Structured Query Language (SQL).

Chapter – 1
Introduction to Structured Query Language (SQL)

Introduction of SQL:

SQL stands for „Structured Query Language" and can be pronounced as „SQL" or „SEQUEL" (Structured English Query Language). It is a special purpose language used to define access and manipulate data in RDBMS. According to ANSI (American National Standards Institute), it is the standard language for relational database.

- SQL provides a set of statements for storing and retrieving data from the relational database.
- The original version called SEQUEL was designed by IBM in mid-1970. SQL was first introduced as a commercial database system in 1979 by Oracle Corporation.
- Now-a-days, almost all MRDBMS (Modern Relational Database Management Systems) like Microsoft SQL-Server, Microsoft Access, Oracle, BD2, Sybase, MySQL and Informix use SQL as standard database language.
- For example Microsoft SQL-Server specific version of the SQL is called T-SQL, Oracle version of SQL is called PL/SQL, and Microsoft Access version is SQL is called JET-SQL etc.
- SQL is different from other programming languages like C++, Visual-Basic etc.
- It is a Non-Procedural, English-like language that processes data in groups of records rather than one record at a time.
- This Non-Procedural nature of SQL makes it easier to access data in application programs.

Types of SQL: SQL is broadly classified into two types.

1. Interactive SQL.
2. Embedded SQL.

1. **Interactive SQL:** "Interactive SQL is used for directly access the data from the database". Whereas the output of operation is used for human consumptions. Once a command is specified is "executed and output is immediately view by the users".

2. **Embedded SQL:** "In the case of Embedded SQL commands are SQL Commands that are written in some another languages". Such as COBAL, ALP (Assembly Level Programming Language). This makes the programmer very fast and powerful manner.

Advantages of SQL:

- SQL is a high level language that provides a greater degree of „abstraction".
- SQL deals with number of database management system at a time.
- SQL is an English like computer language which makes interaction between user and database very simply.
- SQL language which being simple and easy to learn can handle complex situation.
- SQL operations are performed at a set level. One select statement can retrieve multiple rows.

What SQL can Do? :

- SQL can retrieve data from a database.
- SQL can execute queries against a database.
- SQL can insert, update and delete records in a database.
- SQL can create new tables in a database.
- SQL can create views in a database.
- SQL cam set permissions on tables, procedures and views.

***** (Q) Define SQL*PLUS? And illustrate differences with SQL?**

SQL*PLUS is a software product from Oracle Corporation that allows users to interactively use the SQL commands, produce formatted reports and support written command-procedures to access Oracle Database. Though SQL*PLUS, a user can Do:

- Enter, Edit, Store, Retrieve and Run SQL Commands.
- Format, Perform calculations, Print Query results in the form of Reports.
- Access and Copy data between SQL Database.
- Send messages to and accepts responses from an End-Users.

Difference between SQL and SQL*Plus:

SQL	SQL*PLUS
SQL is a special-purpose language used to define, access and manipulate data in RDBMS like Oracle	SQL*PLUS is a command line tool that allows user to type SQL commands to be executed directly against an Oracle Database.
SQL provides set of commands used to communicate with the Database to perform specific tasks.	SQL*PLUS is the environment where we can actually type wer commands to perform specific tasks.
SQL commands can be used to create tables objects like Create, Alter, drop etc.	SQL*PLUS commands can be used to describe the structure of Database objects. For example: DESCRIBE/DESCRIPTION is DESC command.
SQL commands cannot be abbreviated.	In SQL*PLUS commands can be abbreviated. Like the commands EDIT can be ED.
Every SQL commands must be ended with semicolon (;)	In SQL*PLUS semicolon (;) is optional.

***** (Q) Define various datatypes in SQL?**

SQL Datatypes: Most of programming languages require programmed declaration by using datatypes. The most database system required the user to specific the type of each data fields. The available datatypes varying from one programming language to other languages. The SQL supports the following scalar data types.

SQL Datatypes

Numeric Datatype

- Bit (size)
- Tinyint (size)
- Boolean
- Smallint (size)
- Integer (size)
- Decimal (size, d)
- Float (d, p)
- Number (p, q)
- Long
- Raw (n)

String/Character Datatype

- Char (n)
- Varchar2 (n)
- Text

Date & Time Datatype

- Date
- Time
- Datetime
- Year
- Timestamp

LOB Datatype

- BLOB
- CLOB
- BFILE

I. Numeric Datatype

1. **Bit (size) Datatype:** A bit-value type, the number of bits per value is specified in size. The size parameter can hold a value from 1 to 64. The default value for size is 1.

2. **Tinyint (size) Datatype:** A very small integer, signed range is from -128 to 127. Unsigned range is from 0 to 255. The size parameter specifies the maximum display width (which is 255).

3. **Boolean Datatype:** Zero is considered as false, nonzero values are considered as true.

4. **Smallint (size) Datatype:** A small integer, signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The size parameter specifies the maximum display width (which is 255).

5. **Integer (size) Datatype:** An integer, signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255).

6. **Decimal (size, d) Datatype:** An exact fixed-point number, the total number of digits is specified in size*.* The number of digits after the decimal point is specified in the „d" parameter. The maximum number for size is 65. The maximum number for „d" is 30. The default value for size is 10. The default value for „d" is 0.

7. **Float (d, p) Datatype:** A floating point number, the total number of digits is specified in size. The number of digits after the decimal point is specified in the „d" parameter. A floating point number. SQL uses the „p" value to determine whether to use FLOAT or DOUBLE for the resulting data type. If „p" is from 0 to 24, the data type becomes FLOAT(). If „p" is from 25 to 53, the data type becomes DOUBLE().

8. **Number (p, q) Datatype:** Fixed precision and scale numbers. The „p" parameter indicates the maximum total number of digits that can be stored „p" must be a value from 1 to 38. Default is 18. The „q" parameter indicates the maximum number of digits stored to the right of the decimal point. „q" must

be a value from 0 to p. Default value is 0.

9. **Long Datatype:** Allows whole numbers between -2,147,483,648 and 2,147,483,647.

10. **Raw (n) Datatype:** Variable length raw binary data. Maximum of length „n" is 255 bytes.

II. String/Character Datatype

1. **Char (n) Datatype:** A Fixed width character string (can contain letters and special characters). The size parameter specifies the column length in characters - can be from 0 to 255. Default is 1.

2. **Varchar2 (n) Datatype:** A varying length string (can contain letters and special characters). The size parameter specifies the maximum column length in characters - can be from 0 to 65535.

3. **Text Datatype:** Variable width character string. Holds a string with a maximum length of 65,535 bytes or 2GB of text data.

III. Date & Time Datatype

1. **Date Datatype:** A date, Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999- 12-31'. Example: 2020-11-12 or 2020-Nov-12.

2. **Time Datatype:** A time, Format: HH:MM:SS. The supported range is from '00:00:00' to '23:59:59'. Example: 22:45:50.

3. **Datetime Datatype:** A date and time combination. Format: YYYY-MM-DD HH:MM:SS. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time. Example: 2020-Nov-12 22:46:32.

4. **Year Datatype:** A year values allowed in four-digit format: 1000 to 9999. Example: 2020.

5. **Timestamp Datatype:** A timestamp values are stored as the number of seconds since the Universal Time Coordinate (UTC) Format: YYYY-MM-DD HH:MM:SS. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP.

IV. LOB Datatype (Large Object)

Multimedia data like sound, picture and video need more storage space. The LOB datatype such as BLOC, CLOB and BFILE allows us to store large block of data.

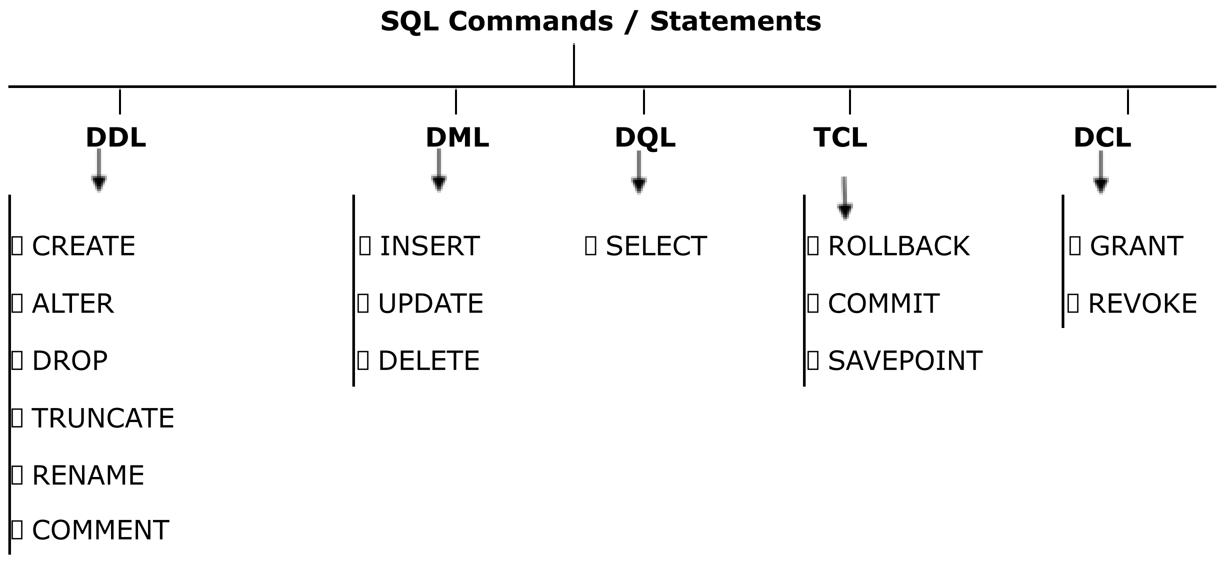
1. **BLOB (Binary Large Object) Datatype:** The BLOB datatype stores unstructured binary data in the database. BLOC can store up to 4GB of binary data.

2. **CLOB (Character Large Object) Datatype:** The CLOB datatype stores can store up to 4GB of character data in the database.

3. **BFILE (Binary File) Datatype:** The BFILE datatype stores unstructured binary data in operating system files outside the database. A BFILE can store up to 4GB of data.

******* (Q) Explain the various types of SQL Commands?**

Structure Query Language (SQL) is a database query language used for storing and managing data in Relational DBMS. SQL was the first commercial language introduced for Dr. E.F Codd's Relational model of database. Today almost all RDBMS (MySql, Oracle, Infomix and Sybase, MS-Access) use SQL as the standard database query language. SQL statements are divided into the following categories:



(I). **Data Definition Language (DDL):** DDL commands are used to "Define the Database Structure" by "Creating, Altering, Dropping, Truncating, Renaming and Commented Table". All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

- **CREATE**: Create is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
- **ALTER**: Alter is used to altering the structure of the database.
- **DROP:** Drop is used to delete objects from the database.
- **TRUNCATE:** Truncate is used to remove all records from a table, including all spaces allocated for the records are removed.
- **RENAME**: Rename is used to rename a table existing in the database.
- **COMMENT:** Comment is used to add comments to the data dictionary.

(II). **Data Manipulation Language (DML):** DML commands allowed the user to "Manipulate data". Manipulate data refers to "Inserting, Updating, Deleting, Merging and calling of data". DML commands are not auto-committed. It means changes are not permanent to database, they can be rolled back.

- **INSERT**: Insert statement is used to insert new records into a table.
- **UPDATE**: Update is used to update existing data within a table.
- **DELETE**: Delete is used to delete records from a database table.

(III). **Data Query Language (DQL):** DQL commands is used to "Fetch or access data from tables based on conditions" that we can easily apply.

- **SELECT**: Select is used to retrieve records from one or more tables from the database.

(IV). **Transaction Control Language (TCL):** TCL commands are to keep a check on other commands and their affect on the database. These commands can annul changes made by other commands by rolling the data back to its original state. It can also make any temporary change permanent.

- **ROLLBACK:** Rollback is used to a transaction in case of any error occurs.
- **COMMIT:** Commit is used to permanently save the transaction into a database.
- **SAVEPOINT:** Savepoint is used to temporally save the transaction into a database.

(V). **Data Control Language (DCL):** DCL is the commands to grant and take back authority from any database user. The Database Administrator (DBA) has the power to give and take the privileges to a specific user, thus giving or revoking access to the data. The task of the DCL is to prevent unauthorized access to data.

- **GRANT**: Grant is used to gives user"s access privileges to database.
- **REVOKE:** Revoke is used to withdraw user"s access privileges given by using the GRANT command.

(I) Data Definition Language Commands (DDL)

1. **Create command:** Create is used to create the database or its objects (like table, index, function, views, store procedure and triggers).

(a). **Create New table:** To create a new table into the database.

Syntax: SQL> create table <Table_Name>

```
( Column_Name1 datatype (size),
  Column_Name2 datatype (size),
  - - - - - - - - - - -
  - - - - - - - - - - -
  Column_Name n datatype (size));
```

```
SQL> create table STUDENT (
        emp_Id number(3),
          emp_Name varchar2(15),
          experience number(2),
          salary number(5));
Table created.
```

```
SQL> desc STUDENT;

    Name            Null?    Type
    ------------ ------ ----------
    EMP_ID                 NUMBER(3)
    EMP_NAME               VARCHAR2(15)
    EXPERIENCE             NUMBER(2)
    SALARY                 NUMBER(5)
```

(b). **Create Table Using another Table:** A copy (duplicate) of an existing table can also be created using CREATE TABLE. The new table gets the same column definitions. All columns or specific columns can be selected. If we create a new table using an existing table, the new table will be filled with the existing values from the old table.

```
Syntax: SQL> create table <New Table_Name> AS
             select (existing Column_Name1, existing Column_Name2 ,…… , existing Column_Name) n,
             from <Existing Table_Name> where <conditions>;
```

```
SQL> create table STUDENT as select emp_id, emp_name, salary from STUDENT;

Table created.
```

```
SQL> desc STUDENT;          <Existing Table>

    Name            Null?     Type
    ------------ ------ ----------
    EMP_ID                  NUMBER(3)
    EMP_NAME                VARCHAR2(15)
    EXPERIENCE              NUMBER(2)
    SALARY                  NUMBER(5)
```

```
SQL> desc STUDENT;          <New Table>

    Name            Null?     Type
    ------------ ------ ----------
    EMP_ID                  NUMBER(3)
    EMP_NAME                VARCHAR2(15)
    SALARY                  NUMBER(5)
```

2. **Alter command:** Alter is used to altering the structure of the database. Alter statements can support any adding new columns from existing tables or any of changes to the size or datatype of an existing column in database or dropped existing columns or rename from existing table to new table in the database.

(a). **Adding New Columns:** To add a New Columns from existing table.

```
Syntax: SQL> alter table <existing Table_Name> ADD
                    ( new Column_Name1 datatype (size),
                      new Column_Name2 datatype (size),
                        - - - - - - - - - - -
                        - - - - - - - - - - -
                      new Column_Name n datatype (size));
```

```
SQL> alter table STUDENT add (Contact_Number number(12), Address varchar2(30));

Table altered.
```

SQL> desc STUDENT;

```
    Name              Null?    Type
                      ----------------------------
  EMP_ID                       NUMBER(3)
  EMP_NAME                     VARCHAR2(15)
  EXPERIENCE                   NUMBER(2)
  SALARY                       NUMBER(5)
  CONTACT_NUMBER               NUMBER(10)
  ADDRESS                      VARCHAR2(15)
```

(b). **Modify the Existing Column values:** To modify the existing table values like datatype and size.

Syntax: SQL> alter table <Table_Name> MODIFY

```
                    ( existing Column_Name1 datatype (size),
                      existing Column_Name2 datatype (size),
                        - - - - - - - - - - -
                        - - - - - - - - - - -
                      existing Column_Name n datatype (size));
```

SQL> alter table STUDENT modify (address char(20)); Table

altered.

SQL> desc STUDENT;

```
 Name              Null?    Type
 --------------- ------ ---------
 EMP_ID                     NUMBER(3)
 EMP_NAME                   VARCHAR2(15)
 EXPERIENCE                 NUMBER(2)
 SALARY                     NUMBER(5)
 CONTACT_NUMBER             NUMBER(10)
 ADDRESS                    CHAR(20)
```

(c). **Drop existing column:** To delete an existing column from existing table.

Syntax: SQL> alter table <Table_Name> DROP COLUMN <Existing Column_Name>;

SQL> alter table STUDENT drop column contact_number;
Table altered.
SQL> desc STUDENT;

```
 Name              Null?    Type
 --------------- ------ ---------
 EMP_ID                     NUMBER(3)
 EMP_NAME                   VARCHAR2(15)
 EXPERIENCE                 NUMBER(2)
 SALARY                     NUMBER(5)
 ADDRESS                    CHAR(20)
```

(d). **Rename Table:** To rename the table from existing table to new table.

Syntax: SQL> alter table <existing Table_Name> RENAME TO < new Table_Name>;

SQL> alter table STUDENT rename to student123;

SQL> select * from STUDENT;

ERROR at line 1: ORA-00942: table or view does not exist

SQL> select * from student123;

```
EMP_ID      EMP_NAME            CONTACT_NUMBER
--------------------------      ---------------
  101     Mohammad student123     9491202987
  102     Shaik Abu               9442011432
```

3. **Drop command:** This command is use "Drop the Structure / Table (or) Delete the Objects / Tables from the Database permanently". When using the drop table statement, we cannot „rollback".

Syntax: SQL> drop table <Table_Name>;

SQL> drop table STUDENT; Table
dropped.

4. **Truncate command:** Truncate is used to remove all records from a table, including all spaces allocated for the records are removed by the table, but table structure will be alive. When using the truncate table statement, we can"t „rollback".

Syntax: SQL> truncate table <Table_Name>;

SQL> truncate table STUDENT;

SQL> select * from STUDENT;

5. **Comments in SQL:** There are two ways in which we can comment in SQL, i.e. either the Single-Line Comments or the Multi-Line Comments.

(a). **Single-Line Comments:** The single line comment starts with two hyphens (--), till the end of a single line will be ignored by the compiler.

SQL> -- Display all the records from student123 table.

SQL> Select * from student123;

(b). **Multi-Line Comments:** The Multi-line comments start with **/*** and end with ***/**. So, any text mentioned between /* and */ will be ignored by the compiler.

```
SQL> /* Display all records which have
  2. from the student123 table */
  3. Select * from student123;
```

(II) Data Manipulation Language (DML)

1. **Insert command:** Insert statement is used to insert new records into a table. There are inserting three methods as following.

(a). Inserting dynamically.
(b). Inserting only with specify Column_Names.
(c). Inserting without any specify Column_Names.

(a). **Inserting dynamically:** In this method values can be inserting through reference address of the Column_Names, and in this method we can also inserted multiple rows without repeat syntax, Just use „/" symbol.

Syntax: SQL> insert into <Table_Name> values

```
                    („& Column_Name1", „& Column_Name2", „& Column_Name3",
                      - - - - - - - - - - -
                      - - - - - - - - - - -
                     „& Column_Name n");
```

```
SQL> insert into STUDENT values („&emp_id", „&emp_name", „&experience", „&salary",
     „&contact_number", „&address");

    Enter value for emp_id: 101
    Enter value for emp_name: Mohammad
    Enter value for experience: 12
    Enter value for salary: 36500
    Enter value for contact_number: 9491202987
    Enter value for address: Tirupati

1 row created.

SQL> /
    Enter value for emp_id: 102
    Enter value for emp_name:
    student123 Enter value for
    experience: 11 Enter value for
    salary: 34650
    Enter value for contact_number: 9442011432
    Enter value for address: Hyderabad

1 row created.

SQL> select * from STUDENT;

   EMP_ID   EMP_NAME    EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
     101    Mohammad        12        36500     9491202987     Tirupati
     102    student123      11        34650     9442011432     Hyderabad

2 rows selected.
```

(b). **Inserting only with specify Column_Names:** In this method values can be inserting only specify columns rather than all.

```
Syntax: SQL> insert into <Table_Name>
          (Column_Name1, Column_Name3, Column_Name6,                    „& Column_Name n“)
          values („value of Column_Name1“, „value of Column_Name3“,       „value of Column_Name n“);
```

```
SQL> insert into STUDENT (emp_id, emp_name, experience, address)
                values (103', 'Shaik', '08', 'Rompicherla');

1 row created.

SQL> insert into STUDENT (emp_id, emp_name, experience, address)
                values ('104', 'Abu', '06', 'Tirupati');

1 row created.

SQL> select * from STUDENT;

     EMP_ID    EMP_NAME     EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
       101     Mohammad         12        36500      9491202987    Tirupati
       102     student123       11        34650      9442011432    Hyderabad
       103     Shaik             8                                 Rompicherla
       104     Abu               6                                 Tirupati

4 rows selected.
```

(c). **Inserting without any specify Column_Names:** In this method values can be inserting without any specify Column_Names into the table.

```
Syntax: SQL> insert into <Table_Name>
          values („value of Column_Name1“, „value of Column_Name2“,       „value of Column_Name n“);
```

```
SQL> insert into STUDENT values ('105', 'Muneer', '5', '12000', '9491202169', 'Mumbai');

1 row created.

SQL> insert into STUDENT values ('106', 'Mujeeb', '2', '8500', '9441201342', 'Chennai');

1 row created.
SQL> select * from STUDENT;

     EMP_ID    EMP_NAME     EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
       101     Mohammad         12        36500      9491202987    Tirupati
       102     student123       11        34650      9442011432    Hyderabad
       103     Shaik             8                                 Rompicherla
       104     Abu               6                                 Tirupati
       105     Muneer            5        12000      9491202169    Mumbai
       106     Mujeeb            2         8500      9441201342    Chennai
6 rows selected.
```

2. **Update command:** The update statement is used to modify the data that is already in the database. The condition in the WHERE clause decides that which row is to be updated. There are three situations to use updated command.

(a). Update single records. (Null values)
(b). Update multiple records. (Replace values)
(c). Update all records at a time. (If omitted where clause)

```
Syntax: update <Table_Name> set <Column_Name1> = <Value 1>,
                                <Column_Name2> = <Value 2>,
                                    - - - - - - - - - - -
                                    - - - - - - - - - - -
                                 <Column_Name n> = <Value n>
                   [where <condition>];
```

(a). **Update Single record (Null values):** if we inserting new value at the place of missing field one by one.

Syntax: SQL> update <Table_Name> set <Column_Name> = <value> [where <condition>];

SQL> update STUDENT set salary = '26580' where emp_id = '103';

1 row updated.
SQL> select * from STUDENT;

```
EMP_ID   EMP_NAME    EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
  101    Mohammad        12        36500     9491202987     Tirupati
  102    student123      11        34650     9442011432     Hyderabad
  103    Shaik            8        26580                    Rompicherla
  104    Abu              6                                 Tirupati
  105    Muneer           5        12000     9491202169     Mumbai
  106    Mujeeb           2         8500     9441201342     Chennai
```
6 rows selected.
Update multiple records (Replace): If we want to update multiple columns, we should separate each field assigned with a comma. And also replace the values from existing values with new values.

```
Syntax: update <Table_Name> set <Column_Name1> = <Value 1>,
                                <Column_Name2> = <Value 2>,
                                    - - - - - - - - - - -
                                    - - - - - - - - - - -
                                 <Column_Name n> = <Value n>
                   [where <condition>];
```

SQL> update STUDENT set emp_name = 'Masthan', address = 'Piler' where emp_id = '103';

SQL> select * from STUDENT;

```
EMP_ID   EMP_NAME    EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
  101    Mohammad        12        36500     9491202987     Tirupati
  102    student123      11        34650     9442011432     Hyderabad
```

```
   103     Masthan            8        26580                         Piler
   104     Abu                6                                      Tirupati
   105     Muneer             5        12000      9491202169         Mumbai
   106     Mujeeb             2         8500      9441201342         Chennai
```

6 rows selected.

(b). **Update all records at a time:** If we want to update all row from a table, then we don't need to use the WHERE clause.

Syntax: SQL> update <Table_Name> set Column_Name = Value;

SQL> update STUDENT set emp_name = „Naveena“;

SQL> select * from STUDENT;

```
EMP_ID   EMP_NAME    EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
   101     naveena           12     36500      9491202987     Tirupati
   102     naveena           11     34650      9442011432     Hyderabad
   103     naveena            8     26580                     Piler
   104     naveena            6                               Tirupati
   105     naveena            5     12000      9491202169     Mumbai
   106     naveena            2      8500      9441201342     Chennai
```
6 rows selected.

3. **Delete command:** The delete statement is used to delete rows from a table. Generally, delete statement removes one or more records form a table. There are three situations to use delete command.

(a). Delete single row.
(b). Delete multiple rows.
(c). Delete all rows.

Syntax: SQL> delete from <Table_Name> [where <condition>];

(a). **Delete single row:** This command is used to remove complete one single record.

Syntax: SQL> delete from <Table_Name> [where <condition>];

SQL> delete from STUDENT where emp_name = „Masthan“;

SQL> select * from STUDENT;

```
EMP_ID   EMP_NAME    EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
   101     Mohammad          12     36500      9491202987     Tirupati
   102     student123        11     34650      9442011432     Hyderabad
   104     Abu                6                               Tirupati
   105     Muneer             5     12000      9491202169     Mumbai
   106     Mujeeb             2      8500      9441201342     Chennai
```

5 rows selected.

(b). **Delete multiple rows:** This command used to remove all records when the condition is matched.

Syntax: SQL> delete from <Table_Name> [where <condition>];

SQL> delete from STUDENT where address = „Tirupati";

2 rows deleted

SQL> select * from STUDENT;

```
EMP_ID   EMP_NAME     EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
  102    student123       11        34650     9442011432     Hyderabad
  105    Muneer            5        12000     9491202169     Mumbai
  106    Mujeeb            2         8500     9441201342     Chennai
```

3 rows selected.

(c). **Delete all rows:** Delete the entire rows from the table. After this, no records left to display. The table will become empty.

Syntax: SQL> delete from <Table_Name>; (or) delete * from <Table_Name>;

SQL> delete from STUDENT;

SQL> select * from STUDENT;

No rows selected

(III) Data Query Language (DQL)

1. **Select command:** The Select statement is used to "fetch or access or retrieve records from one or more tables from the database on conditions". The returns data is stored in table. This is also three ways to fetch the data into the table.

(a). Fetch single / particular column.
(b). Fetch multiple / selective columns.
(c). Fetch all / total columns.

```
Syntax: SQL> select Column_Name1, Column_Name2,
                    - - - - - - - - - - -
                    - - - - - - - - - - -
                    Column_Name n from <Table_Name> [where <condition>];
```

(a). **Fetch single / particular column:** The select statement is used to fetch any particular column into the table.

Syntax: SQL> select Column_Name from <Table_Name>;

SQL> select emp_name from STUDENT;

```
EMP_NAME
---------------
Mohammad
```

```
        student123
        Masthan Abu
        Muneer Mujeeb

    6 rows selected.
```

SQL> select emp_name from STUDENT where emp_name = 'student123';

```
        EMP_NAME
        ---------------
        student123
```

(b). **Fetch multiple / selective columns:** The select statement is used to fetch multiple columns into the table.

Syntax: SQL> select Column_Name1, Column_Name2, ……., Column_Name n from <Table_Name>;

SQL> select emp_name, contact_number from STUDENT;

```
                    EMP_NAME      CONTACT_NUMBER
                    Mohammad         9491202987
                    student123       9442011432
                    Masthan
                    Abu
                    Muneer           9491202169
                    Mujeeb           9441201342
        6 rows selected.
```

(c). **Fetch all / total columns:** The select statement is used to fetch all fields /columns into the table.

Syntax: SQL> Select * from <Table_Name>;

SQL> select * from STUDENT;

```
  EMP_ID   EMP_NAME     EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
    101    Mohammad         12        36500     9491202987     Tirupati
    102    student123       11        34650     9442011432     Hyderabad
    103    Masthan           8        26580                    Piler
    104    Abu               6                                 Tirupati
    105    Muneer            5        12000     9491202169     Mumbai
    106    Mujeeb            2         8500     9441201342     Chennai

6 rows selected.
```

(IV) Transaction Control Language (TCL)

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only. These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping tables.

1. **Rollback command:** Rollback is used to a transaction in case of any error occurs.

Syntax: SQL> rollback;

SQL> delete from STUDENT where emp_id = 104; 1

row deleted.

SQL> select * from STUDENT;

```
EMP_ID   EMP_NAME    EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
  101    Mohammad        12        36500     9491202987     Tirupati
  102    student123      11        34650     9442011432     Hyderabad
  103    Masthan          8        26580                    Piler
  105    Muneer           5        12000     9491202169     Mumbai
  106    Mujeeb           2         8500     9441201342     Chennai
```

5 rows selected.

SQL> rollback;

Rollback complete.

SQL> select * from STUDENT;

```
EMP_ID   EMP_NAME    EXPERIENCE   SALARY   CONTACT_NUMBER   ADDRESS
  101    Mohammad        12        36500     9491202987     Tirupati
  102    student123      11        34650     9442011432     Hyderabad
  103    Masthan          8        26580                    Piler
  104    Abu              6                                 Tirupati
  105    Muneer           5        12000     9491202169     Mumbai
  106    Mujeeb           2         8500     9441201342     Chennai
```

6 rows selected.

2. **Commit command:** Commit is used to permanently save the transaction into a database.

Syntax: SQL> commit;

SQL> delete from STUDENT where emp_id = 104; 1

row deleted.

SQL> select * from STUDENT;

```
 EMP_ID   EMP_NAME    EXPERIENCE  SALARY  CONTACT_NUMBER  ADDRESS
   101    Mohammad        12      36500     9491202987    Tirupati
   102    student123      11      34650     9442011432    Hyderabad
   103    Masthan          8      26580                   Piler
   105    Muneer           5      12000     9491202169    Mumbai
   106    Mujeeb           2       8500     9441201342    Chennai
```

5 rows selected.

SQL> commit;

Commit complete.

SQL> rollback;

Rollback complete.

SQL> select * from STUDENT;

```
 EMP_ID   EMP_NAME    EXPERIENCE  SALARY  CONTACT_NUMBER  ADDRESS
   101    Mohammad        12      36500     9491202987    Tirupati
   102    student123      11      34650     9442011432    Hyderabad
   103    Masthan          8      26580                   Piler
   105    Muneer           5      12000     9491202169    Mumbai
   106    Mujeeb           2       8500     9441201342    Chennai
```

5 rows selected.

3. **Savepoint command:** Savepoint creates points within the groups of transactions in which to rollback. A savepoint is a point in a transaction in which we can roll the transaction back to a certain point without rolling back the entire transaction.

Syntax: SQL> savepoint <savepoint name>;

This command is used only in the creation of savepoint among all the transactions. In general rollback is used to undo a group of transactions.

Syntax for rolling back to Savepoint command:
SQL> rollback to <savepoint name>;

SQL> savepoint spstudent1231;

Savepoint created.

SQL> delete from STUDENT where emp_id = 105;

SQL> select * from STUDENT;

```
 EMP_ID   EMP_NAME    EXPERIENCE  SALARY  CONTACT_NUMBER  ADDRESS
   101    Mohammad        12      36500     9491202987    Tirupati
   102    student123      11      34650     9442011432    Hyderabad
   103    Masthan          8      26580                   Piler
```

```
     104     Abu              6                                 Tirupati
     106     Mujeeb           2        8500      9441201342     Chennai
```

5 rows selected.

SQL> savepoint spstudent1232;

Savepoint created.

SQL> rollback to spstudent1231;

Rollback complete.

SQL> select * from STUDENT;

```
  EMP_ID   EMP_NAME    EXPERIENCE  SALARY  CONTACT_NUMBER  ADDRESS
     101   Mohammad        12       36500    9491202987    Tirupati
     102   student123      11       34650    9442011432    Hyderabad
     103   Masthan          8       26580                  Piler
     104   Abu              6                              Tirupati
     105   Muneer           5       12000    9491202169    Mumbai
     106   Mujeeb           2        8500    9441201342    Chennai
```
6 rows selected.

Release savepoint command: This command is used to remove a savepoint that we have created.

Syntax: SQL> release savepoint <savepoint name>;

Once a savepoint has been released, we can no longer use the rollback command to undo transactions performed since the last savepoint.

(IV) Data Control Language (DCL)

1. **Grant:** This command is used to "Grant the permissions on one table to other user (or) Gives user"s access privileges to Database". To Grant all permission, we can"t drop the table.

Syntax: Grant [type of privilege] ON [user name]. [Table_Name] to [user name] [with Grant option];

- **Grant all:** This command is used to Grant the permissions on all tables to other user. To Grant all permission, we can"t drop the table.

Syntax: Grant All [type of privilege] ON [user name]. [Table_Name] to [user name] [with Grant option];

2. **Revoke:** This command is used to "Take-out some or all permissions on a table from a user". (or) "Withdraw access privileges given with the GRANT command.

Syntax: Revoke [type of privilege] ON [user name]. [Table_Name] from [user name];

******* (Q). Define Select command / statement / operation in SQL?**

The Select command is a DML command. This command used to query the data in a table. A Query is a question, which helps to fetch the data from the database. Every select statement returns the result in the form of table after the execution. The general syntax for select command:

Syntax: SQL> SELECT [DISTINCT | ALL | *] {Column_Names}

```
FROM [{Table_Name [Alias] | View_Name}]...
[WHERE <condition>]
[GROUP BY <conditional Column_Name(s)>]
[HAVING condition]
[ORDER BY {Column_Name [ASC | DESC]} ...
```

- **SELECT command:** The SELECT clause is mandatory and carries out the relational. Select is identifies the columns in the table.

- **DISTINCT command:** It returns only single copy of each column values.

  Syntax: SQL> select distinct (Column_Name) from <Table_Name>;

  SQL> select distinct (marks) from stud_info;

```
MARKS
----------
   83
   87
   95
   85
   94
   99
   98
   92
8 rows selected.
```

- *** / ALL (Astric) operator:** It indicate all Column_Names in the table. Select * means all the rows, columns including duplicate rows will be display. * is also known as wild card character.

- **FROM command:** The FROM clause is also mandatory. It identifies one or more tables and/or views from which to retrieve the column data displayed in a result table.

- **Table_Name:** It is the name of the table from which data is selected.

  Syntax: Select * from <Table_Name>;

SQL> select * from stud_info;

```
          STUD_ID  STUD_NAME   SUBJECT    MARKS
            101    student123  Computers    98
            102    Shaik       Physics      85
            103    Abu Shaik   Biology      98
            104    Hashim      Zoology      94
            105    naveena     Computers    99
            106    Muneer      Botany       92
            107    Mujeeb      Chemistry    95
            108    Masthan     Maths        95
            109    Ismail      Maths        83
            110    Fayaz       Computers    87
     10 rows selected
```

- **WHERE command:** The WHERE clause is optional and carries out the relational select operation. It specifies used to display only those rows for which the condition is true. We have to specify a condition after the WHERE clause. If the condition is not satisfied then no rows will be display from the table.

  Syntax: SQL> Select (Column_Names) from <Table_Name> [where <condition>];

  SQL> select * from stud_info where marks > 95;

```
          STUD_ID  STUD_NAME   SUBJECT    MARKS
            101    student123  Computers    98
            103    Abu Shaik   Biology      98
            105    naveena     Computers    99
   3 rows selected
```

Use expressions in the WHERE command: Expressions can also be used in the WHERE clause of the select statement.

SQL> select * from STUDENT;

```
EMP_ID  EMP_NAME    EXPERIENCE  SALARY  CONTACT_NUMBER  ADDRESS
 101    Mohammad        12      36500     9491202987    Tirupati
 102    student123      11      34650     9442011432    Hyderabad
 103    Masthan          8      26580     7892021949    Piler
 104    Abu              6      22580     1234567890    Tirupati
 105    Muneer           5      12000     9491202169    Mumbai
 106    Mujeeb           2       8500     9441201342    Chennai

6 rows selected.
```

```
SQL> select emp_name, salary, (salary + salary * 0.1) as Bonus_Salary from STUDENT
     where (salary + salary * 0.1) > 30000;

                    EMP_NAME   SALARY   BONUS_SALARY
                    Mohammad    36500          40150
                    student123  34650          38115
```

- **GROUP BY command:** The GROUP BY clause is optional. It is used with select statement to combine a group of rows based on the values of a particular column or expression. SQL groups the result after it retrieves the rows from a table.

  Syntax: select (Column_Names), <aggregation function>(Column_Name) from <Table_Name>
  GROUP BY <conational Column_Name>;

```
SQL> select subject, avg (marks) from stud_info group by subject;

                        SUBJECT      AVG(MARKS)
                        Physics      85
                        Botany       92
                        Chemistry    95
                        Zoology      94
                        Maths        89
                        Computers    94.66667
                        Biology      98
          7 rows selected
```

- **HAVING command:** The optional HAVING clause. It is used to filter database on the group functions. It is used to specify which groups are to be displayed, that is, restrict the groups that we return on the basis of group function. Simply, conditional retrieval of rows from a grouped result is possible with the HAVING clause.

  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.

  Syntax: select (Column_Names), <aggregation function> (Column_Name) from <Table_Name>
  GROUP BY <conational Column_Name> HAVING <condition>;

```
SQL> select subject, avg (marks) from stud_info group by subject having count (subject) >1;

                        SUBJECT      AVG(MARKS)
                        Maths        89
                        Computers    94.66667
          2 rows selected
```

- **ORDER BY command:** The ORDER BY clause is optional. It is used to sort the rows in a table be default sorting order is „ascending" order. We can use the keyword „ASC". The keyword „DESC" is used to sort in descending order. The ORDER BY clause must be the last clause in a select statement.

  Syntax: select (Column_Names) from <Table_Name> ORDER BY (Column_Name) ASC | DESC;

  SQL> select stud_name, subject, marks from stud_info order by (marks) desc;

STUD_NAME	SUBJECT	MARKS
naveena	Computers	99
Abu Shaik	Biology	98
student123	Computers	98
Masthan	Maths	95
Mujeeb	Chemistry	95
Hashim	Zoology	94
Muneer	Botany	92
Fayaz	Computers	87
Shaik	Physics	85
Ismail	Maths	83

10 rows selected.

***** (Q) Explain Aggregate Functions or Grouped Functions in SQL?**

Aggregate functions are the DDL statements. Aggregate functions also called Grouped functions are built-in SQL functions that operate on groups of rows and return one value for the entire group or table. Aggregate functions are used to produce summarized results. The SQL aggregate functions are:

1. COUNT()
2. COUNT(*)
3. SUM()
4. AVG()
5. MAX()
6. MIN()
7. STDDEV()
8. VARIANCE()

Consider the example table of Stud_Info:

SQL> select * from stud_info;

STUD_ID	STUD_NAME	SUBJECT	MARKS
101	student123	Computers	98
102	Shaik	Physics	85
103	Abu Shaik	Biology	98
104	Hashim	Zoology	94
105	naveena	Computers	99
106	Muneer	Botany	92
107	Mujeeb	Chemistry	95
108	Masthan	Maths	95
109	Ismail	Maths	83
110	Fayaz	Computers	87

```
                    10 rows selected
```

1. **COUNT():** COUNT function is used to Count the number of rows in a database table.

Syntax: SQL> select COUNT (DISTINCT Column_Name) from <Table_Name> [where <condition>];

Example 1: Find the number of subjects has in stud_info table.

SQL> select count (subject) from stud_info;

```
        COUNT (SUBJECT)
        -----------------
                10
```

Example 2: Find the only number of computers subject have in stud_info table.

SQL> select count (subject) as computer from stud_info where subject = 'Computers';

```
        COMPUTER
        ----------------
             3
```

Example 3: Find the number of marks which has non redundant in stud_info table;

SQL> select count (distinct marks) as marks from stud_info;

```
        MARKS
        ----------
           8
```

2. **COUNT(*):** COUNT(*) function is used to count the total all number of records (repeated/duplicate) in a database table.

Syntax: SQL> select COUNT(*) from <Table_Name>;

Example: Find the total number of records has in stud_info table;

SQL> select count (*) from stud_info;

```
        COUNT(*)
        ----------
          10
```

3. **SUM():** SUM() function is used to get the sum of the values (total values) of from column name.

Syntax: SQL> select sum (Column_Name) from <Table_Name>;

Example: Compute the total marks for all students in stud_info table.

SQL> select sum (marks) as Totla_Marks from stud_info;

```
        TOTLA_MARKS
```

```
-----------------
       926
```

4. **AVG():** AVG() function is used to get the average of the values of from column name.

Syntax: SQL> select avg (Column_Name) from <Table_Name>;

Example: Compute the average marks about all students in stud_info table.

SQL> select avg (marks) as Average_Marks from stud_info;

```
AVERAGE_MARKS
--------------------
       92.6
```

5. **MAX():** MAX() function is used to find the maximum value (highest) of from column name.

Syntax: SQL> select max (Column_Name) from <Table_Name>;

Example: Compute the highest marks about all students in stud_info table.
SQL> select max (marks) as Maximum_Marks from stud_info;

```
MAXIMUM_MARKS
--------------------
        99
```

6. **MIN():** MIN() function is used to find the minimum value (least) of from column name.

Syntax: SQL> select min (Column_Name) from <Table_Name>;

Example: Compute the lowest marks about all students in stud_info table.

SQL> select min (marks) as Minimum_Marks from stud_info;

```
MINIMUM_MARKS
--------------------
      83
```

7. **STDDEV():** This function is used to compute the statistical standard deviation of the column name.

Syntax: SQL> select stddev (Column_Name) from <Table_Name>; **Example:**

Compute the standard deviation of student marks in stud_info table. **SQL>**

select stddev (marks) as Standard_Deviation from stud_info;

```
STANDARD_DEVIATION
---------------------------
   5.71936282
```

8. **VARIANCE ():** This function is used to compute the statistical variance of the column name.

Syntax: SQL> select variance (Column_Name) from <Table_Name>;

Example: Compute the variance of student marks in stud_info table.

SQL> select variance (marks) as Variance from stud_info;

```
VARIANCE
-------------
32.7111111
```

(Q) Explain the SQL Operators?

"Operators are a symbol, which represent some particular actions. Operator operates on operands". There are different types of operators used in SQL, namely Arithmetic Operators, Comparison Operators, Logical Operators and Special Operators. These operators are used mainly in the WHERE and HAVING clause to filter the data to be selected.

1. **Arithmetic Operators:** Arithmetic operators are used in SQL expressions to addition, subtraction, multiplication, and division data values. The result of this expression is a numeric value.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

2. **Comparison Operators:** Comparison operators are used to compare the column data with specified values in a condition. Comparison operators are also used along with the SELECT statement to filter data based on specific conditions. The result of a comparison is True, False or Unknown. The below table describes each comparison operator.

Operator	Description
=	Equality
!= or <>	Not Equal to
>	Greater than
<	Less than
>=	Greater than Equal
<=	Less than Equal

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

Operator	Description
AND	Return True; If Both conditions are True; otherwise return False.
OR	Return True; If Either / at least one of the conditions is True; otherwise return False
NOT IN	Return True; If the condition is False; Otherwise return False.

1. **Special Operator:** There are some special operators available in SQL which are uses to enhance

the search capabilities of a SQL query. There are LIKE, IN and BETWEEN operators.

Operator	Description
LIKE	Matching a pattern from a column. Column value is similar to specified characters.
IN	Checking a value in a set. Column value is equal to any one of a specified set of values.
BETWEEN	Checking a value within a range. Column value is between two values, including the end values specified in the range.

1. Find the all student names from „stud_info" table, whose names begins with „S" only?

SQL> select stud_name from stud_info where stud_name LIKE 'S%';

```
STUD_NAME
---------------
Shaik
naveena
```

2. Find the all student names from „stud_info" table, whose names begins with „S" and end with „d" only?

SQL> select stud_name from stud_info where stud_name LIKE 'S%_d%';

```
STUD_NAME
------------
naveena
```

3. Find the all student names from „stud_info" table, whose names end with „k" only?

SQL> select stud_name, subject from stud_info where stud_name LIKE '_%k';

```
STUD_NAME
------------
Shaik
Abu Shaik
```

4. Find the name of the student and subject from „stud_info" table, who elective only computers and biology subjects?

SQL> select stud_name, subject from stud_info where subject IN („Computers", 'Biology');

```
STUD_NAME      SUBJECT
student123     Computers
Abu Shaik      Biology
naveena        Computers
Fayaz          Computers
```

5. Find the all student names, subject and their marks from „stud_info" table, which are secure marks between 90 to 98 marks only?

SQL> select stud_name, subject, marks from stud_info where marks between 90 and 98;

```
STUD_NAME      SUBJECT    Marks
student123     Computers    98
```

Abu Shaik	Biology	98
Hashim	Zoology	94
Muneer	Botany	92
Mujeeb	Chemistry	95
Masthan	Maths	95

******* (Q) Explain various Integrity / Imposition Constraints in SQL?**

SQL uses integrity constraints to prevent invalid data entry into the tables of the database. Constraints are a part of the table definition that is used to limit the values entered into its columns. Constraints are basically used to impose rules on the table, whenever a row is inserted, updated or deleted from the table.

The constraints clause is used to define the integrity constraints in “Create Table or Alter Table” statements. The constraints can be defined in two ways:

1. Column-Level Constraints.
2. Table-Level Constraints.

1. **Column Level Constrains:** The Constraints can be specified immediately after the column definition. This is called Column-Level constraints.

General Syntax of Column Level Constraints:

Syntax: create table <table name> (column name 1 data type (size) constrain <constrain name> primary key, column name 2 data type(size) constrain <constrain name> references referenced_table [(primary_column_name of referenced table)], column name 3 data type(size) constrain <constrain name> check (<condition>), column name 4 datatype (size) NOT NULL);

2. **Table Level Constrains:** The Constraints can be specified after all the columns are defined. This is called Table-Level Constraints. Therefore the table level constraints are defined at the end of the create table statement which can impose rules on any columns in the table. This syntax can define any type of constraints except NOT NULL.

General Syntax of Table Level Constraints:

Syntax: create table <table name> (column name 1 data type(size), column name 2 data type (size),column name 3 data type(size),….column name n data type(size), constrain<constrain name> primary key (column name 1) constrain <constrain name> foreign key(foreign column name) references referenced_table [(primary _column _name of referenced_table)], constrain <constrain name> check (<condition>));

Types of Integrity Constraints:

1. NOT NULL
2. Unique
3. Primary Key
4. Foreign / Reference Key
5. Check
6. Default

1. **NOT NULL:** The NOT NULL constraint enforces a column to always contain a value. By default, a table column can hold NULL values. The NOT NULL constraint enforces a column to not accept NULL values. This constraint is placed immediately after the datatype of a column.

Example: SQL> create table student123 (Emp_Id Number (5) **NOT NULL,**
Emp_Name Varchar2 (30) **NOT NULL,**
Address Varcahr2 (40),
Phone_No Number (10) **NOT NULL**);
SQL> insert into student123 values („& Emp_Id", „& Emp_Name", „& Address", „& Phone_No"); Enter value for Emp_Id:
Enter value for Emp_Name: Shaik Mohammad
student123 Enter value for Address:
Enter value for Phone_No: 9491202987

In the above example it is clear that when we try to insert a null value into „Emp_Id" we get error message. But „Address" have empty value it"s not any error, because it"s not have any constraint.

2. **Unique:** The Unique constraint imposes that every value in a column or set of columns be unique. It means that no two rows of a table can have duplicate values in a specified column or set of columns.

Example: SQL> create table student123 (Emp_Id Number (5) NOT NULL,
Emp_Name Varchar2 (30) NOT NULL,
Address Varcahr2 (40),
Phone_No Number (10) **UNIQUE**);

Case 1: SQL> insert into student123 values („& Emp_Id", „& Emp_Name", „& Address", „& Phone_No"); Enter value for Emp_Id: 10101
Enter value for Emp_Name: Shaik Mohammad
student123 Enter value for Address: Tirupati
Enter value for Phone_No: 9491202987
Case 2: SQL> insert into student123 values („& Emp_Id", „& Emp_Name", „& Address", „&

Phone_No"); Enter value for Emp_Id: 10102

Enter value for Emp_Name: Mohammad student123
Enter value for Address: Chittoor
Enter value for Phone_No: **9491202987**

In the above case 2 it is clear that when we try to insert duplicate values into „Phone_No" we get error message. But „Emp_Name" have duplicate value it"s not any error, because it"s not have unique constraint.

Example: UNIQUE (Emp_Id, Phone_No);

3. **Primary Key:** A Primary Key constraint creates a primary key for the table. Only one primary key can be created for each table. The primary key uniquely identifies every row in the table. In simple terms primary key is a combination of UNIQUE and NOT NULL. The primary key constraints can be defined at the column-level or table-level.

Example: SQL> create table student123 (Emp_Id Number (5) **PRIMARY KEY,**
Emp_Name Varchar2 (30) NOT NULL,
Address Varcahr2 (40),
Phone_No Number (10) UNIQUE);

Case 1: SQL> insert into student123 values („& Emp_Id", „& Emp_Name", „& Address", „& Phone_No"); Enter value for Emp_Id: 10101
Enter value for Emp_Name: Shaik Mohammad
student123 Enter value for Address: Tirupati
Enter value for Phone_No: 9491202987

Case 2: SQL> insert into student123 values („& Emp_Id", „& Emp_Name", „& Address", „& Phone_No"); Enter value for Emp_Id: **10101**
Enter value for Emp_Name: Mohammad student123
Enter value for Address: Hyderabad
Enter value for Phone_No:

In the above case 2 it is clear that when we try to insert duplicate values into „Emp_Id" we get error message. But „Phone_No" have NULL value it"s not any error, because it"s have only unique constraint.

A Composite Primary Key is created by using the table-level definition:

Example: SQL> create table student123 (Emp_Id Number (5),
Emp_Name Varchar2 (30) NOT NULL,
Address Varcahr2 (40),
Phone_No Number (10),
Primary Key (Emp_Id, Phone_No));

4. **Foreign / Reference Key:** The Foreign / Reference integrity constraint, when a foreign key in one relation references primary key in another table, the foreign key value must match with the primary key value. In other words, the referential integrity say"s „pointed to" information must exist.

The foreign key is defined in the „child table", and the table containing the referenced column is the „parent table". Foreign key constraint can be defined at the column-level or table-level. A composite foreign key must be created by using the table-level definition.

- **Foreign Key:** It is used to define the column in the child table at the table-level constraint.

- **References:** It identifies the parent table and primary key column in the parent table.

Example: Consider the parent table is „student123_DEPT" and child table is „STUDENT, the child table which has foreign key which references primary key in parent table. It is to be noted that the parent table should be created first, then the child table.

- **Parent table: student123_DEPT:** Dept_Id, Dept_Name, Location

- **Child table: STUDENT:** Emp_Id, Department_Id, Emp_Name

student123_DEPT

Dept_Id	**Dept_Name**	**Location**
D100	Electrical	Rompicherla
D101	Civil	Chittoor
D102	Computer	Tirupati

STUDENT

Emp_Id	**Department_Id**	**Emp_Name**
E505	D101	Shaik
E503	D100	Mohammad
E507	D102	student123

Parent table: student123_DEPT table is created with „Dept_Id" as primary key.

```
SQL> create table student123_DEPT (Dept_Id varchar2(5) Primary key,
                                   Dept_Name varcahr2(15),
                                   Location varchar2(20));
```

Child table: STUDENT table is created with „Department_Id" as references „Dept_Id".

```
SQL> create table STUDENT (Emp_Id varchar2(5),
                           Department_Id varchar2(5),
                           Emp_Name varchar2(20),
                           Foreign key (Department_Id references student123_DEPT(Dept_Id));
```

```
SQL> insert into STUDENT values („E502", „D105", „Shaik Abu");
```

Here the values are not able to insert into the STUDENT table, because we have try to insert „Department_Id" into the STUDENT table (child table) which is not matched with „Dept_Id" of the student123_DEPT table (parent table).

5. **Check:** The „Check" constraint defines a condition that each row must satisfy. It is used to limit the value range that can be placed in a column. The check constraint is added to the declaration of the attribute. Attribute value check is checked only when the value of the attribute is inserted or updated.

Example 1:
```
SQL> create table voter (Voter_Name varchar2(15),
                         Voter_Age number(3),
                         Voter_Address varchar2(25),
                         CHECK (Voter_Age >=18));
```

```
SQL> insert into voter values („Mohammad student123", „17", „Tirupati");
```

In the above example we can observe that we can try to insert a value which violates the check constraint, we get error message.

Example 2:
```
SQL> create table person (Name varchar2(15),
                          Gender char(8) CHECK (Gender IN („M", „F")),
                          Address varchar2(25));
```

```
SQL> insert into person values („Mohammad student123", „Male", „Tirupati");
```

In the above example we can observe that we can try to insert a gender value „Male" instated of „M" which violates the check constraint, we get error message.

6. **Default:** The „Default" constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified. The most common default value is NULL. This can be used with columns not defined with a NOT NULL. Oracle automatically inserted the value specified as the default value when no value is specified for the particular column during inserting.

Example 1: SQL> create table shaikemp (Emp_No number(4),
Emp_Name varchar2(15),
Hiredate **DEFAULT Sysdate,**
Salary number(7, 2),
Primary key(Emp_No));

Example 2: SQL> create table student123 (First_Name
varchar2(15),
Last_Name varchar2(15),
Address varchar2(15) **DEFAULT „TIRUPATI",**
Phone_No number(10));

********** (Q) Explain Join Operations in SQL?**

SQL Joins are used to related information in different tables. Joins are used to combine columns from different tables. The connection between tables is established through the WHERE clauses, called SQL Joins condition.

Definition of Join: "A Join is a Query that enables retrieval of rows from more than one table having a common column between both the tables".

Types of Joins: A Join consists of the following types.

1. Equi Join / Natural Join
2. Non Equi Join
3. Cartesian Join / Cross Join
4. Outer Join
5. Self Join

1. Equi Join (Natural join):

- When two tables are joined together using equality of values in one more columns, they make an Equi Join. It is also called a Natural Join.
- The Equi Join has a predicate based on equality.
- In this Join, the condition is specified using "=" operator between two columns of same datatype of two different tables.

Syntax for Equi Join: select table1.<column names> , table2.<column names> from <table name1>, <table name2> where table1.<common column name>= table2.<common column names>;

SQL> select flight_sch.airbus, depart_time, flight_date, cost from flight_sch, flight
where flight_sch.airbus = flight.airbus;

```
AIRBUS   DEPART_TIME   FLIGHT_DATE    COST
 A001       12.00      12-AUG-2020     5000
 B002       15.30      18-SEP-2020     8000
 C003       18.30      10-FEB-2020     9500
 C003       18.30      10-MAR-2020    12500
 C003       18.30      10-MAR-2020    12500
 T006       22.00      25-JAN-2020    15000
```

2. Non Equi Join:

- we can also join values in two columns that are not equal.
- In this Join, the condition is specified using "<>" operator between two columns of same datatype of two different tables.

Syntax for Non Equi Join: select table1.<column names>, table2.<column names> from <table name1>, <table name2> where table1.<common column name> <> table2.<common column names>;

SQL> select flight.airbus, flight_date, cost from flight, flight_sch
where flight.airbus <> flight_sch.airbus;

```
AIRBUS   FLIGHT_DATE    COST
 G005    14-MAR-2020     4500
 H008    26-DEC-2020     1520
 W009    19-JUNE-2020    2500
```

3. **Cartesian Join:** The Cartesian joins performs a relational product of two tables. The Cartesian product results set are the number of rows in the first table multiplied by the number of rows in the second table. This is also known as Cross Join. A Cartesian join that does not have a Where clauses, in this case the tables are joined without any condition. If Where clause is added, the cross join behave as an Inner join.

4. **Outer Join:** An Outer Join is a Join similar to a Join with a small difference. An Outer Join is different from all other types of Joins it returns.

When table are joined, rows which contain matching values in the join predicate, are returned. Sometimes we may want both matching and non-matching rows returned for the tables that kind of join is known as „Outer Join".

- ✔ All the rows that are turned by a simple join and
- ✔ All the rows of one table that don"t match with the rows in the other table.
- ✔ Outer Join makes use of the operator **"+"**

sign Outer Join has two kinds there are:

A. Left Outer Join.
B. Right Outer Join.

A. **Left Outer Join:** This join can display the information from two tables according to given condition and also it displays additional information from left table. It is represented with "+" symbol with in parentheses. If should be placed before equal to (=) operator.

Syntax for Left Outer Join: select table1.<column names>, table2.<column names>
from <table name1>, <table name2>
where table1.<common column name>(+)= table2.<common column names>;

```
SQL> select flight_sch.airbus, depart_time, flight_date, cost from flight_sch, flight
        where flight_sch.airbus(+) = flight.airbus;

AIRBUS   DEPART_TIME   FLIGHT_DATE    COST
 A001       12.00      12-AUG-2020     5000
 B002       15.30      18-SEP-2020     8000
 C003       18.30      10-FEB-2020     9500
 C003       18.30      10-MAR-2020    12500
 C003       18.30      10-MAR-2020    12500
 ****       *****      14-MAR-2020     4500
 ****       *****      26-DEC-2020     1520
 T006       22.00      25-JAN-2020    15000
 ****       *****      19-JUN-2020     2500
```

B. **Right Outer Joint:** It is opposite to Left Outer Join, if provided the additional information out of given condition from Right table. It is also indicates with "+" symbol with in parentheses. It should be take place at the end of equi outer join.

Syntax for Right Outer Join: select table1.<column names>, table2.<column names>
from <table name1>, <table name2>
where table1.<common column name>= table2.<common column names>(+);

```
SQL> select flight_sch.airbus, depart_time, flight_date, cost from flight_sch, flight
        where flight_sch.airbus = flight.airbus(+);

AIRBUS   DEPART_TIME   FLIGHT_DATE    COST
 A001       12.00      12-AUG-2020     5000
 B002       15.30      18-SEP-2020     8000
 C003       18.30      10-FEB-2020     9500
 C003       18.30      10-MAR-2020    12500
 C003       18.30      10-MAR-2020    12500
 D004       12.05         ********     ****
 F009       18.30         ********     ****
 T006       22.00      25-JAN-2020    15000
```

5. **Self Join:** The Self Join is used to match and retrieve rows that can have a matching value in different columns of the same table. Or other words a Join is made on a single table as through as two separate tables".

```
SQL> create table emply_student123 (Emp_No varchar2(5), Emp_Name varchar2(15),
                                    Manager_No varchar2(5));

Table created.

SQL> insert into emply_student123 values

('E001','Shaik','E002'); 1 row created.
```

View the resulting table:

```
SQL> select * from emply_student123;

EMP_NO   EMP_NAME    MANAGER_NO
  E001   Shaik          E002
  E002   Mohammad       E005
  E003   student123     E004
  E004   Abu           *****
  E005   SMF           *****
```

Syntax for Self Join: select alias name1. column names, alias name2. column names
from actual table name alias name1, actual table name alias name2
where alias name1.column name = alias name2.column name;

```
SQL> select empfar.emp_name, margfar.emp_name as manager_name from emply_student123 empfar,
     emply_student123 margfar where empfar.manager_no = margfar.emp_no;

EMP_NAME     MANAGER_NAME
Shaik        Mohammad
student123   Abu
Mohammad     SMF
```

***** (Q) Explain Set Operators in SQL?**

Set operators are used to combine information of similar type from one or more than one table. In SQL Set operators can be generally classified into following categories:

1. Union
2. Union all
3. Intersect
4. Minus

```
SQL> create table student123_shaik (Emp_Id number(5), Emp_Name varchar2(15), Salary

number(5)); Table created.
```

Inserting records into the table:

```
SQL> insert into student123_shaik values (100, 'Shaik',

50000); 1 row created.
```

View the resulting table:

SQL> select * from student123_shaik;

```
EMP_ID   EMP_NAME    SALARY
   100   Shaik        50000
   101   Mohammad     75000
   102   student123   60000
   103   Shaik Abu    55000
   104   SMF          48000
```

Table_Name 2: shaik_student123. **Column_Name:** Emp_Id, Emp_Name, Salary

Creating Table Structured:
SQL> create table shaik_student123 (Emp_Id number (5), Emp_Name varchar2(15), Salary number(5)); Table created.
Description about Table:

SQL> desc shaik_student123;

```
   Name           Null?    Type
 --------------- -------- -----------------
   EMP_ID                  NUMBER(5)
   EMP_NAME                VARCHAR2(15)
   SALARY                  NUMBER(5)
```

Inserting records into the table:

SQL> insert into shaik_student123 values (101, 'Mohammad',

75000); 1 row created.

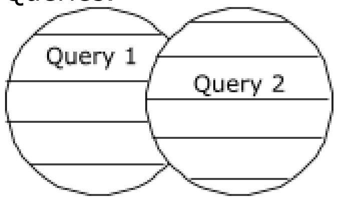
View the resulting table:

SQL> select * from shaik_student123;

```
EMP_ID   EMP_NAME    SALARY
   101   Mohammad     75000
   103   Shaik Abu    55000
   105   Muneer       54000
   106   Mujeeb       50000
   107   Sudheer      58000
```

1. UNION Operator:

- SQL Union operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.
- To use „Union", each SELECT must have the same number of columns selected, the same number of column expressions, the same datatype, and have them in the same order, but they do not have to be the same length.
- Aggregate functions can"t be use with Union clause.
- Union"s can"t be use in Sub-Queries.



Syntax for UNION Operator:

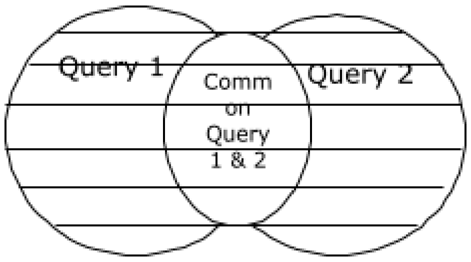
```
SQL> Select <Column names> from <Table Name1> where [<condition>]
                                   UNION
      Select <Column names> from <Table Name2> where [<condition>];
```

```
Example: SQL> select Emp_Id, Emp_Name, salary from student123_shaik
                                   UNION
                 select Emp_Id, Emp_Name, salary from shaik_student123;
```

EMP_ID	EMP_NAME	SALARY
100	Shaik	50000
101	Mohammad	75000
102	student123	60000
103	Shaik Abu	55000
104	SMF	48000
105	Muneer	54000
106	Mujeeb	50000
107	Sudheer	58000

2. UNION ALL Operator:

- The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.
- The same rules that apply to UNION apply to the UNION ALL operator.



Syntax for UNION ALL Operator:

```
SQL> Select <Column names> from <Table Name1> where [<condition>]
                                   UNION ALL
      Select <Column names> from <Table Name2> where [<condition>];
```

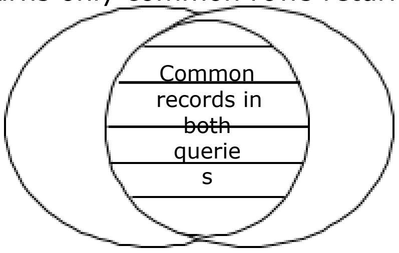
```
Example: SQL> select Emp_Id, Emp_Name, salary from student123_shaik
                                   UNION ALL
                 select Emp_Id, Emp_Name, salary from shaik_student123;
```

EMP_ID	EMP_NAME	SALARY
100	Shaik	50000
101	Mohammad	75000
102	student123	60000
103	Shaik Abu	55000
104	SMF	48000
101	Mohammad	75000
103	Shaik Abu	55000
105	Muneer	54000
106	Mujeeb	50000

107	Sudheer	58000

3. INTERSECT Operator:

- The SQL Intersect operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.
- This means „Intersect" returns only common rows returned by the two SELECT statements.



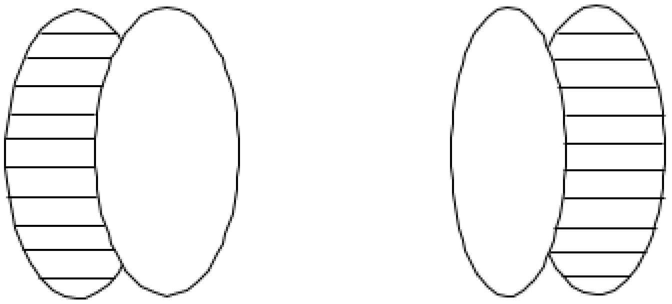
Syntax for INTERSECT Operator:

```
SQL> Select <Column names> from <Table Name1> where [<condition>]
                              INTERSECT
     Select <Column names> from <Table Name2> where [<condition>];
```

Example: SQL> select Emp_Id, Emp_Name, salary from student123_shaik
INTERSECT
select Emp_Id, Emp_Name, salary from shaik_student123;

EMP_ID	EMP_NAME	SALARY
101	Mohammad	75000
103	Shaik Abu	55000

4. **MINUS Operator:** The Minus Operator combine the result of the two query and returns only those values selected by the "First Query and not the Second Query".



A – B **B – A**

Syntax for MINUS Operator:

```
SQL> Select <Column names> from <Table Name1> where [<condition>]
                              MINUS
     Select <Column names> from <Table Name2> where [<condition>];
```

Example (A – B): SQL> select Emp_Id, Emp_Name, salary from student123_shaik
MINUS
select Emp_Id, Emp_Name, salary from shaik_student123;

EMP_ID	EMP_NAME	SALARY
100	Shaik	50000
102	student123	60000

```
104      SMF               48000
```

Example (B – A): SQL> select Emp_Id, Emp_Name, salary from shaik_student123
MINUS
select Emp_Id, Emp_Name, salary from student123_shaik;

```
EMP_ID   EMP_NAME    SALARY
   105   Muneer       54000
   106   Mujeeb       50000
   107   Sudheer      58000
```

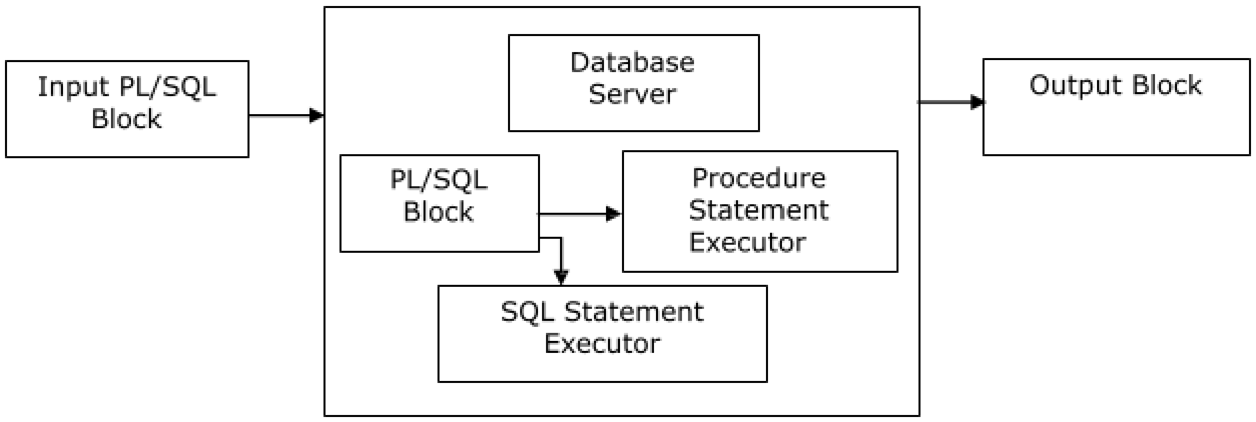
UNIT – V
Procedural Language / Structured Query Language (PL/SQL).
Chapter – 1
Introduction to PL/SQL

Introduction of PL/SQL:

- PL/SQL is provided by Oracle as a procedural extension to SQL.
- PL/SQL is a block structured language. This means a PL/SQL program is made up of blocks, where block is a smallest piece of PL/SQL code.
- SQL is a declarative language the statements have no control to the program and can be executed in any order.
- PL/SQL arose from the desire of programmers to have a language structure that was more familiar than SQL"s purely declarative nature.
- PL/SQL supports almost all programming constructs like variables, conditional statements, looping etc.,

(Q) Explain Basic Structure or Architecture of PL/SQL?

PL/SQL Engine



PL/SQL Block Structure: The Programming area in PL/SQL is called block. This block mainly contains three parts. They are:

1. Declarative Part.
2. Executable Part.
3. Exception Handling part.

Structure:

```
SQL> Declare
          ……………  ⎫
          ……………  ⎬ Variable Declaration
          ……………  ⎭
        Begin
             Statement 1 ⎫
             Statement 2 ⎬ Executable Part
              ……………     ⎭
        End;
      Exception              Error Handling Part
```

1. **Declarative Part:** Declarative section of PL/SQL block starts with the keyword „Declare". This section is used to declare any placeholders (storage area) like variables, constants, records and cursors, which are used to manipulate data in the executable section.

2. **Executable Part:** The executable section of a PL/SQL block starts with the keyword „Begin" and ends with „End". This section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements from the part of executable section. It is a mandatory section.

3. **Exception Part:** The exception section of a PL/SQL block starts with the keyword „Exception". 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 immediately with errors.

NOTE: Every PL/SQL program must consist of at least one block, which may consist of any number of nested sub-blocks.

Input & Output Statement in PL/SQL:

1. **Input Statement:** The Input statement in PL/SQL is "&" Operator. The „&" Operator it is to give the Input values to the Variables.

Example:
```
a : = & a;
x : = & x; ………
```

2. Output Statement: The Output Statement in PL/SQL is:

dbms_output.put_line(„Message"/Variables); **Example: SQL>** dbms_output.put_line („Welcome to

Shaik Mohammad student123. MCA, M.Sc CS, IRPM"); **Advantages of PL/SQL:**

1. **Support for SQL:** PL/SQL allows the use all SQL commands, as well as all SQL functions, operations and datatypes. So we can manipulate Oracle data flexibility and safely.

2. **Block Structured:** PL/SQL is a block-structured language. Each program written in PL/SQL is written as a block. Blocks can also be nested. Each block is meant for a particular task. PL/SQL blocks can be stored in the database and reused.

3. **Modularity:** Modularity means divide an application or a process into manageable, well defined modules. PL/SQL allows process to be divided into different modules such as procedures and functions, called sub-programs.

4. **Control Structures:** PL/SQL allows control structures like If-statement, for loop-statement, while loop-statement to use in the block. Control structures are most important in PL/SQL extension to SQL. It controls the procedural flow of the program, deciding if and when SQL and other actions are to be executed.

5. **Integration:** As PL/SQL is the product of Oracle Corporation. So, it integrates well with SQL*Plus and other application development products of Oracle. It bridges the gap between convenient access to database technology and the need for procedural programming capability.

6. **Portability:** Applications written in PL/SQL are portable to any platform on which Oracles runs. Once we write a program in PL/SQL, it can be used in any environment without any changes.

7. **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.

Short comings (disadvantages): SQL is a powerful tool for accessing the database but it suffers from some deficiencies as follows:

- SQL statements can be executed only one at a time. Every time to execute a SQL statement, a call is made to Oracle Engine, thus it results in an increase in database overheads.

- While processing an SQL statement. If an error occurs, Oracle generates its own error message, which is some time difficult to understand.
- If a user want to display some other meaningful error message. SQL does not have provision for that.

(Q) Explain about Lexical units / Fundamental / Language Elements of PL/SQL?

Lexical Unit: A line of PL/SQL program contains groups of characters known as lexical units, which can be classified as follows:

- Delimiters
- Identifiers
- Literals
- Comments

1. **Delimiters:** A delimiter is a simple or compound symbol that has a special meaning to PL/SQL. Simple symbol consists of one character, while compound symbol consists of more than one character. PL/SQL supports following simple symbol delimiters: + - * / = @ ; <> != || -- /* */ := etc.

2. **Identifiers:** Identifiers are used in the PL/SQL programs to name the PL/SQL program items as constants, variables, cursors, subgroups, etc.

Rules of Identifies:

- Identifiers can consists of alphabets, numbers, dollar signs, underscores and number signs only.
- An identifier must begin with an alphabetic letter optionally followed by one or more characters.
- An identifier cannot contain more than 30 characters.
- An identifier cannot be reserved word.

3. **Literals:** A Literal is an explicit defined character, string, numeric or Boolean value, which is not represented by an identifier.

4. **Comment:** A comment can appear anywhere in the program code. The compiler ignores comments. A PL/SQL comment may be a single-line or multiline comment.

- **Single-line:** Double hyphen (--) anywhere on a line and extend to the end of the line.
- **Multi-line:** Begin with a slash-asterisk (/*) and end with an asterisk-slash (*/).

(Q) Explain about various datatypes using in PL/SQL?

A datatype specifies the space to be reserved in the memory, type of operations that can be performed, and valid range of values. PL/SQL supports all the built-in SQL datatypes. A part from those datatypes, PL/SQL provides some other datatypes. Some commonly used PL/SQL datatypes are as follows:

1. Number
2. Char
3. Varchar2
4. Date
5. Boolean
6. %Type
7. %Rowtype

1. **Number:** The Number datatype to store fixed or floating point number of virtually any size. we can specify precision, which is the total number of digits, and scale, which determines where rounding, occurs.

Syntax: number (Precision, scale);

2. **Char:** The Char datatype to store fixed-length character data.

Syntax: char (maximum length);

3. **Varchar2:** The Varchar2 datatype to store varying-length character data.

Syntax: varchar2 (maximum length);

4. **Date:** The Date datatype to store fixed-length data value.

Syntax: date;

5. **Boolean:** A Boolean datatype is assigned to those variables, which are required for logical operations.

Syntax: Boolean;

6. **%Type:** To avoid type and size conflict between a variable and the column of a table, the attribute „%type" is used. Advantage of this method of defining a variable is that, whenever the type and / or size of a column in the table are changed, it is automatically reflected in the variables declaration.

Syntax: <new column name> <table name>.<existing column name>%type;

Example: Temp_name shaikemp.ename%type;

Here, the new variable name as „Temp_name" will be of the same size and type as that of the „ename" column of „shaikemp" table.

7. **%Rowtype:** The %Rowtype attribute provide a record type that represents a row in a database table or view. The record can stored an entire row of data selected from the table or fetched from a cursor or

cursor variable. In case, variables for the entire row of a table need to be declared, then instead of declaring then individually, the attribute „%rowtype" is used.

Syntax: <variable name> <table name>%rowtype;

Example: emp_row shaikemp%rowtype;

Here, the variable „emp_row" will be a composite variable, consisting of the column names of the table as its members.

(Q). Explain Operators Precedence in PL/SQL?

The operators within an expression are done in a particular order depending on their precedence (priority). Table below listed the operator"s level of precedence from top to bottom. Operators with higher precedence are applied first, but if parentheses are used, expression within innermost parenthesis is evaluated first.

Operators	Descriptions
( )	Function Call
++, --	Incremental and Detrimental
* / %	Multiplication, Division, Modulo Division
+ -	Addition and Subtraction
<, >, < =, > =	Less than, Greater Than, Less than or equal to, greater than or equal.
== , !=	Equality and Inequality
AND	Logical AND / Conjunction
OR	Logical OR / Disjunction
NOT IN	Logical NOT
LIKE	Comparison of Like
BETWEEN	Comparison of Between
IN	Comparison of In

(Q) Explain how to declare the variables and constants in PL/SQL?

(I). Variables in PL/SQL:

- Variables are placeholders (storage area) used to hold the data values that can changes through the PL/SQL block.
- Each variable in PL/SQL has a specific datatype, which determines the size and lawet of the variables memory, the range of values that can be stored within that memory and he set of operations that can be applied to the variables.

Variable declaration on PL/SQL: PL/SQL variables must be declared in the declaration section of PL/SQL block. When we declare a variable, PL/SQL allocates memory for the variable"s value and the storage location is identified by the variable name.

Syntax: <variable name> datatype (size); **(or)** <variable name> datatype [Not Null:= Value];

Example: SQL> Declare
Salary Number (5);

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 two ways given below:

(a). **Direct method:** We can directly assign values to variables:

Syntax: <variable name>:= Value;

(b). **Select...Into Statement:** We assign values to variables directly from the database columns by using a Select…Into statement.

Syntax: select column name into <variable name> from <table name> where [<condition>];

Example: The below program will get the salary of an employee with id '9491202987'.

```
SQL> DECLARE
          var_salary number(6);
          var_emp_id number(10):= 9491202987 ;
        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.
In PL/SQL block, variables are declared in the DECLARE section.

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 an inner block and cannot be referenced by outside blocks.

Global Variables: These are declared in an outer block and can be referenced by itself and by its inner blocks.

```
DECLARE                                   Global Variables            }
     var_num1 number;
     var_num2 number;
 BEGIN
     var_num1 : = 100;                    Local Variables           →
     var_num2 : = 200;                                          Inner
   DECLARE
        var_mult number;
      BEGIN
```

Outer Block
Block

```
            var_mult : = var_num1 *
            var_num2;
    END;
END;
/
```

(II). **Constants in PL/SQL:** 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. we can declare a constant and use it instead of actual value.

Syntax: <constant name> CONSTANT datatype (size): = 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. we cannot assign a value later.

For example: To declare salary_increase, we can write code as follows:

```
SQL>  DECLARE
            salary_increase CONSTANT number (3):= 100;
```

****** (Q) Explain the Control Statements in PL/SQL?**

PL/SQL supports programming language features like conditional statements, iterative statements. PL/SQL control structure allows we to control the behaviour of the block as it runs. we can change the logical flow of statements within the PL/SQL block with a number of control structures.

(I). **Conditional Control Structures:** A conditional control structure tests a condition to find out whether it is true or false and accordingly executes the different blocks of SQL statements. Conditional control is generally performed by IF statement. There are three forms of IF statements:

(a). IF…THEN - END IF.
(b). IF…THEN…ELSE – END IF.
(c). IF…THEN…ELSIF – END IF.

(a). **IF…THEN – END IF:** It is used test only one condition. It returns when if the condition is True. Otherwise the condition will exit. Every IF statement should be ended with keyword "End if".

Syntax:
```
If <condition> then
          Statement 1;
          Statement 2;
            ………………….
      End If;
```

Example: SQL> ed biggest.sql;
```
Declare
      a number;
      b number;
 Begin
      a:= 20;
      b:= 10;
   if (a > b) then
         dbms_output.put_line („A is Bigger than
   B"); End If;
End;
/
```

Test Data: SQL> @ biggest.sql;

```
                   A is Bigger than B
```

PL/SQL procedure successfully completed.

NOTE: To enable the server output, the "set serveroutput on" command must be given at the SQL*PLUS prompt to the execution of the „dbms_output.put_line()“ procedure. Hence, the different variables are concatenated with double pipe (||) symbol.

(b). **IF…THEN…ELSE – END IF:** It is used to test a condition if condition is true then return True statement, otherwise its return"s false statements.

Syntax:
```
If <condition> then
        Statements;
    Else
        Statements;
        ………………….
    End If;
```

Example: SQL> ed twobiggest.sql;
```
Declare
    a number;
    b number;
 Begin
    a:= 10;
    b:= 20;
   if (a > b) then
        dbms_output.put_line („A is Bigger than B“);
   else
        dbms_output.put_line („B is Bigger than A“);
   End If;
End;
/
```
Test Data: SQL> @ twobiggest.sql;

```
                   B is Bigger than A
```

PL/SQL procedure successfully completed.

(c). **IF…THEN…ELSIF – END IF:** This statement selects an action from several mutually exclusive alternatives.

Syntax:
```
if <condition 1> then
        Statement 1;
    Elsif <condition 2> then
        Statement 2;
    Elsif <condition 3> then
        Statement 3;
        …………………….
    Else
          Statement n;
    End if;
```

Example: SQL> ed threebiggest.sql;

```
                  declare
                        a number(2);
                        b number(2);
                        c number(2);
                                                   begin a:=&a;
                                                         b:=&b;
                                                         c:=&c;
                                                            if ((a>b) and (a>c)) then
                                                                  dbms_output.put_line ('The Biggest Number is
                                                                  a');
                                                    else  if (b<c) and (a>c) then
                                                             dbms_output.put_line ('The Biggest Number is b');

                                                    else     dbms_output.put_line ('The
                                                          Biggest Number is c'); end if;
                        end if;
                     end;
                  /
```

Test Data: SQL> @ threebiggest.sql;

```
Enter value for a: 54
Enter value for b: 97
Enter value for c: 91

      The Biggest Number is b
```

PL/SQL procedure successfully completed.

NOTE: The „&" symbol is called an input operator which is mainly used to input a data value for a variable at execution time.

(II). **Case Statement:** Case statement comparing one by one sequencing conditions. Case statement attempts to match expression that is specified in one or more WHEN condition.

Syntax: Case

W
h
e
n
s
e
a
r
c
h
_
c
o
n
d

i
t
i
o
n
1
T
h
e
n
r
e
s
u
l
t
1
W
h
e
n
s
e
a
r
c
h
_
c
o
n
d
i
t
i
o
n
2
T
h
e
n
r
e
s

u
l
t
2
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
When search_condition-N Then result-N

```
Else
        Statement
End;
```

Here searched case expression has no selector. Also, it"s WHEN clause contains search conditions that yield a Boolean value, not expressions that can yield a value of any type.

Example: SQL> ed grade.sql;

```
Declare
     grade char(1):= upper („& grade“);
     appraisal varchar2(20);
Begin                              A
                                     p
                                     p
                                     r
                                     a
                                     i
 End; End;                           s
                                     a
                                     l
                                     :
                                     =
                                     C
                                     A
                                     S
                                     E
                                     g
                                     r
                                     a
                                     d
                                     e
                                     W
                                     H
                                     E
                                     N
                                     „
                                     A
                                     “
                                     T
                                     H
                                     E
                                     N
                                     „
                                     E
                                     x
                                     c
                                     e
                                     l
                                     l
                                     e
                                     n
                                     t
                                     “
                                     W
                                     H
                                     E
                                     N
                                     „
```

B“ THEN „VeryGood“ WH

EN „C“ THEN „Good“

Else „No Such Grade“

dbms_output.put_line („The Grade: “ || grade || „Appraisal“ || appraisal);

Test Data: SQL> @ grade.sql;

Enter the value for grade: a

The Grade: A Appraisal Excellent

PL/SQL procedure successfully completed.

(III). **Loop Control Structures:** Iterative control statements or Loop control statements are used when we want to repeat the execution of one or more statements for specified number of times. PL/SQL provides the following types of loops:

(a). Simple / Basic Loop.
(b). While Loop.
(c). For Loop.

(a). **Simple / Basic Loop:** The simplest form of loop statement is the basic (or infinite) loop. Which enclose a sequence of statements between the keywords „LOOP“ and „END LOOP“. Each time the flow of execution reaches the „END LOOP“ statement, control is returned to the corresponding „LOOP“ statement above it. A basic loop allows execution of its statement at least once, even if the condition is already met upon entering the loop. Without the „EXIT“ statement, the loop would be infinite or endless.

EXIT Statement: The EXIT statement to terminate a loop. Control passes to the next statement after the „END LOOP“ statement. The EXIT is used either as an action within an IF statement or as a stand-alone statement within the loop. The EXIT statement must be placed inside of a loop. we can also attach a „WHEN“ clause to allow conditional termination of the loop. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated.

Syntax:
```
LOOP
        Statements;
        - - - - - - - - - -
        - - - - - - - - - -
```

```
    EXIT [WHEN condition];
END LOOP;
```

Example: SQL> set serveroutput on;

SQL> ed factorial.sql;

```
        declare
            n number;
            f  number;
            i number;
        begin
                                                                                    f:= 1;
                                                                                    i:= 1;
                                                                                    n:= &number;
                                                                            loop
                                                                                    if (i>n) then
            exit;
                    else
                        f:= f * i;
                        i:= i+1;
                end if;
        end loop;
        dbms_output.put_line ('Factorial of ' || n || ' is : ' || f);
        end;
    /
```

Test Data: SQL> @ factorial.sql;

```
Enter the value for number: 5

Factorial of 5 is: 120
```

PL/SQL procedure successfully completed.

(b). **While Loop:** The While loop to repeat a sequence of statements until the controlling condition is „True". The condition is evaluated at the start of each iteration. The loop terminates when the condition is „False". If the condition if False at the start of the loop, then no further iterations are performed.

Syntax: While <Condition> Loop

```
                Statements;
                - - - - - - - - - - -
                - - - - - - - - - - -
            End Loop;
```

NOTE: If the variables involved in the conditions do not change during the body of the loop, then the condition remains True and the loop does not terminate.

Example: SQL> ed factwhile.sql;

```
        declare
            n number;
            f  number;
            i number;
        begin                                                      f:= 1;
                                                                   i:= 1;
                                                                   n:= &number;
        while (i <= n)
        loop
                f:= f * i;
                i:= i+1;
        end loop;
        dbms_output.put_line ('Factorial of ' || n || ' is : ' || f);
        end;
    /
```

Test Data: SQL> @ factwhile.sql;

Enter the value for number: 4

Factorial of 4 is: 24

PL/SQL procedure successfully completed.

(c) For Looping: This looping statement will execute loop body repeatedly initial values to final value for every execution. The loop counter will be incremented by one. So by using this statement we can also find the number of iterations. This statement also called as "Recursive Looping" statement.

Syntax: for counter IN [Reverse]

```
              <lower bound>..<upper bound>
          Loop
               Statement 1;
                 ……………. ⎫
                 ……………  ⎬ Loop Body.
                 …………..  ⎭
          End Loop;
```

- **Counter:** Is an implicitly declared integer whose value automatically increases or decreases by 1 (if the reverse keyword is used) until the upper or lower bound is reached.

- **Reverse:** Reverse causes the counter to decrement with each iteration from the upper bound to the lower bound.

- **Lower bound:** It specifies the lower bound for the range of counter values.

- **Upper bound:** It specifies the upper bound for the range of counter values.

Example: SQL> forfactorial.sql;

```
        declare
           n number;
           f number;
        begin
              f:= 1;
              n:= &number;
```

```
    for i in 1..n loop
                                         f:= f * i;
    end loop;

    dbms_output.put_line ('Factorial of ' || n || ' is : ' || f);
    end;
  /
```

Test Data: SQL> @ forfactorial.sql;

```
Enter the value for number: 7

Factorial of 4 is: 5040
```

PL/SQL procedure successfully completed.

(IV) **Sequential Control:** By default, all PL/SQL blocks are executed in a top-down sequential process. The process begins with a „Begin" statement and terminates with an „End" statement. So, to change the sequence of execution of statements, we can use following unconditional statements.

(a). Goto Statement.
(b). Continue Statement.
(c). NULL Statement.

(a). **Goto Statement:** The Goto statement immediately transfer program control unconditionally to a named statement label or block label. It can be in the forward direction or in the backward direction. The statement or label name must be unique in the block. Overuse of Goto statement may increase the complexity, thus as far as possible avoid the use of Goto statement.

Syntax:
```
Goto <<Label Name>>;
        - - - - - - -
        - - - - - - -
    <<Label Name>>
        - - - - - - -
        - - - - - - -
```

Example: SQL> ed gotolabel.sql;

```
    declare
      a integer;
    begin
      for a in 1..5 loop
        dbms_output.put_line(a);
          if a = 4 then
            goto
            student123;
          end if;
      end loop;
      <<student123>>
    dbms_output.put_line(„End of the Loop
  here"); end
```

/

Test Data: SQL> @ gotolabel.sql;

```
1   2   3   4   End of the Loop here
```

PL/SQL procedure successfully completed.

(b). **Continue Statement:** The Continue statement unconditionally skip the current loop iteration and next iteration as normal, only skip matched condition.

Syntax:
```
if <condition> then
          continue;
      end if;
```

Example: SQL> ed continueloop.sql;

```
declare
   num number:=0;
begin
   for num in 1..5 loop
      if num = 4 then
         continue;
      end if;
         dbms_output.put_line („Iteration:   " ||
   num); end loop
end;
/
```

Test Data: SQL> @ continueloop.sql;

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 5
```

PL/SQL procedure successfully completed.

(c). **NULL Statement:** Generally when we write a statement in the program, we want it to do something but in some cases we want to tell PL/SQL to do nothing and in such cases, NULL statement can be used. The NULL statement does nothing other than pass control to the next statement.

Syntax: NULL;

Example: SQL> ed nullstatement.sql;

```
declare
   a
number;
begin
      a:= &a;
      if a mod 2 != 0 then
         dbms_output.put_line („Number is Odd");
      else
```

```
NULL;
```

```
          end if;
      end;
      /
```

Test Data: SQL> @ nullstatement.sql;

Enter value for a: = 6

PL/SQL procedure successfully completed.

Chapter – 2
Cursors

********* (Q) Discusses about the Cursors in PL/SQL?**

Cursors are a private working area or cursors are a temporary working area. "It is mainly used for retrieving multiple rows based on multiple attributes at a time. Cursors are very useful to update big structure of a table at a time". Cursors can calculate calculation according to given conditions and fetching rows into base tables.

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.

Cursors are mainly divided into two types in SQL. There are:

I. Implicit Cursor. (System defined)
II. Explicit Cursor. (User defined)

I. **Implicit Cursor:** The Implicit Cursor is declared by PL/SQL implicitly 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. These are automatically executed by SQL Engine Internally.

Ex: Create Table, Alter Table, Create View, Update etc..,

Implicit Cursor Attributes: These attributes are associated with the implicit SQL cursor and can be accessed by appending the attribute name to the implicit cursor name (SQL).

Syntax: SQL%<attribute name>;

In PL/SQL implicit cursor has 4 attributes there are:

1. SQL%ISOPEN
2. SQL%FOUND
3. SQL%NOTFOUND
4. SQL%ROWCOUNT

1. **SQL%ISOPEN:** It is used to determine if a cursor is already open. It is always „False“ because Oracle automatically closes implicit cursor after executing its associated SQL statements.

2. **SQL%FOUND:** It is always „True“ if the most recently executed DML statement as successful.

3. **SQL%NOTFOUND:** It is „True“ of the most recently executed DML statement was not successful.

4. **SQL%ROWCOUNT:** This attribute is used to determine the number of rows that are processed by SQL statements.

Procedure: Open RUN prompt type „cmd“ command type „sqlplus“ on screen enter the user name: „scott“ enter the password: „student123“ and press enter key enter „SQL> set serveroutput on“ command in SQL type „SQL> edit <file name>.sql;“ now opened SQL*PLUS Notepad here enter the PL/SQL Code after the completion of PL/SQL code now run the code again go for SQL*PLUS DOS prompt Run the code „SQL> @ <file name>.sql;“

Example: Consider a PL/SQL code to display message to check whether the record is deleted or not.

SQL> edit stud_infoplsql.sql;

```
  declare
    begin
      delete from stud_info where stud_id = &stud_id;
      if SQL%FOUND then
          dbms_output.put_line ('Record Deleted');
      else
          dbms_output.put_line ('Record Not Deleted');
      end if;
    end;
  /
```

Test Data 1: SQL> @ stud_infoplsql.sql;

```
Enter value for stud_id: 110
old   3: delete from stud_info where stud_id =
&stud_id; new          3: delete from stud_info where
stud_id = 110; Record Deleted
PL/SQL procedure successfully completed.
```

Test Data 2: SQL> @ stud_infoplsql.sql;

```
Enter value for stud_id: 25122020
old   3: delete from stud_info where stud_id = &stud_id;
new   3: delete from stud_info where stud_id = 25122020;
Record Not Deleted
PL/SQL procedure successfully completed.
```

II. **Explicit Cursor:** The Explicit cursor is declared explicitly by the user, along with other identifiers to be used in a PL/SQL block. These are also known as „User defined“ cursors, defined in the „Declare“ section of the PL/SQL block. They must be declared when we are executing a „Select“ statement that returns more than one row. Even through the cursor stores multiple records, only one record can be processed at a time, which is called as „Current row“ or „active set“.

A PL/SQL program opens a cursor to process rows returned by a query and then closes the cursor.

Steps to create a cursor:

1. Declare the cursor
2. Open the cursor
3. Fetch data from the cursor
4. Close the cursor

1. **Declare the cursor:** Before using any cursor that should be declared first. In one PL/SQL Program we may use many cursors; all cursors must be defined under declarative part of the PL/SQL block. A cursor is defined in the declarative part by naming it and specifying a „SELECT" statement.

Syntax: CURSOR <cursor_name> IS <SELECT – statements>;

- **cursor_name:** A suitable name for the cursor.
- **select – statement:** A select query which returns multiple rows.

Example: declare

```
          cursor student123_cur is select * from shaikemp where sal > 2000;
```

In the above example, we are declaring a cursor „student123_cur" on a query which returns the records of all the employees with salary greater than 2000. Here „shaikemp" is the table which contains records of all the employees.

NOTE: The cursor name is undeclared identifier not a PL/SQL variable. It is used only to reference the query.

2. **Open the cursor:** After declaration, the cursor is opened with an „Open" statement for processing rows in the cursor. The SELECT statement associated with the cursor is executed when the cursor is opened.

Syntax: OPEN <cursor_name>;

Example: OPEN student123_cur;

3. **Fetch / Retrieve data from the cursor:** After the cursor is opened, the current row is loading into variables. The current row is the row at which the cursor is currently pointing. The transfer of data into PL/SQL variables is done through „FETCH" statement.

Syntax: FETCH <cursor_name> INTO <variables>; (or) FETCH <cursor_name> INTO <record_named>

NOTE: For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also their data types must be compatible.

SQL> edit shaikplsql.sql;

```
 declare
     eno shaikemp.empno%type;
     name shaikemp.ename%type;
     salary shaikemp.sal%type;
          cursor student123_cur is select empno, ename, sal from shaikemp where sal > 5000;
```

```
  begin
        open
        student123_cur;
        loop
            fetch student123_cur into eno, name,
            salary; exit when
            student123_cur%notfound;
            dbms_output.put_line (ENO || '    ' || NAME || '    ' || SALARY);
        end loop;
        close student123_cur;
 end;
 /
```

SQL> set serveroutput.on;

SQL> @ shaikplsql.sql;

PL/SQL procedure successfully completed.

4. **Close the cursor:** After working successfully with a cursor, it must be closed when we close a cursor. It will release all recourse.

Syntax: CLOSE <cursor_name>;

Example: CLOSE student123_cur;

Explicit cursor attributes: We use these attributes to avoid errors while accessing cursors through Open, Fetch and Close statements. There are the attributes available to check the status of an explicit cursor.

In PL/SQL explicit cursor has 4 attributes there are:

1. %ISOPEN
2. %FOUND
3. %NOTFOUND
4. %ROWCOUNT

1. **%ISOPEN:** „True", if the cursor is already open in the program otherwise „False" the cursor is not opened.

Syntax: <cursor_name>%ISOPEN;

Example: student123_cur%ISOPEN;

2. **%FOUND:** „True", if fetch statement returns at least one row otherwise „False", if fetch statement doesn"t return a row.
Syntax: <cursor_name>%FOUND;

Example: fetch student123_cur into eno, name,
salary; if student123_cur%found then

3. **%NOTFOUND:** „True", if fetch statement doesn"t return a row otherwise „False", if fetch

statement returns at least one row.

Syntax: <cursor_name>%NOTFOUND;

Example: fetch student123_cur into eno, name,
salary; exit when
student123_cur%notfound;

4. **%ROWCOUNT:** Return the number of rows fetched by the fetch statement. If no row is returned, the PL/SQL statement returns an error.

Syntax: <cursor_name>%ROWCOUNT;

Example: fetch student123_cur into eno, name,
salary; if student123_cur%found then

Query: Using cursors display the details of all those employees from „shaikemp" table whose sum of salary and commission is more than 3000?

```
SQL> set linesize 50;
SQL> set pagesize 100;
SQL> set serveroutput on;
SQL> edit
student123explict.sql;
   declare
         pempno shaikemp.empno%type;
         pname shaikemp.ename%type;
         psal shaikemp.sal%type;
         pdeptno shaikemp.deptno%type;
         phdate shaikemp.hiredate%type;
           cursor student123 is select empno, ename, sal, deptno, hiredate from
           shaikemp where sal+nvl(comm, 0)>1000;
     begin
       open
       student123;
       loop
         fetch student123 into pempno, pname, psal, pdeptno,
           phdate; if student123%found then
           dbms_output.put_line(pempno|| ' ' || pname || ' ' || psal || ' '|| pdeptno || ' ' || phdate);
             else
             exit;
           end if;
         end loop;
         close
         student123;
    end;
  /
SQL>@ student123explict.sql;
               EMPNO       ENAME        SAL   DEPTNO   HIREDATE
                7499    Abu             1600     30     20-Feb-81
                7521    James           1250     30     22-Feb-81
                7566    Faridha         2975     20     02-Apr-81
```

7654	Saleem	1250	30	28-Sep-81
7698	Masthan	2850	30	01-May-81
7782	Sudheer	2450	10	09-Jun-81
7788	Muneer	3000	20	19-Apr-87

```
7839   Mohammad          5000    10    17-Nov-81
       student123
7844   Suneel            1500    30    08-Sep-81
7876   Adam              1100    20    23-May-87
7902   Mujeeb            3000    20    03-Dec-81
7934   Raheem            1300    10    23-Jan-82
```

PL/SQL procedure successfully completed.

Chapter – 3
Subprograms & Triggers

(Q) Explain the Subprograms with an example in PL/SQL?

The PL/SQL programs can be stored in the database as stored programs and can be invoked whenever required. Such stored programs in PL/SQL are known as „subprograms“. PL/SQL has two types of subprograms called „Procedures“ and „Functions“. Generally, we use a procedure to perform an action and a function to compute a value.

The subprograms have a declaration section, execution section and optional 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.

A procedure has a header and body. The header consists of the name of the procedure and the parameters or variables passed to the procedure.

Block structure for PL/SQL Subprograms:

```
<Header> IS / AS                              Subprogram Specification

    Declaration section Begin
        Executable section Exception          Subprogram Body
      (optional)
        Exception section End;
```

Subprogram Specification: The header is relevant for named blocks only and determines the way that the program until is called or invoked.

- The PL/SQL subprogram type, that is, either a procedure or a function.
- The name of the subprogram.
- The parameter list, if one exists.
- The Return clause, which applies only to functions.
- The IS or AS keyword is mandatory.

Subprogram Body:
- **Declare:** The declaration section of the block between IS or AS and Begin. The keyword „Declare; that is used to indicate the start of the declaration section in anonymous blocks is not used here.

- **Execution:** The execution section between the Begin and End keyword is mandatory, enclosing the body of actions to be performed. There must be at least one statement existing in this section

- **Exception:** The exception section between Exception and End is optional. This section traps predefined error conditions. In this section, we define actions to take of the specified error condition arises.

Types of subprograms: PL/SQL subprograms are only divided into two types they are.

I. Procedure.
II. Function.

(I). **Procedure:** A Procedure or stored procedure is a named PL/SQL block that can accept parameters (sometimes referred to as arguments), and be invoked. Generally a procedure is mainly used to perform one or more specific task. A procedure has a header, declaration section, executable section and an optional exception-handling section.

Procedure can create with the „Create Procedure" statement, which may declare a list of parameters and must define the actions to be performed by the standard PL/SQL block. The „Create" clause enables to create stand-alone procedures, which are stored in an Oracle.

Syntax of Create Procedure Statement:

Procedure Specification

```
CREATE [OR REPLACE] PROCEDURE <Procedure
Name>
[(Parameter 1 [mode1] datatype 1, Parameter 2
[mode2] datatype 2 …….)]
{IS / AS}
[Local
    vari
    able
    s
    decl
    arati
    on]
    BEG
    IN
      PL/SQL Executable Statements

    [EXCEPTION
        Exception Handlers] END [<Procedure
    Name>];
```

Procedure Body

- **OR REPLACE:** This option indicates that if the procedure exists, it will be dropped and replaces with the new version created by the statement.

- **Procedure Name:** Name of the Procedure.

- **Parameters:** Name of the PL/SQL variables whose value is passed to the procedure or populated by the calling environment, or both, depending on the mode begin used.

- **Mode:** Type of arguments: IN (default) or OUT or IN OUT.

IN: Specifies that a value for the argument must be specified when calling the procedure.

OUT: Specifies that the procedure pass a value, for this argument back to its calling environment after execution.

IN OUT: Specifies that a value for the argument must be specified when calling the procedure and that the procedure passes a value, for this argument back to its calling environment after execution.

- **Datatype:** Datatype of the argument or parameter. It can be any SQL or PL/SQL datatype. It can be of „%TYPE“, „%ROWTYPE“, or any scalar or composite datatype.

Example: Compute the addition of two numbers using procedure in PL/SQL?

Step 1: Create a Procedure.

```
SQL> set serveroutput on;
SQL> create or replace procedure student123_proc(x number, y number)
         IS num number;
           begin
              num:= x + y;
                  dbms_output.put_line ('Sum of Two Numbers is: ' || num);
           end student123_proc;
           /
```

Procedure created.

Step 2: Method 1: Calling a Procedure: A Procedure can be called from any PL/SQL program by giving its names following by the parameters.

Syntax: Procedure Name (argument1, argument2, ……,);

```
SQL> begin
           student123_proc(20,38);      // Calling
       Procedure. end;
        /
```

Output: Sum of Two Numbers is: 58
PL/SQL procedure successfully completed.
Method 2:

```
SQL> declare
              num1 number:= 25;
              num2 number:= 38;
       begin                                         student123_proc(num1, num2);

       end;
       /
```

Output: Sum of Two Numbers is: 63

PL/SQL procedure successfully completed.

NOTE: Procedure can also be called / invoked from the SQL prompt using the „Execute“ command.

Syntax: SQL> Execute <Procedure Name (argument1, argument2, ……,);

```
SQL> execute student123_proc(35,74);
```

Output: Sum of Two Numbers is: 109

PL/SQL procedure successfully completed.

Step 3: Dropping a Procedure: To drop procedure using „Drop Procedure" command.

Syntax: DROP PROCEUDRE <Procedure Name>;

SQL> drop procedure student123_proc;

Procedure dropped.

(II). **Function:** A Function is similar to Procedure except that the function have a must „Return" clause one only value to the calling environment. A function must have a „Return" clause in the header section and at least one „Return" statement in the execution section.

Syntax of Create Function Statement:

Function Specification

```
CREATE [OR REPLACE] FUNCTION <Function
Name>
[(Parameter 1 [mode1] datatype 1, Parameter 2
[mode2] datatype 2 …….)] RETURN Datatype
{IS / AS}
[Local
    vari
    able
    s
    decl
    arati
    on]
    BEG
    IN
      PL/SQL Executable Statements

    [EXCEPTION
        Exception Handlers] END [<Function
    Name>];
```

Function Body

- **OR REPLACE:** This option indicates that if the function exists, it will be dropped and replaces with the new version created by the statement.

- **Function Name:** Name of the Function.

- **Parameters:** Name of the PL/SQL variables whose value is passed to the function or populated by the calling environment, or both, depending on the mode begin used.

- **Mode:** Type of arguments: IN (default) or OUT or IN OUT.

  IN: Specifies that a value for the argument must be specified when calling the function.

  OUT: Specifies that the function pass a value, for this argument back to its calling environment after execution.

  IN OUT: Specifies that a value for the argument must be specified when calling the function and that the function passes a value, for this argument back to its calling environment after execution.

- **Datatype:** Datatype of the argument or parameter. It can be any SQL or PL/SQL datatype. It can be of „%TYPE“, „%ROWTYPE“, or any scalar or composite datatype.

- **Return datatype:** Datatype of the RETURN value that must be return by the function.

Example: Compute the addition of two numbers using function in PL/SQL?

Step 1: Create a Function.

```
SQL> set serveroutput on;

SQL> create or replace function student123_fun (x number, y
        number) return number IS
        ret
    number; begin
        ret:= x + y;
        return (ret);
    end;
    /
```

Function created.

Step 2: Function Execution / Calling a Function: A function may accept one or many parameters, but must return a single value. A function as part of PL/SQL expressions, using variables to hold the returned value.

```
SQL> declare
        num1
        number:=10;
        num2
        number:=37;
        student123
        number;
    begin
           student123:= student123_fun(num1, num2);
           dbms_output.put_line('The Sum of Two numbers results is: '|| student123);
    end;
    /
```

PL/SQL procedure successfully completed.

```
SQL> set serveroutput on;

SQL> /
```

Output: The Sum of Two numbers results is: 47

PL/SQL procedure successfully completed.

Step 3: Dropping a Procedure: To drop function using „Drop Function" command.

Syntax: DROP FUNCTION <Function Name>;

```
SQL> drop function student123_proc;
```

Function dropped.

(Q) Define Formal versus Actual / Host parameters / variables in Subprogram?

Formal Parameters: The Formal parameters are variables declared in the parameter list of a subprogram specification.

Actual / Host Parameters: The Actual / Host parameters are variables or expression referenced in the parameter list of a subprogram call.

```
SQL> create or replace function student123_fun (x number, y
        number) return number IS
        ret number;
     begin                                                  Formal Parameters
        ret:= x + y; return (ret);
     end;
     /
```

Function created.

```
SQL> declare
        num1  number:=10;  num2  number:=37;
        student123 number;
     begin                                          Actual / Host Parameters

           student123:= student123_fun(num1, num2);
           dbms_output.put_line ('The Sum of Two numbers results is: '|| student123);
     end;
     /
```

Output: The Sum of Two numbers results is: 47

PL/SQL function successfully completed.

(Q) Define various Arguments Modes in Subprogram?

Arguments modes are used to define the behavior of formal parameters. There are three argument modes to be used with any subprograms (Procedure or Function).

1. **IN:** This mode passes a constant value from the calling environment into the subprogram (Procedure or Function). IN mode is the default argument mode for a subprogram.

Example: Compute the addition of two numbers using procedures of IN mode arguments?

```
SQL> create or replace procedure student123_inmode (x IN number, y IN number)

        AS
           num number;
        begin
           num:= x + y;
           dbms_output.put_line ('The Sum of Two Numbers is: ' || num);
        end student123_inmode;
       /
```

Procedure created.

```
SQL> declare
            num1 number:= 25;
            num2 number:= 53;
      begin
         student123_inmode (num1,
      num2); end;
   /
```

Output: The Sum of Two Numbers is: 78

PL/SQL procedure successfully completed.

2. **OUT:** This mode passes a value from the subprogram (procedure) to the calling environment.

Example: Compute the addition and multiplication of two numbers using function of OUT mode arguments?

```
SQL> create or replace function fun_out (x OUT number)
         return number
         IS
            a number:= 27;
            b number:= 61;
         begin
            x:= a + b;
            return (a * b);
         end;
         /
```

Function created.

```
SQL> declare
         mul
         number;
         add
         number;
      begin
         mul:= fun_out(add);
         dbms_output.put_line („The Multiplication of A and B value is: ' || mul);
         dbms_output.put_line ('The Addition of A and B value is: ' || add);
      end;
      /
```

Output:

```
The Multiplication of A and B value is: 1647
'The Addition of A and B value is: 88
```

PL/SQL procedure successfully completed.

3. **IN OUT:** This mode passes a value from the calling environment into the subprogram (procedure or function) and a possibly different value from the subprogram (procedure or function) back to the environment using the dame parameter.

Example: Illustrate the IN OUT argument using function in PL/SQL?

```
SQL> create or replace function in_out (x IN OUT number)
          return number
          IS
             y number:=245;
          begin
             x:= x + y;
             return(x - y);
          end;
          /
```

Function created.

```
SQL> declare
             add number:=45;
             sub number;
          begin
             sub:=in_out(add);
                 dbms_output.put_line ('The Addition of X and Y is: ' || add);
                 dbms_output.put_line („Calling Back the same parameter is: ' || sub);
          end;
          /
Output:    The Addition of X and Y is: 290
             Calling Back the same parameter is: 45
```

PL/SQL procedure successfully completed.

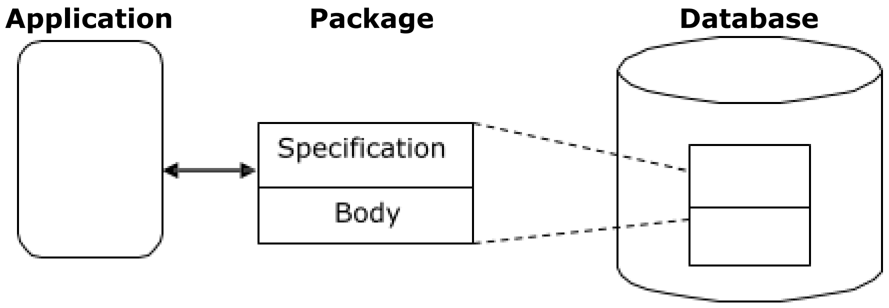
(Q). Explain about Packages in PL/SQL?

Package: A package can be defined ad a collection of related program objects, such as procedures, functions, and associated cursors and variables together as a single unit in the database.

Parts of Package: A package has two parts:

I. Package Specification
II. Package Body.

Package Interface:



(I). **Package Specification:** This contains the list of variables, constants, functions, procedure names which are the part of the package. PL/SQL specification are public declaration and visible to a program.

Syntax for Package Specification:

```
Create [or Replace] PACKAGE <Package Name> {IS | AS}
        [Variable declaration;]
        [Constant declaration;]
        [Exception declaration;]
        [Cursor specification;]
        [Procedure <Procedure Name> [(argument {IN | OUT | IN OUT} datatype)]]
            End [Procedure Name];
        [Function <Function Name> [(argument IN datatype)] Return datatype]
            End [Function Name];
```

Example: Create a package on arithmetical operations.

Step 1: Creating a Package Specification named as „arith“.

```
SQL> create or replace package arithmetic IS
            procedure addnum (a number, b number);
            procedure subnum (a number, b number);
            procedure multnum (a number, b number);
            procedure divnum (a number, b number);
            end;
        /
```

Package created.

(II). **Package Body:** This contains the actual PL/SQL statement code implementing the logics of procedure and function which are already before declared in package specification.

Syntax for Package Body:

```
Create [or Replace] PACKAGE BODY <Package Name> {IS | AS}
    [Procedure <Procedure Name> [(argument {IN | OUT | IN OUT} datatype)]]
      End [Procedure Name];
    [Function <Function Name> [(argument IN datatype)] Return datatype]
      End [Function Name];
    Private variable declaration;
    Private type definition;
    Cursor definition
      [begin
        Executable statements;
      [Exception
        Exception handlers]]
End <Package Name>;
```

Example: Body of the Package for arithmetic operation?

Step 2: Create a package body.

```
SQL> create or replace package body arithmetic is
    procedure addnum (a number, b number) is c number;
    begin
      c:= a + b;
      dbms_output.put_line ('The Sum of Two Numbers is: ' || c);
    end;
  procedure subnum (a number, b number) is c number;
    begin
      c:= a - b;
      dbms_output.put_line ('The Subtraction of Two Numbers is: ' || c);
    end;
  procedure multnum (a number, b number) is c number;
    begin
      c:= a * b;
      dbms_output.put_line ('The Multiplication of Two Numbers is: ' || c);
    end;
  procedure divnum (a number, b number) is c number;
    begin
      c:= a / b;
      dbms_output.put_line ('The Division of Two Numbers is: ' || c);
    end;
  end arithmetic;
 /
```

Package body created.

PL/SQL Package Calling: Now we have a one package „arithmetic", to call package define procedure also pass the parameters and get the result.

Step 3: Calling a package in PL/SQL block:

SQL> ed far_pack.sql;

```
SQL> declare
         n1
         number;
         n2
         number;
       begin
         n1:=&n1;
          n2:=&n2;
            arithmetic.addnum(n1,n2);
            arithmetic.subnum(n1,n2);
            arithmetic.multnum(n1,n2);
            arithmetic.divnum(n1,n2);
       end;
    /
```

Execute Package: Now execute the above package program

```
SQL> @ far_pack.sql;

         Enter value for n1: 4
         old    5:     n1:=&n1;
         new 5:      n1:=4;

         Enter value for n2: 5
         old    6:      n2:=&n2;
         new 6:      n2:=5;

         The Sum of Two Numbers is: 9

         The Subtraction of Two Numbers is: -1

         The Multiplication of Two Numbers is: 20

         The Division of Two Numbers is: 8

       PL/SQL procedure successfully completed.
```

Drop Package: If we want to remove package use „Drop Package" command.

Syntax: drop package <package name>;

Example: drop package arithmetic;

(Q) Discuss about Exception Handling in PL/SQL?

Exception Handling: PL/SQL makes it easy to detect and process error conditions called „Exception". An exception is an error condition is raised. That is, normal execution stops and control transfers to the exception-handling part of PL/SQL block.

For example if try to division a number by zero, the pre defined execution ZREO-DIVIDE is raised automatically. User-define exceptions must be raised explicitly by RASIE statement.

Exceptions are two types:

1. Internally / Pre-defined Exception.
2. User-defined Exception.

1. Internally /Pre-define Exception: An internal exception is raised implicitly whenever a PL/SQL program violates an Oracle rule or exceeds a system dependent limit. Every Oracle error has a number, but exceptions must be handled by named.

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

2. **User-define Exception:** PL/SQL user define exception to make own exception. User-define exception must be declare werself and RAISE statement to raise explicitly.

a. **Declare Exception:** we must have to declare user define exception name in „Declare" block.
b. **RAISE Exception:** RAISE statement to raised defined exception name and control transfer to an „Exception" block.
c. **Implement Exception:** In PL/SQL Exception block add When condition to implement user define action.

Syntax for User-define Exception:

```
SQL> Declare
        User-define exception_name EXCEPTION;
     Begin
        Statements;
        If condition then
           RAISE user-define exception_name;
        End if;
     EXCEPTION
           WHEN user-define exception_name THEN
              Statements (action) will be taken;
     End;
```

Example: Implement the user-define exceptions?

SQL> ed excep.sql;

```
    declare
        m number(4);
        myerror exception;
    begin
        select comm into m from empstudent123 where
    empno=7839; if m is null then
        raise myerror;
    end if;
    exception
        when no_data_found then
            dbms_output.put_line('Error of Data');
        when too_many_rows then
            dbms_output.put_line('Error of too many rows');
        when myerror then
            dbms_output.put_line('Error Found Null');
    end;
  /
```

Output 1: SQL> @ excep.sql;

```
Error Found Null
```

PL/SQL procedure successfully completed.

Output 2: SQL> select comm into m from empstudent123 where empno=4000;

SQL> @ excep.sql;

```
Error of Data
```

PL/SQL procedure successfully completed.

********* (Q) What is Triggers? Explain about Triggers?**

Triggers:

- "If a condition started with in the trigger is met, then a prescribed action is taken".
- A Trigger is a stored procedure that implicitly executed, when an Insert, Update or Delete (DML statement) is issued against the associated table.
- We can make a Trigger to Fire (Executed) only for DML statements (Insert, Update, and Deleted).
- The Trigger code is stored in the database and runs automatically whenever the Triggering event such as an update occurs.
- Since Triggers are stored and executed in the database.
- They execute against all applications that access the database.
- We can delete the Trigger by using Drop command.

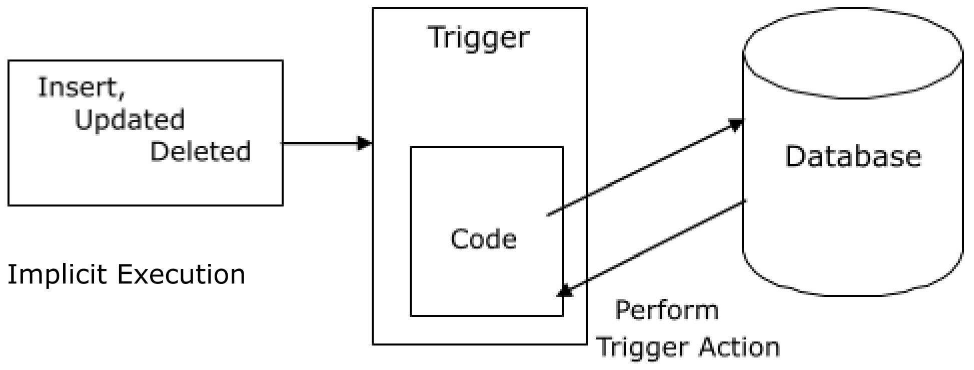
Parts of Triggers: There are mainly 3 parts.

1. Trigger Statement (Event)
2. Trigger Restriction (Conditions)
3. Trigger Body (Action)

1. **Trigger Statement:** The Trigger statement specified DML statement like Insert, Update, Delete and that causes a trigger to be fired.

2. **Trigger Restriction:** We can specify the condition inside trigger to when trigger is fire.

3. **Trigger Body:** When the trigger SQL statement is execute, trigger automatically call and PL/SQL trigger block execute.



Syntax for Creating Trigger:

SQL> CREATE [OR REPLACE] TRIGGER
<Trigger_Name> {BEFORE / AFTER / INSTEAD OF}
{INSERT / DELETE/ UPDATE} ON
<Table_Name>

Trigger Event

[FOR EACH ROW/ STATEMENT [WHEN
(Trigger_Name)]] <condition> DECLARE
Declaration of Variable/ constants;

Trigger Condition

BEGIN
Body of the Statement;

EXCEPTION
Exception of PL/SQL Block; END;

TriggerAction

Types of Trigger: Triggers are mainly divided into two parts based on usage. There are

1. Before / After
2. For each Row / Statement.

1. **Before / After:** The Before / After options can be used to specify when the trigger body should be fire with respect to trigger statement.

- ☐ If Before option is used then Oracle Engine Fires the Trigger before executing the statement.
- ☐ If After option is used then Oracle Engine Fires the Trigger After executing the Trigger statement.

2. for each Row/Statement:

- **Statement Level Trigger:** A Trigger which is created only for one record are called "Statement Level Trigger". In practice these are not used frequently by default. Every Trigger is statement level Trigger.

- **Row Level Trigger:** The Trigger which is created on the total table is known as "Row Level Trigger" by using Row Level Trigger. We can make a Trigger to fire on selected columns for more than once. When working with Row Level Triggers. We can use two qualifiers called "Old" & "New". The most commonly used Triggers are Row Level Triggers.

Example: we may use a Database Trigger to log the fact that an account receivable clerk has lowered the amount owing on an outstanding account.

Ex:
```
Create or replace trigger T1 before insert
    On bank for each row
     Begin
       dbms_output.put_line("Trigger Fired");
    End;
```

Ex:
```
create or replace trigger T2 before update
     On bank for each row
      Begin
   Update bank set balance= 1000;
           dbms_output.put_line("Update");
      End;
```

Enabling & Disabling Triggers: Triggers don"t affect all rows in the table. They affect only Transactions of the specified type. When the Triggers Enable. The Triggers will not affect any transactions created before to a Trigger by default. Triggers are enabled when it is created, but sometimes we may need to disable Triggers disabling Triggers during data loads improve the performances of the data load. In such a way the data load partially succeeded and the Triggers was executed for a portion of the data load records. It is also possible to since a trigger twice for the same transaction. To enable a trigger "Alter Trigger" command is used with the "enable" keyword.

Syntax for Enable / Disable Trigger:

```
SQL>    Alter table <table name> enable / disable all triggers;
```

Example: SQL> Alter table bank disable all trigger;

Dropping Trigger: If there are any unwanted Triggers in the data base. They must be removed from that the database, it is also called Dropping a Trigger. To drop any Trigger the following syntax can be used.

Syntax: SQL> drop trigger <trigger name>;

Example: SQL> drop trigger bank_person;
Trigger dropped.

Advantages of Triggers: Triggers can be used to makes the RDBMS to take some action. When a database related event has occurred the following advantages of Triggers.

- The Business rules can be stored in the database consistently with each and every updated operation.
- The reduce complexity of the application program that the database.

Disadvantages of Trigger:

- When the business rules are moved in to the database, setting up, the database becomes a more complex task.

- With the triggers, the business rules are hidden in the database and the application program can cause an enormous (plenty) amount of database activity.

Example: PL/SQL program to implement the Triggers operations.

Steps for doing trigger program:

(a). First create a table called person(don"t insert any values into created table).
(b). Write code for trigger and run the trigger program.
(c). Insert values into the created table after successfully completion of trigger program and see the trigger output.

Step 1: First create a table called person (don"t insert any values into created table).

SQL> create table person (id integer, name varchar2(20), dob date, primary key(id));

Table created.

Step 2: Write PL/SQL code for „Before Insert" Trigger.

```
SQL>  create or replace trigger person_insert_before
          before insert on person for each row
      begin
          dbms_output.put_line('Before Insert of '||:new.name);
     end;
   /
```
Trigger created.

Step 3: Insert values into the created table after successfully completion of trigger program.

```
SQL> insert into person values(1, 'student123', sysdate);

    Before Insert of student123

    1 row created.

SQL> set serveroutput on;
```

Step 4: View the Resulted Table.

```
SQL> select * from person;

      ID NAME           DOB
--------- ------------ ---------
        1   student123   31-Dec-2020
```

Step 5: Write the PL/SQL code for „After Insert“ Trigger.

```
SQL> create or replace trigger person_insert_after
              after insert on person for each row
           begin
              dbms_output.put_line('After Insert of '||:new.name);
           end;
        /
Trigger created.
```

Step 6: Insert values into the created table after successfully completion of trigger program.

```
SQL> insert into person values(2, 'Mohammad', sysdate);

        Before Insert of Mohammad
        After Insert of Mohammad

1 row created.
```

Step 7: View the Resulted Table.

```
SQL> select * from person;

      ID NAME                 DOB
--------- ------------- --------------
        1  student123       31-Dec-2020
        2  Mohammad         31-Dec-2020

SQL> insert into person values(3, 'Shaik', sysdate);
           Before Insert of Shaik
           After Insert of Shaik

        1 row created.
```

```
SQL> insert into person values(4, 'Muneer', sysdate);

Before Insert of Muneer
After Insert of Muneer

1 row created.
SQL> insert into person values(5, 'Pasha', sysdate);
Before Insert of Pasha
After Insert of Pasha
```

Step 8: View the Resulted Table.

```
SQL> select * from person;

      ID NAME                  DOB
--------- ------------       --------------
       1  student123          31-Dec-2020
       2  Mohammad            31-Dec-2020
       3  Shaik               31-Dec-2020
       4  Muneer              31-Dec-2020
       5  Pasha               31-Dec-2020
```

Step 9: Write PL/SQL code for „Before Update statement" trigger

```
SQL> create or replace trigger person_update_before
           before update on person
      begin
           dbms_output.put_line('Before Updating Some Persons');
     end;
       /
Trigger created.
```

Step 10: Perform Update command on „Person"

```
table. SQL> update person set dob= '01-Jan-2021';

Before Updating Some Persons

5 rows updated.
```

Step 11: View the Resulted Table.

```
SQL> select * from person;

      ID NAME                       DOB
--------- --------------------    -------------
       1  student123              01-Jan-2021
       2  Mohammad                01-Jan-2021
       3  Shaik                   01-Jan-2021
```

```
        4  Muneer                    01-Jan-2021
        5  Pasha                     01-Jan-2021
```

Step 12 : Write PL/SQL code „For each row before update“ trigger.

```
SQL> create or replace trigger person_update_before
          before update on person for each row
      begin
          dbms_output.put_line('Before Updating'||to_char(:old.dob,'hh:mi:ss')||'to'||
           to_char(:new.dob,'hh:mi:ss'));
     end;
     /
Trigger created.
```

Step 13: Perform Update command on „Person“

table. SQL> update person set dob= '02-Jan-2021';

```
Before Updating 12:00:00 to 12:00:00
Before Updating 12:00:00 to 12:00:00
Before Updating 12:00:00 to 12:00:00
Before Updating 12:00:00 to 12:00:00
Before Updating 12:00:00 to 12:00:00

5 rows updated.
```

Step 14: View the Resulted Table.

SQL> select * from person;

```
      ID NAME                   DOB
--------- --------------------   -------------
        1  student123          02-Jan-2021
        2  Mohammad            02-Jan-2021
        3  Shaik               02-Jan-2021
        4  Muneer              02-Jan-2021
        5  Pasha               02-Jan-2021
```

Step 15: Write PL/SQL code „For if statement“ trigger.

```
SQL> create or replace trigger person_biud
        before insert or update or delete on person for each row
      begin
         if inserting then
            dbms_output.put_line('Inserting Person: '||:new.name);
            else if updating then
                 dbms_output.put_line('Updating Person: '||:old.name||'to'||:new.name);
            else if deleting then
                dbms_output.put_line('Deleting Person: '||:old.name);
            end if;
         end if;
      end if;
   end;
   /
```

Trigger created.

Step 16: Insert values into the created table after successfully completion of trigger program.

```
SQL> insert into person values(6, 'Abu', '03-Jan-2021');

Inserting Person : Abu
Before Insert of Abu
After Insert of Abu
```

1 row created.

Step 17: View the Resulted Table.

```
SQL> select * from person;

     ID NAME                  DOB
--------- ------------      --------------
       1  student123         02-Jan-2021
       2  Mohammad           02-Jan-2021
       3  Shaik              02-Jan-2021
       4  Muneer             02-Jan-2021
       5  Pasha              02-Jan-2021
       6  Abu                03-Jan-2021
```

6 rows selected.

Step 18: Perform Update command on „Person" table.

```
SQL> update person set name= 'Abu Hashim' where name= 'Abu';
Updating Person: Abu to Abu Hashim
Before Updating 12:00:00 to 12:00:00.
```

1 row updated.

Step 19: View the Resulted Table.

```
SQL> select * from person;

     ID NAME                  DOB
--------- ------------      --------------
       1  student123         02-Jan-2021
       2  Mohammad           02-Jan-2021
       3  Shaik              02-Jan-2021
       4  Muneer             02-Jan-2021
       5  Pasha              02-Jan-2021
       6  Abu Hashim         03-Jan-2021
```

6 rows selected.

Step 20: Perform Delete command on „Person“

table. SQL> delete person where name= 'Shaik';

Deleting Person: Shaik

1 row deleted.

Step 21: View the Resulted Table.

SQL> select * from person;

```
     ID    NAME                  DOB
  --------- ------------       --------------
      1  student123         02-Jan-2021
      2  Mohammad           02-Jan-2021
      4  Muneer             02-Jan-2021
      5  Pasha              02-Jan-2021
      6  Abu Hashim         03-Jan-2021
```

****

Prepaired By:
Shaik Mohammad student123. MCA, M.Sc CS, IRPM. H.O.D
Department of Computer Science & Applications Mail ID:
student123shaik47@gmail.com
Web Portal: www.student123.com
Contact Number: 9491202987