where p.product_id =
101
```

```
product_name         supplier_name        order_id
------------------   ------------------   ----------
  Television             Onida              5103
```

CORRELATED SUBQUERY

A query is called correlated subquery when both the inner query and the outer query are interdependent. For every row processed by the inner query, the outer query is processed as well. The inner query depends on the outer query before it can be processed.

```
SELECT p.product_name FROM product p
WHERE p.product_id = (SELECT o.product_id FROM
order_items o
WHERE o.product_id = p.product_id);
```

NOTE:
1) You can nest as many queries you want but it is recommended not to nest more than 16 Subqueries in oracle.
2) If a subquery is not dependent on the outer query it is called a non-correlated subquery.

SQL Index

Index in sql is created on existing tables to retrieve the rows quickly.

When there are thousands of records in a table, retrieving information will take a long time. Therefore indexes are created on columns which are accessed frequently, so that the information can be retrieved quickly. Indexes can be created on a single column or a group of columns. When a index is created, it first sorts the data and then it assigns a ROWID for each row.

Syntax to create Index:

```
CREATE INDEX index_name
ON table_name (column_name1,column_name2...);
```

Syntax to create SQL unique Index:

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name1,column_name2...);
```

- ***index_name*** is the name of the INDEX.
- ***table_name*** is the name of the table to which the indexed column belongs.
- ***column_name1, column_name2..*** is the list of columns which make up the INDEX.

In Oracle there are two types of SQL index namely, implicit and explicit.

Implicit Indexes:

They are created when a column is explicity defined with PRIMARY KEY, UNIQUE KEY Constraint.

Explicit Indexes:

They are created using the `"create index.. " syntax.`

NOTE:
1) Even though sql indexes are created to access the rows in the table quickly, they slow down DML operations like INSERT, UPDATE, DELETE on the table, because the indexes and tables both are updated along when a DML operation is performed. So use indexes only on columns which are used to search the table frequently.
2) Is is not required to create indexes on table which have less data.
3) In oracle database you can define up to sixteen (16) columns in an INDEX.

SQL GRANT Command

SQL GRANT is a command used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

```
GRANT privilege_name
ON object_name
TO {user_name |PUBLIC |role_name}
[WITH GRANT OPTION];
```

- ***privilege_name*** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
- ***object_name*** is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
- ***user_name*** is the name of the user to whom an access right is being granted.
- ***user_name*** is the name of the user to whom an access right is being granted.
- ***PUBLIC*** is used to grant access rights to all users.
- ***ROLES*** are a set of privileges grouped together.
- ***WITH GRANT OPTION*** - allows a user to grant access rights to other users.

For Example: GRANT SELECT ON employee TO user1;This command grants a SELECT permission on employee table to user1.You should use the WITH GRANT option carefully because for example if you GRANT SELECT privilege on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, still user2 will have SELECT privilege on employee table.

SQL REVOKE Command:

The REVOKE command removes user access rights or privileges to the database objects. The Syntax for the REVOKE command is:

```
REVOKE privilege_name
ON object_name
FROM {user_name |PUBLIC |role_name}
```

For Example: REVOKE SELECT ON employee FROM user1;This command will REVOKE a SELECT privilege on employee table from user1.When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.

PRIVILEGES AND ROLES:

Privileges: Privileges defines the access rights provided to a user on a database object. There are two types of privileges.

1) System privileges - This allows the user to CREATE, ALTER, or DROP database objects.
2) Object privileges - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply. Few CREATE system privileges are listed below:

System Privileges	Description
CREATE object	allows users to create the specified object in their own schema.
CREATE ANY object	allows users to create the specified object in any schema.

The above rules also apply for ALTER and DROP system privileges. Few of the object privileges are listed below:

Object Privileges	Description
INSERT	allows users to insert rows into a table.
SELECT	allows users to select data from a database object.
UPDATE	allows user to update data in a table.
EXECUTE	allows user to execute a stored procedure or a function.

Roles: Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges. You can either create Roles or use the system roles pre-defined by oracle.

Some of the privileges granted to the system roles are as given below:

System Role	Privileges Granted to the Role
CONNECT	CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.
RESOURCE	CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.
DBA	ALL SYSTEM PRIVILEGES

Creating Roles:

The Syntax to create a role is:
```
CREATE ROLE role_name
[IDENTIFIED BY password];
```
For Example: To create a role called "developer" with password as "pwd",the code will be as follows
```
CREATE ROLE testing
[IDENTIFIED BY pwd];
```

It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definitely have to identify it with the password. We can GRANT or REVOKE privilege to a role as below.
For example: To grant CREATE TABLE privilege to a user by creating a testing role:

First, create a testing Role

```
CREATE ROLE testing
```

Second, grant a CREATE TABLE privilege to the ROLE testing. You can add more privileges to the ROLE.

```
GRANT CREATE TABLE TO testing;
```

Third, grant the role to a user.

```
GRANT testing TO user1;
```

To revoke a CREATE TABLE privilege from testing ROLE, you can write:

```
REVOKE CREATE TABLE FROM testing;
```

The Syntax to drop a role from the database is as below:

```
DROP ROLE role_name;
```

For example: To drop a role called developer, you can write:

```
DROP ROLE testing;
```

CHAPTER –7

PL/SQL (PROCEDURAL LANGUAGE EXTENSION OF SQL)

PL/SQL

PL/SQL stands for Procedural Language extension of SQL.
PL/SQL is a combination of SQL along with the procedural features of programming languages.
It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

The PL/SQL Engine:

Oracle uses a PL/SQL engine to processes the PL/SQL statements. A PL/SQL code can be stored in the client system (client-side) or in the database (server-side).

A Simple PL/SQL Block:

Each PL/SQL program consists of SQL and PL/SQL statements which from a PL/SQL block.

A PL/SQL Block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory).
- The Exception (or Error) Handling section (optional).

Declaration Section:

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.

Execution Section:

The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

Exception Section:

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.

Every statement in the above three sections must end with a semicolon **;** . PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

This is how a sample PL/SQL Block looks.

```
DECLARE
   Variable declaration
BEGIN
   Program Execution
EXCEPTION
   Exception handling
END;
```

ADVANTAGES OF PL/SQL

- ***Block Structures*****:** PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.
- ***Procedural Language Capability*****:** PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).
- ***Better Performance*****:** PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.
- ***Error Handling*****:** PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

PL/SQL PLACEHOLDERS

Placeholders are temporary storage area. Placeholders can be any of Variables, Constants and Records. Oracle defines placeholders to store data temporarily, which are used to manipulate data during the execution of a PL SQL block.
Depending on the kind of data you want to store, you can define placeholders with a name and a datatype. Few of the datatypes used to define placeholders are as given below.

Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile

PL/SQL VARIABLES

These are placeholders that store the values that can change through the PL/SQL Block. The General Syntax to declare a variable is:

```
variable_name datatype [NOT NULL:= value];
```

- *variable_name* is the name of the variable.
- *datatype* is a valid PL/SQL datatype.
- *NOT NULL* is an optional specification on the variable.
- value or *DEFAULT* value is also an optional specification, where you can initialize a variable.
- Each variable declaration is a separate statement and must be terminated by a semicolon.

For example, if you want to store the current salary of an employee, you can use a variable.

```
DECLARE
salary number (6);
```

* “salary” is a variable of datatype number and of length 6.
When a variable is specified as NOT NULL, you must initialize the variable when it is declared.

For example: The below example declares two variables, one of which is a not null.

```
DECLARE
salary number(4);
dept varchar2(10) NOT NULL := “HR Dept”;
```

The value of a variable can change in the execution or exception section of the PL/SQL Block. We can assign values to variables in the two ways given below.
1) We can directly assign values to variables.
The General Syntax is:

```
variable_name:= value;
```

2) We can assign values to variables directly from the database columns by using a SELECT.. INTO statement. The General Syntax is:

```
SELECT column_name
INTO variable_name
FROM table_name
[WHERE condition];
```

Example: The below program will get the salary of an employee with id '1116' and display it on the screen.

```
DECLARE
 var_salary number(6);
 var_emp_id number(6) = 1116;
BEGIN
 SELECT salary
 INTO var_salary
 FROM employee
 WHERE emp_id = var_emp_id;
 dbms_output.put_line(var_salary);
 dbms_output.put_line('The employee’
        || var_emp_id || ' has salary  ' ||
var_salary);
END;
/
```

NOTE: The backward slash '/' in the above program indicates to execute the above PL/SQL Block.

SCOPE OF VARIABLES

PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.

Local variables - These are declared in a inner block and cannot be referenced by outside Blocks.

- *Global* variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

For Example: In the below example we are creating two variables in the outer block and assigning their product to the third variable created in the inner block.

The variable 'var_mult' is declared in the inner block, so cannot be accessed in the outer block i.e. it cannot be accessed after line 11. The variables 'var_num1' and 'var_num2' can be accessed anywhere in the block.

```
1> DECLARE
2>  var_num1 number;
3>  var_num2 number;
4> BEGIN
5>  var_num1 := 100;
6>  var_num2 := 200;
7>  DECLARE
8>   var_mult number;
9>   BEGIN
10>    var_mult := var_num1 * var_num2;
11>   END;
12> END;
13> /
```

PL/SQL CONSTANTS

As the name implies a constant is a value used in a PL/SQL Block that remains unchanged throughout the program. A constant is a user-defined literal value. You can declare a constant and use it instead of actual value.
For example: If you want to write a program which will increase the salary of the employees by 25%, you can declare a constant and use it throughout the program. Next time when you want to increase the salary again you can change the value of the constant which will be easier than changing the actual value throughout the program.

The General Syntax to declare a constant is:

```
constant_name CONSTANT datatype:= VALUE;
```

- *constant_name* is the name of the constant i.e. similar to a variable name.
- The word *CONSTANT* is a reserved word and ensures that the value does not change.
- *VALUE* - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

For example, to declare salary_increase, you can write code as follows:

```
DECLARE
salary_increase CONSTANT number (3) := 10;
```

You *must* assign a value to a constant at the time you declare it. If you do not assign a value to a constant while declaring it and try to assign a value in the execution section, you will get a error. If you execute the below Pl/SQL block you will get error.

```
DECLARE
 salary_increase CONSTANT number(3);
BEGIN
 salary_increase := 100;
 dbms_output.put_line (salary_increase);
END;
```

PL/SQL RECORDS

Records are another type of datatypes which oracle allows to be defined as a placeholder. Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc. Each scalar data types in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row.

Declaring a record:

To declare a record, you must first define a composite datatype; then declare a record for that type.

The General Syntax to define a composite datatype is:

```
TYPE record_type_name IS RECORD
(first_col_name column_datatype,
second_col_name column_datatype, ...);
```

- *record_type_name* – it is the name of the composite type you want to define.
- *first_col_name, second_col_name, etc.,- it is the* names the fields/columns within the record.
- *column_datatype* defines the scalar datatype of the fields.

There are different ways you can declare the datatype of the fields.
1) You can declare the field in the same way as you declare the fieds while creating the table.
2) If a field is based on a column from database table, you can define the field_type as follows:

```
col_name table_name.column_name%type;
```

By declaring the field datatype in the above method, the datatype of the column is dynamically applied to the field. This method is useful when you are altering the column specification of the table, because you do not need to change the code again.

NOTE: You can use also *%type* to declare variables and constants. The General Syntax to declare a record of a uer-defined datatype is:

```
record_name record_type_name;
```

The following code shows how to declare a record called *employee_rec* based on a user-defined type.

```
DECLARE
TYPE employee_type IS RECORD
(employee_id number(5),
 employee_first_name varchar2(25),
 employee_last_name employee.last_name%type,
 employee_dept employee.dept%type);
 employee_salary employee.salary%type;
 employee_rec employee_type;
```

If all the fields of a record are based on the columns of a table, we can declare the record as follows:

```
record_name table_name%ROWTYPE;
```

For example, the above declaration of employee_rec can as follows:

```
DECLARE
 employee_rec employee%ROWTYPE;
```

The advantages of declaring the record as a ROWTYPE are:
1) You do not need to explicitly declare variables for all the columns in a table.
2) If you alter the column specification in the database table, you do not need to update the code.

The disadvantage of declaring the record as a ROWTYPE is:
1) When u create a record as a ROWTYPE, fields will be created for all the columns in the table and memory will be used to create the datatype for all the fields. So use ROWTYPE only when you are using all the columns of the table in the program. **NOTE:** When you are creating a record, you are just creating a datatype, similar to creating a variable. You need to assign values to the record to use them. The following table consolidates the different ways in which you can define and declare a pl/sql record.

Syntax	Usage
TYPE record_type_name IS RECORD (column_name1 datatype, column_name2 datatype, ...);	Define a composite datatype, where each field is scalar.
col_name table_name.column_name%type;	Dynamically define the datatype of a column based on a database column.
record_name record_type_name;	Declare a record based on a user-defined type.
record_name table_name%ROWTYPE;	Dynamically declare a record based on an entire row of a table. Each column in the table corresponds to a field in the record.

Passing Values To and From a Record

When you assign values to a record, you actually assign values to the fields within it. The General Syntax to assign a value to a column within a record direclty is:

```
record_name.col_name:= value;
```

If you used %ROWTYPE to declare a record, you can assign values as shown:

```
record_name.column_name:= value;
```

We can assign values to records using SELECT Statements as shown:

```
  SELECT col1, col2
  INTO record_name.col_name1,
record_name.col_name2
  FROM table_name
  [WHERE clause];
```

If %ROWTYPE is used to declare a record then you can directly assign values to the whole record instead of each columns separately.

In this case, you must SELECT all the columns from the table into the record as shown:

```
  SELECT * INTO record_name
  FROM table_name
  [WHERE clause];
```

Lets see how we can get values from a record.

The General Syntax to retrieve a value from a specific field into another variable is:

```
  var_name:= record_name.col_name;
```

The following table consolidates the different ways you can assign values to and from a record:

Syntax	**Usage**
record_name.col_name := value;	To directly assign a value to a specific column of a record.
record_name.column_name := value;	To directly assign a value to a specific column of a record, if the record is declared using %ROWTYPE.
SELECT col1, col2 INTO record_name.col_name1, record_name.col_name2 FROM table_name [WHERE clause];	To assign values to each field of a record from the database table.
SELECT * INTO record_name FROM table_name [WHERE clause];	To assign a value to all fields in the record from a database table.
variable_name := record_name.col_name;	To get a value from a record column and assigning it to a variable.

CONDITIONAL STATEMENTS IN PL/SQL

As the name implies, PL/SQL supports programming language features like conditional statements, iterative statements.

The programming constructs are similar to how you use in programming languages like Java and C++.

IF THEN ELSE STATEMENT

1)
IF condition
THEN
statement 1;
ELSE
statement 2;
END IF;
 	
2)
IF condition 1
THEN
statement 1;
statement 2;
ELSIF condtion2 THEN
statement 3;
ELSE
statement 4;
END IF
	3)
	IF condition 1
	THEN
	statement 1;
	statement 2;
	ELSIF condtion2
	THEN
	statement 3;
	ELSE
	statement 4;
	END IF;
	
	4)
	IF condition1 THEN
	ELSE
	IF condition2 THEN
	statement1;
	END IF;
	ELSIF condition3
	THEN
	statement2;
	END IF;

ITERATIVE STATEMENTS IN PL/SQL

An iterative control Statements are used when we want to repeat the execution of one or more statements for specified number of times.

There are three types of loops in PL/SQL:

1. **Simple Loop**
2. **While Loop**
3. **For Loop**

1) Simple Loop
A Simple Loop is used when a set of statements is to be executed at least once before the loop terminates. An EXIT condition must be specified in the loop, otherwise the loop will get into an infinite number of iterations. When the EXIT condition is satisfied the process exits from the loop.

The General Syntax to write a Simple Loop is:

```
LOOP
   statements;
   EXIT;
   {or EXIT WHEN condition;}
END LOOP;
```

These are the important steps to be followed while using Simple Loop.

- Initialize a variable before the loop body.
- Increment the variable in the loop.
- Use a EXIT WHEN statement to exit from the Loop. If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

2) While Loop
A WHILE LOOP is used when a set of statements has to be executed as long as a condition is true. The condition is evaluated at the beginning of each iteration. The iteration continues until the condition becomes false. The General Syntax to write a WHILE LOOP is:

```
WHILE <condition>
 LOOP statements;
END LOOP;
```

Important steps to follow when executing a while loop:

- Initialize a variable before the loop body.
- Increment the variable in the loop.
- EXIT WHEN statement and EXIT statements can be used in while loops but it's not done oftenly.

3) FOR Loop
A FOR LOOP is used to execute a set of statements for a predetermined number of times. Iteration occurs between the start and end integer values given. The counter is always incremented by 1. The loop exits when the counter reaches the value of the end integer. The General Syntax to write a FOR LOOP is:

```
FOR counter IN val1..val2
  LOOP statements;
END LOOP;
```

- val1 - Start integer value.
- val2 - End integer value.

Important steps to follow when executing a while loop:

- The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicity.
- The counter variable is incremented by 1 and does not need to be incremented explicitly.
- EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done oftenly.

NOTE: The above Loops are explained with a example when dealing with Explicit Cursors.

PL/SQL CURSORS

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.

There are two types of cursors in PL/SQL:

Implicit cursors: These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

Explicit cursors:
They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.
Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

- **Implicit Cursors:**

When you execute DML statements like DELETE, INSERT, UPDATE and SELECT statements, implicit statements are created to process these statements.
Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.
For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected.
When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

The status of the cursor for each of these attributes are defined in the below table.

Attributes	**Return Value**	**Example**
%FOUND	The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ....INTO statement return at least one row.	SQL%FOUND
	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT....INTO statement do not return a row.	
%NOTFOUND	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE at least one row and if SELECT ....INTO statement return at least one row.	SQL%NOTFOUND
	The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ....INTO statement does not return a row.	
%ROWCOUNT	Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT	SQL%ROWCOUNT

For Example: Consider the PL/SQL Block that uses implicit cursor attributes as shown below:

```
DECLARE var_rows number(5);
BEGIN
```

```
      UPDATE employee
      SET salary = salary + 1000;
      IF SQL%NOTFOUND THEN
        dbms_output.put_line('None of the
salaries where updated');
      ELSIF SQL%FOUND THEN
        var_rows:= SQL%ROWCOUNT;
        dbms_output.put_line('Salaries for' ||
var_rows || 'employees are updated');
      END IF;
    END;
```

In the above PL/SQL Block, the salaries of all the employees in the 'employee' table are updated. If none of the employee's salary are updated we get a message 'None of the salaries where updated'. Else we get a message like for example, 'Salaries for 1000 employees are updated' if there are 1000 rows in 'employee' table.

PL/SQL STORED PROCEDURES

A stored procedure or in simple a proc is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages. A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.
We can pass parameters to procedures in three ways.

1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters
A procedure may or may not return any value.

General Syntax to create a procedure is:

```
      CREATE    [OR    REPLACE]    PROCEDURE
proc_name [list of parameters]
      IS
         Declaration section
      BEGIN
         Execution section
      EXCEPTION
        Exception section
      END;
```

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

The below example creates a procedure 'employer_details' which gives the details of the employee.

```
      1> CREATE OR REPLACE PROCEDURE
employer_details
      2> IS
      3>  CURSOR emp_cur IS
      4>  SELECT first_name, last_name, salary
FROM emp_tbl;
      5>  emp_rec emp_cur%rowtype;
      6> BEGIN
      7>  FOR emp_rec in sales_cur
      8>  LOOP
      9>  dbms_output.put_line(emp_cur.first_name
|| ' ' ||emp_cur.last_name
      10>    || ' ' ||emp_cur.salary);
      11> END LOOP;
      12>END;
      13> /
```

How to execute a Stored Procedure?

There are two ways to execute a procedure.
1) From the SQL prompt.

```
        EXECUTE [or EXEC] procedure_name;
```

2) Within another procedure – simply use the procedure name.

```
          procedure_name;
```

NOTE: In the examples given above, we are using backward slash '/' at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

PL/SQL FUNCTIONS

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

The General Syntax to create a function is:

```
      CREATE [OR REPLACE] FUNCTION
function_name [parameters]
      RETURN return_datatype;
      IS
      Declaration_section
      BEGIN
      Execution_section
      Return return_variable;
      EXCEPTION
      exception section
      Return return_variable;
      END;
```

1) Return Type: The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.

2) The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let’s create a function called "employer_details_func' similar to the one created in stored procedure.

```
        1> CREATE OR REPLACE FUNCTION
        employer_details_func
        2>    RETURN VARCHAR(20);
        3> IS
        5>    emp_name VARCHAR(20);
        6> BEGIN
        7>  SELECT first_name INTO emp_name
        8>  FROM emp_tbl WHERE empID = '100';
        9>  RETURN emp_name;
        10> END;
        11> /
```

In the example we are retrieving the ‘first_name’ of employee with empID 100 to variable ‘emp_name’.
The return type of the function is VARCHAR which is declared in line no 2.
The function returns the 'emp_name' which is of type VARCHAR as the return value in line no 9.

How to execute a PL/SQL Function?

A function can be executed in the following ways.
1) Since a function returns a value we can assign it to a variable.

```
        employee_name :=employer_details_func;
```

If ‘employee_name’ is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

2) As a part of a SELECT statement

```
        SELECT employer_details_func FROM
dual;
```

3) In a PL/SQL Statements like,

```
        dbms_output.put_line(employer_details_
func);
```

This line displays the value returned by the function.

PARAMETERS IN PROCEDURE AND FUNCTIONS

How to pass parameters to Procedures and Functions in PL/SQL?
In PL/SQL, we can pass parameters to procedures and functions in three ways.

1) IN type parameter: These types of parameters are used to send values to stored procedures.

2) OUT type parameter: These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
3) IN OUT parameter: These types of parameters are used to send values and get values from stored procedures.
NOTE: If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.

1) IN parameter:

This is similar to passing parameters in programming languages. We can pass values to the stored procedure through these parameters or variables. This type of parameter is a read only parameter. We can assign the value of IN type parameter to a variable or use it in a query, but we cannot change its value inside the procedure.

The General syntax to pass a IN parameter is-

```
     CREATE [OR REPLACE] PROCEDURE
procedure_name (
       param_name1 IN datatype, param_name12
IN datatype ... )
```

- param_name1, param_name2... are unique parameter names.
- datatype - defines the datatype of the variable.
- IN - is optional, by default it is a IN type parameter.

2) OUT Parameter:

The OUT parameters are used to send the OUTPUT from a procedure or a function. This is a write-only parameter i.e, we cannot pass values to OUT parameters while executing the stored procedure, but we can assign values to OUT parameter inside the stored procedure and the calling program can receive this output value.
The General syntax to create an OUT parameter is-

```
   CREATE [OR REPLACE] PROCEDURE proc2
(param_name OUT datatype)
```

The parameter should be explicity declared as OUT parameter.

3) IN OUT Parameter:

The IN OUT parameter allows us to pass values into a procedure and get output values from the procedure. This parameter is used if the value of the IN parameter can be changed in the calling program.
By using IN OUT parameter we can pass values into a parameter and return a value to the calling program using the same parameter. But this is possible only if the value passed to the procedure and output value have a same datatype. This parameter is used if the value of the parameter will be changed in the procedure.
The General syntax to create an IN OUT parameter is-

```
   CREATE [OR REPLACE] PROCEDURE proc3
(param_name IN OUT datatype)
```

The below examples show how to create stored procedures using the above three types of parameters.

Example1:

Using IN and OUT parameter:

Let’s create a procedure which gets the name of the employee when the employee id is passed.

```
   1> CREATE OR REPLACE PROCEDURE emp_name
(id IN NUMBER, emp_name OUT NUMBER)
   2> IS
   3> BEGIN
   4>    SELECT first_name INTO emp_name
   5>    FROM emp_tbl WHERE empID = id;
   6> END;
   7> /
```

We can call the procedure ‘emp_name’ in this way from a PL/SQL Block.

```
  1> DECLARE
  2>  empName varchar(20);
  3>  CURSOR id_cur SELECT id FROM emp_ids;
  4> BEGIN
  5> FOR emp_rec in id_cur
  6> LOOP
  7>   emp_name(emp_rec.id, empName);
  8>   dbms_output.putline('The employee ' ||
empName || ' has id ' || emp-rec.id);
  9> END LOOP;
  10> END;
  11> /
```

In the above PL/SQL Block
In line no 3; we are creating a cursor 'id_cur' which contains the employee id.
In line no 7; we are calling the procedure 'emp_name', we are passing the 'id' as IN parameter and 'empName' as OUT parameter.
In line no 8; we are displaying the id and the employee name which we got from the procedure 'emp_name'.

Example 2:

Using IN OUT parameter in procedures:

```
  1> CREATE OR REPLACE PROCEDURE emp_salary_increase
  2> (emp_id IN emptbl.empID%type, salary_inc IN OUT
emptbl.salary%type)
  3> IS
  4>    tmp_sal number;
  5> BEGIN
  6>    SELECT salary
  7>    INTO tmp_sal
  8>    FROM emp_tbl
  9>    WHERE empID = emp_id;
  10>   IF tmp_sal between 10000 and 20000 THEN
  11>      salary_inout := tmp_sal * 1.2;
  12>   ELSIF tmp_sal between 20000 and 30000 THEN
  13>      salary_inout := tmp_sal * 1.3;
  14>   ELSIF tmp_sal > 30000 THEN
  15>      salary_inout := tmp_sal * 1.4;
  16>   END IF;
  17> END;
  18> /
```

The below PL/SQL block shows how to execute the above 'emp_salary_increase' procedure.

```
  1> DECLARE
  2>    CURSOR updated_sal is
  3>    SELECT empID,salary
  4>    FROM emp_tbl;
  5>    pre_sal number;
  6> BEGIN
  7>   FOR emp_rec IN updated_sal LOOP
  8>       pre_sal := emp_rec.salary;
  9>       emp_salary_increase(emp_rec.empID,
emp_rec.salary);
  10>       dbms_output.put_line('The salary of ' ||
emp_rec.empID ||
  11>                ' increased from '|| pre_sal ||
' to '||emp_rec.salary);
  12>   END LOOP;
  13> END;
  14> /
```

EXCEPTION HANDLING

In this section we will discuss about the following,
1) What is Exception Handling.
2) Structure of Exception Handling.
3) Types of Exception Handling.

1) What is Exception Handling?
PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling. Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs a messages which explains its cause is received.

PL/SQL Exception message consists of three parts.

1) Type of Exception
2) An Error Code
3) A message
By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

2) Structure of Exception Handling.
The General Syntax for coding the exception section

```
 DECLARE
   Declaration section
 BEGIN
   Exception section
 EXCEPTION
 WHEN ex_name1 THEN
    -Error handling statements
 WHEN ex_name2 THEN
    -Error handling statements
 WHEN Others THEN
   -Error handling statements
END;
```

General PL/SQL statements can be used in the Exception Block.
When an exception is raised, Oracle searches for an appropriate exception handler in the exception section. For example in the above example, if the error raised is 'ex_name1', then the error is handled according to the statements under it. Since, it is not possible to determine all the possible runtime errors during testing for the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled. Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

```
DELCARE
   Declaration section
 BEGIN
    DECLARE
      Declaration section
    BEGIN
      Execution section
    EXCEPTION
      Exception section
    END;
 EXCEPTION
   Exception section
 END;
```

In the above case, if the exception is raised in the inner block it should be handled in the exception block of the inner PL/SQL block else the control moves to the Exception block of the next upper PL/SQL Block. If none of the blocks handle the exception the program ends abruptly with an error.

3) Types of Exception.

There are 3 types of Exceptions.

a) Named System Exceptions
b) Unnamed System Exceptions
c) User-defined Exceptions

a) Named System Exceptions

System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule. There are some system exceptions which are raised frequently, so they are pre-defined and given a name in Oracle which are known as Named System Exceptions.
For example: NO_DATA_FOUND and ZERO_DIVIDE are called Named System exceptions.

Named system exceptions are:
1) Not Declared explicitly,
2) Raised implicitly when a predefined Oracle error occurs,
3) caught by referencing the standard name within an exception-handling routine.

Exception Name	Reason	Error Number
CURSOR_ALREADY_OPEN	When you open a cursor that is already open.	ORA-06511
INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.	ORA-01001
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.	ORA-01403
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.	ORA-01422
ZERO_DIVIDE	When you attempt to divide a number by zero.	ORA-01476

For Example: Suppose a NO_DATA_FOUND exception is raised in a proc, we can write a code to handle the exception as given below.

```
BEGIN
  Execution section
EXCEPTION
WHEN NO_DATA_FOUND THEN
 dbms_output.put_line ('A SELECT...INTO did
not return any row.');
 END;
```

b) Unnamed System Exceptions

Those system exception for which oracle does not provide a name is known as unnamed system exception. These exceptions do not occur frequently. These Exceptions have a code and an associated message.

There are two ways to handle unnamed system exceptions:
1. By using the WHEN OTHERS exception handler, or
2. By associating the exception code to a name and using it as a named exception.

We can assign a name to unnamed system exceptions using
a Pragma called EXCEPTION_INIT.
EXCEPTION_INIT will associate a predefined Oracle error number to a programmer_defined exception name.

Steps to be followed to use unnamed system exceptions are-

• They are raised implicitly.
• If they are not handled in WHEN Others they must be handled explicity.

• To handle the exception explicity, they must be declared using Pragma EXCEPTION_INIT as given above and handled referencing the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION_INIT is:

```
      DECLARE
         exception_name EXCEPTION;
         PRAGMA
         EXCEPTION_INIT (exception_name,
Err_code);
      BEGIN
      Execution section
      EXCEPTION
        WHEN exception_name THEN
           handle the exception
      END;
```

For Example: Lets consider the product table and order_items table from sql joins.
Here product_id is a primary key in product table and a foreign key in order_items table.
If we try to delete a product_id from the product table when it has child records in order_id table an exception will be thrown with oracle code number -2292.
We can provide a name to this exception and handle it in the exception section as given below.

```
      DECLARE
        Child_rec_exception EXCEPTION;
        PRAGMA
         EXCEPTION_INIT (Child_rec_exception, -
2292);
      BEGIN
        Delete FROM product where product_id= 104;
      EXCEPTION
         WHEN Child_rec_exception
         THEN Dbms_output.put_line('Child records
are present for this product_id.');
      END;
      /
```

c) User-defined Exceptions

Apart from system exceptions we can explicity define exceptions based on business rules. These are known as user-defined exceptions.

Steps to be followed to use user-defined exceptions:
• They should be explicitly declared in the declaration section.
• They should be explicitly raised in the Execution Section.
• They should be handled by referencing the user-defined exception name in the exception section.

For Example: Lets consider the product table and order_items table from sql joins to explain user-defined exception. Lets create a business rule that if the total no of units of any particular product sold is more than 20, then it is a huge quantity and a special discount should be provided.

```
DECLARE
  huge_quantity EXCEPTION;
  CURSOR product_quantity is
  SELECT p.product_name as name, sum(o.total_units)
as units
  FROM order_tems o, product p
  WHERE o.product_id = p.product_id;
  quantity order_tems.total_units%type;
  up_limit CONSTANT order_tems.total_units%type :=
20;
  message VARCHAR2(50);
BEGIN
  FOR product_rec in product_quantity LOOP
    quantity := product_rec.units;
     IF quantity > up_limit THEN
      message := 'The number of units of product '
|| product_rec.name ||
                 ' is more than 20. Special
discounts should be provided.
                     Rest of the records are
skipped. '
     RAISE huge_quantity;
     ELSIF quantity < up_limit THEN
      v_message:= 'The number of unit is below the
discount limit.';
     END IF;
     dbms_output.put_line (message);
  END LOOP;
 EXCEPTION
   WHEN huge_quantity THEN
     dbms_output.put_line (message);
 END;
/
```

RAISE_APPLICATION_ERROR is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999. Whenever a message is displayed using RAISE_APPLICATION_ERROR, all previous transactions which are not committed within the PL/SQL Block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements). RAISE_APPLICATION_ERROR raises an exception but does not handle it. RAISE_APPLICATION_ERROR is used for the following reasons,
a) to create a unique id for an user-defined exception.
b) to make the user-defined exception look like an Oracle error.

The General Syntax to use this procedure is:

```
  RAISE_APPLICATION_ERROR (error_number,
error_message);
```

• The Error number must be between -20000 and -20999
• The Error_message is the message you want to display when the error occurs.

Steps to be followed to use RAISE_APPLICATION_ERROR procedure:

1. Declare a user-defined exception in the declaration section.
2. Raise the user-defined exception based on a specific business rule in the execution section.
3. Finally, catch the exception and link the exception to a user-defined error number in RAISE_APPLICATION_ERROR.

Using the above example we can display a error message using RAISE_APPLICATION_ERROR.

```
    DECLARE
      huge_quantity EXCEPTION;
      CURSOR product_quantity is
      SELECT p.product_name as name,
sum(o.total_units) as units
      FROM order_tems o, product p
      WHERE o.product_id = p.product_id;
      quantity order_tems.total_units%type;
      up_limit CONSTANT
order_tems.total_units%type := 20;
      message VARCHAR2(50);
    BEGIN
      FOR product_rec in product_quantity LOOP
        quantity := product_rec.units;
         IF quantity > up_limit THEN
            RAISE huge_quantity;
         ELSIF quantity < up_limit THEN
          v_message:= 'The number of unit is below
the discount limit.';
         END IF;
         Dbms_output.put_line (message);
      END LOOP;
     EXCEPTION
       WHEN huge_quantity THEN
          raise_application_error(-2100, 'The
number of unit is above the discount limit.');
     END;
    /
```

PL/SQL TRIGGERS

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Syntax of Triggers

The Syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
 {BEFORE | AFTER | INSTEAD OF }
 {INSERT [OR] | UPDATE [OR] | DELETE}
 [OF col_name]
 ON table_name
 [REFERENCING OLD AS o NEW AS n]
 [FOR EACH ROW]
 WHEN (condition)
 BEGIN
   --- sql statements
 END;
```

- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *{BEFORE | AFTER | INSTEAD OF }* - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- *{INSERT [OR] | UPDATE [OR] | DELETE}* - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- *[OF col_name]* - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *[ON table_name]* - This clause identifies the name of the table or view to which the trigger is associated.
- *[REFERENCING OLD AS o NEW AS n]* - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- *[FOR EACH ROW]* - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- *WHEN (condition)* - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For Example: The price of a product changes constantly. It is important to maintain the history of the prices of the products.
We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'product_price_history' table

```
      CREATE TABLE product_price_history
      (product_id number(5),
      product_name varchar2(32),
      supplier_name varchar2(32),
      unit_price number(7,2) );

      CREATE TABLE product
      (product_id number(5),
      product_name varchar2(32),
      supplier_name varchar2(32),
      unit_price number(7,2) );
```

2) Create the price_history_trigger and execute it.

```
      CREATE or REPLACE TRIGGER
price_history_trigger
      BEFORE UPDATE OF unit_price
      ON product
      FOR EACH ROW
      BEGIN
      INSERT INTO product_price_history
      VALUES
      (:old.product_id,
       :old.product_name,
       :old.supplier_name,
       :old.unit_price);
      END;
      /
```

3) Lets update the price of a product.

```
      UPDATE PRODUCT SET unit_price = 800 WHERE
product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.

4) If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

Types of PL/SQL Triggers

There are two types of triggers based on which level it is triggered.

1) Row level trigger - An event is triggered for each row updated, inserted or deleted.
2) Statement level trigger - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

1) BEFORE statement trigger fires first.

2) Next BEFORE row level trigger fires, once for each row affected.

3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.

4) Finally the AFTER statement level trigger fires.

For Example: Let's create a table 'product_check' which we can use to store messages when triggers are fired.

```
      CREATE TABLE product
      (Message varchar2(50),
       Current_Date number(32)
      );
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

1) BEFORE UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' before a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER Before_Update_Stat_product
BEFORE
UPDATE ON product
Begin
INSERT INTO product_check
Values('Before update, statement level',sysdate);
END;
/
```

2) BEFORE UPDATE, Row Level: This trigger will insert a record into the table 'product_check' before each row is updated.

```
CREATE or REPLACE TRIGGER Before_Upddate_Row_product
 BEFORE
 UPDATE ON product
 FOR EACH ROW
 BEGIN
 INSERT INTO product_check
 Values('Before update row level',sysdate);
 END;
 /
```

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' after a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER After_Update_Stat_product
AFTER
UPDATE ON product
BEGIN
INSERT INTO product_check
Values('After update, statement level', sysdate);
End;
/
```

4) AFTER UPDATE, Row Level: This trigger will insert a record into the table 'product_check' after each row is updated.

```
CREATE or REPLACE TRIGGER After_Update_Row_product
AFTER
insert On product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values('After update, Row level',sysdate);
END;
/
```

Now lets execute a update statement on table product.

```
UPDATE PRODUCT SET unit_price = 800
WHERE product_id in (100,101);
```

Lets check the data in 'product_check' table to see the order in which the trigger is fired.

```
SELECT * FROM product_check;
```

Output:

```
      Message                                     Current_Date
------------------------------------------------------------
Before update, statement level                26-Nov-2008
Before update, row level                      26-Nov-2008
After update, Row level                       26-Nov-2008
Before update, row level                      26-Nov-2008
After update, Row level                       26-Nov-2008
After update, statement level                 26-Nov-2008
```

The above result shows 'before update' and 'after update' row level events have occurred twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

How to know Information about Triggers?

We can use the data dictionary view 'USER_TRIGGERS' to obtain information about any trigger.
The below statement shows the structure of the view 'USER_TRIGGERS'

```
    DESC USER_TRIGGERS;
```

```
   NAME                            Type
--------------------------------------------
TRIGGER_NAME                 VARCHAR2(30)
TRIGGER_TYPE                 VARCHAR2(16)
TRIGGER_EVENT                VARCHAR2(75)
TABLE_OWNER                  VARCHAR2(30)
BASE_OBJECT_TYPE             VARCHAR2(16)
TABLE_NAME                   VARCHAR2(30)
COLUMN_NAME                  VARCHAR2(4000)
REFERENCING_NAMES            VARCHAR2(128)
WHEN_CLAUSE                  VARCHAR2(4000)
STATUS                       VARCHAR2(8)
DESCRIPTION                  VARCHAR2(4000)
ACTION_TYPE                  VARCHAR2(11)
TRIGGER_BODY                 LONG
```

This view stores information about header and body of the trigger.

```
   SELECT * FROM user_triggers WHERE trigger_name = 'Before_Update_Stat_product';
```

The above sql query provides the header and body of the trigger 'Before_Update_Stat_product'.
You can drop a trigger using the following command.

```
   DROP TRIGGER trigger_name;
```

CYCLIC CASCADING in a TRIGGER

This is an undesirable situation where more than one trigger enter into an infinite loop. while creating a trigger we should ensure the such a situation does not exist.

The below example shows how Trigger's can enter into cyclic cascading.

Let's consider we have two tables 'abc' and 'xyz'. Two triggers are created.

1) The INSERT Trigger, triggerA on table 'abc' issues an UPDATE on table 'xyz'.
2) The UPDATE Trigger, triggerB on table 'xyz' issues an INSERT on table 'abc'.

In such a situation, when there is a row inserted in table 'abc', triggerA fires and will update table 'xyz'.
When the table 'xyz' is updated, triggerB fires and will insert a row in table 'abc'.
This cyclic situation continues and will enter into a infinite loop, which will crash the database.