Unit-3

Topics Covered: *Basics of SQL-DDL,DML,DCL,TCL , Structure Creation, alternation , Defining Constraints-Primary Key, Foreign Key, Unique, not null, check, IN operator , Functions-aggregation functions , Built-in Functions-numeric, date, string functions, string functions, Set operations, Sub Queries, correlated sub queries , Nested Queries, View , Transaction Control Commands , Commit, Rollback, Save point , PL/SQL Concepts- Cursors , Stored Procedure, Functions Triggers and Exception Handling .*

SQL | DDL, DQL, DML, DCL and TCL Commands

The SQL commands are mainly categorized into four categories as:

1. DDL – Data Definition Language
2. DQL - Data Query Language
3. DML – Data Manipulation Language
4. DCL – Data Control Language

DDL (Data Definition Language):

DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.DDL is a set of SQL commands used to create, modify, and delete database structures but not data. These commands are normally not used by a general user, who should be accessing the database via an application.

List of DDL commands:

- **CREATE**: This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).
  Syntax:
  CREATE DATABASE database_name;

  database_name: name of the database.
  Example Query:
  This query will create a new database in SQL and name the database as *my_database*.
  CREATE DATABASE my_database;

- **DROP**: This command is used to delete objects from the database.
  Syntax:
  DROP object object_name

Examples:
DROP TABLE table_name;
table_name: Name of the table to be deleted.

DROP DATABASE database_name;
database_name: Name of the database to be deleted.

- **ALTER:** This is used to alter the structure of the database. ALTER TABLE – ADD

ADD is used to add columns into the existing table. Sometimes we may require to add additional information, in that case we do not require to create the whole database again, ADD comes to our rescue.

Syntax:

```
 ALTER TABLE table_name
            ADD (Columnname_1  datatype,
            Columnname_2  datatype,
            …
            Columnname_n  datatype);
```

- **TRUNCATE:** This is used to remove all records from a table, including all spaces allocated for the records are removed. TRUNCATE statement is a Data Definition Language (DDL) operation that is used to mark the extents of a table for deallocation (empty for reuse). The result of this operation quickly removes all data from a table, typically bypassing a number of integrity enforcing mechanisms. It was officially introduced in the SQL:2008 standard.
  The TRUNCATE TABLE mytable statement is logically (though not physically) equivalent to the DELETE FROM mytable statement (without a WHERE clause).

  Syntax:

TRUNCATE TABLE table_name;
table_name**: Name of the table to be truncated.**
DATABASE name - student_data

- **RENAME:** This is used to rename an object existing in the database.

Syntax (Oracle, MySQL, MariaDB):

ALTER TABLE table_name

RENAME TO new_table_name;

DQL (Data Query Language):

DQL statements are used for performing queries on the data within schema objects. The purpose of the DQL Command is to get some schema relation based on the query passed to it. We can define DQL as follows it is a component of SQL statement that allows getting data from the database and imposing order upon it. It includes the SELECT statement. This command allows getting the data out of the database to perform operations with it. When a SELECT is fired against a table or tables the result is compiled into a further temporary table, which is displayed or perhaps received by the program i.e. a front-end.

List of DQL Commands:

- **SELECT:** It is used to retrieve data from the database. Select is the most commonly used statement in SQL. The SELECT Statement in SQL is used to retrieve or fetch data from a database. We can fetch either the entire table or according to some specified rules. The data returned is stored in a result table. This result table is also called result-set.
  Syntax:
  SELECT column1, column2 FROM table_name
  column1, column2**: names of the fields of the table**
  table_name: **from where we want to fetch**

DML (Data Manipulation Language):

The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. It is the component of the SQL statement that controls access to data and to the database. Basically, DCL statements are grouped with DML statements.

List of DML commands:

- **INSERT** : It is used to insert data into a table.

The INSERT INTO statement of SQL is used to insert a new row in a table. There are two ways of using INSERT INTO statement for inserting rows:

1. **Only values:** First method is to specify only the value of data to be inserted without the column names.

   INSERT INTO table_name VALUES (value1, value2, value3,…);
   table_name*: name of the table.*
   value1, value2,.. *: value of first column, second column,… for the new record*

2. **Column names and values both:** In the second method we will specify both the columns which we want to fill and their corresponding values as shown below:
   INSERT INTO table_name (column1, column2, column3,..) VALUES (value1, value2, value3,..);
   table_name*: name of the table.*
   column1*: name of first column, second column …*
   value1, value2, value3 *: value of first column, second column,… for the new record*

- **UPDATE:** It is used to update existing data within a table.
  The UPDATE statement in SQL is used to update the data of an existing table in database. We can update single columns as well as multiple columns using UPDATE statement as per our requirement.

  Basic Syntax
  UPDATE table_name SET column1 = value1, column2 = value2,...
  WHERE condition;

  table_name: name of the table
  column1: name of first , second, third column....
  value1: new value for first, second, third column....
  condition: condition to select the rows for which the
  values of columns needs to be updated.
  Example Queries
  Updating single column: Update the column NAME and set the value to 'PRATIK' in all the rows where Age is 20.
  UPDATE Student SET NAME = 'PRATIK' WHERE Age = 20;

- **DELETE** : It is used to delete records from a database table.
  The DELETE Statement in SQL is used to delete existing records from a table. We can delete a single record or multiple records depending on the condition we specify in the WHERE clause.

  Basic Syntax:
  DELETE FROM table_name WHERE some_condition;

  table_name: name of the table
  some_condition: condition to choose particular record.

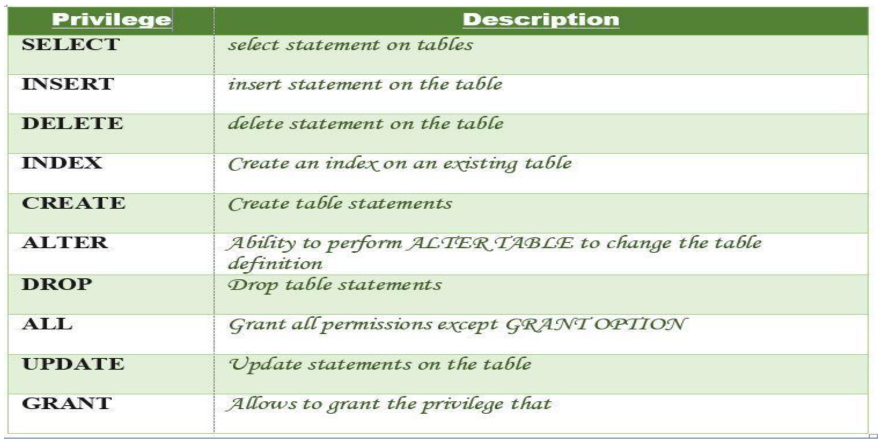
DCL (Data Control Language):
DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

List of DCL commands:

- **GRANT:** This command gives users access privileges to the database.

Syntax:
GRANT privileges_names ON object TO user;

Parameters Used:
privileges_name: These are the access rights or privileges granted to the user.
Object: It is the name of the database object to which permissions are being granted. In the case of granting privileges on a table, this would be the table name.
User: It is the name of the user to whom the privileges would be granted.
Privileges:
The privileges that can be granted to the users are listed below along with description:



Privilege	Description
SELECT	*select statement on tables*
INSERT	*insert statement on the table*
DELETE	*delete statement on the table*
INDEX	*Create an index on an existing table*
CREATE	*Create table statements*
ALTER	*Ability to perform ALTER TABLE to change the table definition*
DROP	*Drop table statements*
ALL	*Grant all permissions except GRANT OPTION*
UPDATE	*Update statements on the table*
GRANT	*Allows to grant the privilege that*

1. **Granting SELECT Privilege to a User in a Table**: To grant Select Privilege to a table named “users” where User Name is Amit, the following GRANT statement should be executed.
   GRANT SELECT ON Users TO'Amit'@'localhost;

2. **Granting more than one Privilege to a User in a Table**: To grant multiple Privileges to a user named “Amit” in a table “users”, the following GRANT statement should be executed.
   GRANT SELECT, INSERT, DELETE, UPDATE ON Users TO 'Amit'@'localhost

1. **Granting All the Privilege to a User in a Table**: To Grant all the privileges to a user named “Amit” in a table “users”, the following Grant statement should be executed.

GRANT ALL ON Users TO 'Amit'@'localhost;

2. **Granting a Privilege to all Users in a Table**: To Grant a specific privilege to all the users in a table "users", the following Grant statement should be executed.
   GRANT SELECT ON Users TO '*'@'localhost;

   In the above example the "*" symbol is used to grant select permission to all the users of the table "users".

- **REVOKE:** This command withdraws the user's access privileges given by using the GRANT command.
  Revoke command withdraw user privileges on database objects if any granted. It does operations opposite to the Grant command. When a privilege is revoked from a particular user U, then the privileges granted to all other users by user U will be revoked.

  Syntax:
  revoke privilege_name on object_name

  from {user_name | public | role_name}

  Example:
  grant insert,

  select on accounts to Ram

By the above command user ram has granted permissions on accounts database object like he can query or insert into accounts.

revoke insert,

select on accounts from Ram

By the above command user ram's permissions like query or insert on accounts database object has been removed.

List of TCL commands:

- **COMMIT:** Commits a Transaction.

If everything is in order with all statements within a single transaction, all changes are recorded together in the database is called **committed**. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

Syntax:
COMMIT;

- **ROLLBACK:** Rollbacks a transaction in case of any error occurs

  ROLLBACK: If any error occurs with any of the SQL grouped statements, all changes need to be aborted. The process of reversing changes is called **rollback**. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.
  Syntax:

  ROLLBACK;

- **SAVEPOINT:** Sets a save point within a transaction. It creates points within the groups of transactions in which to ROLLBACK. A SAVEPOINT is a point in a transaction in which you can roll the transaction back to a certain point without rolling back the entire transaction.

Syntax for Savepoint command:
SAVEPOINT SAVEPOINT_NAME;

This command is used only in the creation of SAVEPOINT among all the transactions.

Sample:

SAVEPOINT SP1;

//Savepoint created.

DELETE FROM Student WHERE AGE = 20;

//deleted

SAVEPOINT SP2;

//Savepoint created.

SQL Constraints

SQL constraints are used to specify rules for the data in a table. Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value

Example

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar (255) NOT NULL,
   FirstName varchar (255) NOT NULL,
   Age int
);
```

- UNIQUE - Ensures that all values in a column are different

Example

```
CREATE TABLE Persons (
   ID int NOT NULL UNIQUE,
   LastName varchar (255) NOT NULL,
   FirstName varchar (255),
   Age int
);
```

- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

Example

```
CREATE TABLE Persons (
   ID int NOT NULL PRIMARY KEY,
   LastName varchar (255) NOT NULL,
    FirstName varchar (255),
    Age int
);
```

- FOREIGN KEY - Prevents actions that would destroy links between tables

Example

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

- CHECK - Ensures that the values in a column satisfies a specific condition

Example

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar (255) NOT NULL,
    FirstName varchar (255),
    Age int,
    CHECK (Age>=18)
);
```

- DEFAULT - Sets a default value for a column if no value is specified

Example

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar (255) NOT NULL,
    FirstName varchar (255),
    Age int,
    City varchar (255) DEFAULT 'Sandnes'
);
```

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE ():

```
CREATE TABLE Orders (
   ID int NOT NULL,
   OrderNumber int NOT NULL,
   OrderDate date DEFAULT GETDATE ()
);
```

- CREATE INDEX - Used to create and retrieve data from the database very quickly

Example

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

Aggregate functions in SQL

In database management an aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning.

Various Aggregate Functions

1) Count ()

2) Sum ()

3) Avg ()

4) Min ()

5) Max ()

Example: Table Employee

```
Id    Name    Salary
-----------------------
1     A       80
2     B       40
3     C       60
```

4	D	70
5	E	60
6	F	Null

Count ():
Count (*): Returns total number of records .i.e. 6.
Count (salary): Return number of Non Null values over the column salary. i.e.5.
Count (Distinct Salary): Return number of distinct Non Null values over the column salary .i.e. 4

Sum ():

Sum (salary): Sum all Non Null values of Column salary i.e., 310
sum (Distinct salary): Sum of all distinct Non-Null values i.e., 250.

Avg ():
Avg (salary) = Sum (salary) / count (salary) = 310/5
Avg (Distinct salary) = Sum (Distinct salary) / Count (Distinct Salary) = 250/4

Min ():
Min (salary): Minimum value in the salary column except NULL i.e., 40.

Max (salary): Maximum value in the salary i.e., 80.

SQL Built In String Functions

Function	**Description**
ASCII	Returns the ASCII value for the specific character
CHAR	Returns the character based on the ASCII code
CHARINDEX	Returns the position of a substring in a string

CONCAT	Adds two or more strings together
Concat with +	Adds two or more strings together
CONCAT_WS	Adds two or more strings together with a separator
DATALENGTH	Returns the number of bytes used to represent an expression
DIFFERENCE	Compares two SOUNDEX values, and returns an integer value
FORMAT	Formats a value with the specified format
LEFT	Extracts a number of characters from a string (starting from left)
LEN	Returns the length of a string
LOWER	Converts a string to lower-case
LTRIM	Removes leading spaces from a string
NCHAR	Returns the Unicode character based on the number code
PATINDEX	Returns the position of a pattern in a string
QUOTENAME	Returns a Unicode string with delimiters added to make the string a valid SQL Server delimited identifier

REPLACE	Replaces all occurrences of a substring within a string, with a new substring
REPLICATE	Repeats a string a specified number of times
REVERSE	Reverses a string and returns the result
RIGHT	Extracts a number of characters from a string (starting from right)
RTRIM	Removes trailing spaces from a string
SOUNDEX	Returns a four-character code to evaluate the similarity of two strings
SPACE	Returns a string of the specified number of space characters
STR	Returns a number as string
STUFF	Deletes a part of a string and then inserts another part into the string, starting at a specified position
SUBSTRING	Extracts some characters from a string
TRANSLATE	Returns the string from the first argument after the characters specified in the second argument are translated into the characters specified in the third argument.
TRIM	Removes leading and trailing spaces (or other specified characters) from a string

UNICODE	Returns the Unicode value for the first character of the input expression
UPPER	Converts a string to upper-case

SQL Maths/Numeric Functions

Function	Description
ABS	Returns the absolute value of a number
ACOS	Returns the arc cosine of a number
ASIN	Returns the arc sine of a number
ATAN	Returns the arc tangent of a number
ATN2	Returns the arc tangent of two numbers
AVG	Returns the average value of an expression
CEILING	Returns the smallest integer value that is >= a number
COUNT	Returns the number of records returned by a select query
COS	Returns the cosine of a number

COT	Returns the cotangent of a number
DEGREES	Converts a value in radians to degrees
EXP	Returns e raised to the power of a specified number
FLOOR	Returns the largest integer value that is <= to a number
LOG	Returns the natural logarithm of a number, or the logarithm of a number to a specified base
LOG10	Returns the natural logarithm of a number to base 10
MAX	Returns the maximum value in a set of values
MIN	Returns the minimum value in a set of values
PI	Returns the value of PI
POWER	Returns the value of a number raised to the power of another number
RADIANS	Converts a degree value into radians
RAND	Returns a random number
ROUND	Rounds a number to a specified number of decimal places
SIGN	Returns the sign of a number

SIN	Returns the sine of a number
SQRT	Returns the square root of a number
SQUARE	Returns the square of a number
SUM	Calculates the sum of a set of values
TAN	Returns the tangent of a number

SQL Date Functions

Function	**Description**
CURRENT_TIMESTAMP	Returns the current date and time
DATEADD	Adds a time/date interval to a date and then returns the date
DATEDIFF	Returns the difference between two dates
DATEFROMPARTS	Returns a date from the specified parts (year, month, and day values)
DATENAME	Returns a specified part of a date (as string)
DATEPART	Returns a specified part of a date (as integer)
DAY	Returns the day of the month for a specified date

GETDATE	Returns the current database system date and time
GETUTCDATE	Returns the current database system UTC date and time
ISDATE	Checks an expression and returns 1 if it is a valid date, otherwise 0
MONTH	Returns the month part for a specified date (a number from 1 to 12)
SYSDATETIME	Returns the date and time of the SQL Server
YEAR	Returns the year part for a specified date

SQL Server Advanced Functions

Function	**Description**
CAST	Converts a value (of any type) into a specified datatype
COALESCE	Returns the first non-null value in a list
CONVERT	Converts a value (of any type) into a specified datatype
CURRENT_USER	Returns the name of the current user in the SQL Server database
IIF	Returns a value if a condition is TRUE, or another value if a condition is FALSE

ISNULL	Return a specified value if the expression is NULL, otherwise return the expression
ISNUMERIC	Tests whether an expression is numeric
NULLIF	Returns NULL if two expressions are equal
SESSION_USER	Returns the name of the current user in the SQL Server database
SESSIONPROPERTY	Returns the session settings for a specified option
SYSTEM_USER	Returns the login name for the current user
USER_NAME	Returns the database user name based on the specified id

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

Types of Set Operation

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

1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.

- The union operation eliminates the duplicate rows from its result set.

Syntax

1. SELECT column_name FROM table1
2. UNION
3. SELECT column_name FROM table2;

2. **Union All**

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

Syntax:

1. SELECT column_name FROM table1
2. UNION ALL
3. SELECT column_name FROM table2;

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

Syntax

1. SELECT column_name FROM table1
2. INTERSECT
3. SELECT column_name FROM table2;

4. Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax:

1. SELECT column_name FROM table1
2. MINUS
3. SELECT column_name FROM table2;

Sub Queries, correlated sub queries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.

A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a **SELECT**, **UPDATE**, or **DELETE** statement.

```
SELECT column1, column2….

FROM table1 outer

WHERE column1 operator

                    (SELECT column1, column2

                     FROM table2

                     WHERE expr1 =

                               outer.expr2);
```

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

Nested Subqueries versus Correlated Subqueries:
With a normal nested subquery, the inner **SELECT** query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

EXAMPLE of Correlated Subqueries: Find all the employees who earn more than the average salary in their department.

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

Other uses of correlation are in **UPDATE** and **DELETE**

CORRELATED UPDATE:

```
UPDATE table1 alias1
 SET column = (SELECT expression
               FROM table2 alias2
               WHERE alias1.column =
                     alias2.column);
```

Use a correlated subquery to update rows in one table based on rows from another table.

CORRELATED DELETE:

```
DELETE FROM table1 alias1
 WHERE column1 operator
               (SELECT expression
                FROM table2 alias2
                WHERE alias1.column = alias2.column);
```

Use a correlated subquery to delete rows in one table based on the rows from another table.

QUERIES:

1. Find the product_no and description of non- moving products.
2. Find the customer name, address, city and pin code for the client who has placed order no "019001"
3. Find the client names who have placed order before the month of may 96.
4. Find out if product "1.44 Drive" is ordered by only client and print the client_no name to whom it was sold.
5. Find the names of client who have placed orders worth Rs.10000 or more.
6. Select the orders placed by 'Rahul Desai"
7. Select the names of persons who are in Mr. Pradeep's department and who have also worked on an inventory control system.
8. Select all the clients and the salesman in the city of Bombay.
9. Select salesman name in "Bombay" who has at least one client located at "Bombay"
10. Select the product_no, description, qty_on-hand, cost_price of non_moving items in the product_master table.

View

- Views in SQL are considered as a virtual table. A view also contains rows and columns.
- To create the view, we can select the fields from one or more tables present in the database.
- A view can either have specific rows based on certain condition or all the rows of a table.

Sample table:

Student_Detail

STU_ID	NAME	ADDRESS

1	Stephan	Delhi
2	Kathrin	Noida
3	David	Ghaziabad
4	Alina	Gurugram

Student_Marks

STU_ID	NAME	MARKS	AGE
1	Stephan	97	19
2	Kathrin	86	21
3	David	74	18
4	Alina	90	20
5	John	96	18

1. Creating view

A view can be created using the **CREATE VIEW** statement. We can create a view from a single table or multiple tables.

Syntax:

1. CREATE VIEW view_name AS
2. SELECT column1, column2.....
3. FROM table_name
4. WHERE condition;

2. Creating View from a single table

In this example, we create a View named DetailsView from the table Student_Detail.

Query:

1. CREATE VIEW DetailsView AS
2. SELECT NAME, ADDRESS
3. FROM Student_Details
4. WHERE STU_ID < 4;

Just like table query, we can query the view to view the data.

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS
Stephan	Delhi
Kathrin	Noida
David	Ghaziabad

3. Creating View from multiple tables

View from multiple tables can be created by simply include multiple tables in the SELECT statement. In the given example, a view is created named MarksView from two tables Student_Detail and Student_Marks.

Query:

1. CREATE VIEW MarksView AS
2. SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.MARKS

3. FROM Student_Detail, Student_Marks
4. WHERE Student_Detail.NAME = Student_Marks.NAME;

To display data of View MarksView:

SELECT * FROM MarksView;

NAME	ADDRESS	MARKS
Stephan	Delhi	97
Kathrin	Noida	86
David	Ghaziabad	74
Alina	Gurugram	90

4. Deleting View

A view can be deleted using the Drop View statement.

Syntax

1. DROP VIEW view_name;

PL/SQL Concepts

- PL/SQL is a block structured language that can have multiple blocks in it.
- The programs of PL/SQL are logical blocks that can contain any number of nested sub-blocks.
- Pl/SQL stands for "Procedural Language extension of SQL" that is used in Oracle. PL/SQL is integrated with Oracle database (since version 7).
- The functionalities of PL/SQL usually extended after each release of Oracle database. Although PL/SQL is closely integrated with SQL language, yet it adds some programming constraints that are not available in SQL. With PL/SQL, you can use SQL statements to manipulate Oracle data and flow of control statements to process the data.

- The PL/SQL is known for its combination of data manipulating power of SQL with data processing power of procedural languages. It inherits the robustness, security, and portability of the Oracle Database.
- PL/SQL is not case sensitive so you are free to use lower case letters or upper case letters except within string and character literals.
- A line of PL/SQL text contains groups of characters known as lexical units. It can be classified as follows:

o Delimiters
o Identifiers
o Literals
o Comments

PL/SQL Concepts- Stored Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

o **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
o **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

How to pass parameters in procedure:

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

NOTE: A procedure may or may not return any value.

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(parameter [,parameter]) ]
 IS
[declaration_section]
  BEGIN
executable_section
[EXCEPTION    exception_section]
END [procedure_name];
```

Create procedure example

In this example, we are going to insert record in user table. So you need to create user table first.

Table creation:

```
Create table user
(Id        number    (10) primary key,
Name      varchar2 (100));
```

Now write the procedure code to insert record in user table.

Procedure Code:

```
Create or replace procedure Insertuser (Id IN NUMBER, Name IN VARCHAR2)
 Is/As
Begin
Insert into user values (id, name);
End;
/
```

Output:

```
Procedure Created
```

PL/SQL program to call procedure

Let's see the code to call above created procedure.

```
EXECUTE Insertuser(101,’Rahul’);
```

```
 BEGIN
Insertuser (101,'Rahul');
dbms_output.put_line ('record inserted successfully');
END;
   /
```

Now, see the "USER" table, you will see one record is inserted.

ID	Name
101	Rahul

PL/SQL Drop Procedure

Syntax for drop procedure

DROP PROCEDURE procedure_name;

Sample Procedures

```
CREATE OR REPLACE PROCEDURE Greetings //(Standalone Procedure)
AS
BEGIN
 dbms_output.put_line ('Hello World!');
END;
/
```

Executing a Standalone Procedure
A standalone procedure can be called in two ways:

- Using the EXECUTE keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named 'Greetings' can be called with the EXECUTE keyword as:

EXECUTE greetings;

The above call would display:

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block:

```
BEGIN
Greetings;
END;
/
```

The above call would display:

Hello World

PL/SQL procedure successfully completed.

PL/SQL Concepts- Functions

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax to create a function:

1. **CREATE** [OR REPLACE] **FUNCTION** function_name [parameters]
2. [(parameter_name [IN | **OUT** | IN **OUT**] type [, ...])]
3. **RETURN** return_datatype
4. {**IS** | **AS**}
5. **BEGIN**
6. < function_body >
7. **END** [function_name];

Here:

- **Function_name:** specifies the name of the function.
- **[OR REPLACE]** option allows modifying an existing function.
- The **optional parameter list** contains name, mode and types of the parameters.
- **IN** represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

The function must contain a return statement.

- RETURN clause specifies that data type you are going to return from the function.
- Function_body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

PL/SQL Function Example

```
Create or replace function adder (n1 in number, n2 in number)
Return number
Is
n3 number (8);
Begin
n3:=n1+n2;
Return n3;
End;
 /
```

Now write another program to **call the function.**

```
DECLARE
 n3 number(2);
BEGIN
 n3:= adder (11, 22);
dbms_output.put_line ('Addition is: ' || n3);
END;
 /
```

Output:

```
Addition is: 33
```

```
Statement processed.
0.05 seconds
```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
   a number;
   b number;
   c number;
CREATE OR REPLACE FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Maximum of (23, 45): 45

PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

```
n! = n*(n-1)!
  = n*(n-1)*(n-2)!
    ...
  = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively −

```
DECLARE
  num number;
  factorial number;

FUNCTION fact(x number)
RETURN number
IS
  f number;
BEGIN
  IF x=0 THEN
    f := 1;
  ELSE
    f := x * fact(x-1);
  END IF;
RETURN f;
END;

BEGIN
  num:= 6;
  factorial := fact(num);
  dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

PL/SQL Concepts- Cursors

When an SQL statement is processed, Oracle creates a memory area known as context area. A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

A cursor is used to refer to a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit Cursors
- Explicit Cursors

1) PL/SQL Implicit Cursors

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement. These are created by default to process the statements when DML statements like INSERT, UPDATE, and DELETE etc. are executed.

PL/SQL Implicit Cursor Example

Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000

3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

Create procedure:

```
DECLARE
  Total_rows number (2);
BEGIN
  UPDATE customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line ('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line (total_rows || ' customers updated ');
  END IF;
END;
/
```

Output:

```
6 customers updated
PL/SQL procedure successfully completed.
```

Now, if you check the records in customer table, you will find that the rows are updated.

Select * **from** customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	25000
2	Suresh	22	Kanpur	27000
3	Mahesh	24	Ghaziabad	29000
4	Chandan	25	Noida	31000
5	Alex	21	Paris	33000
6	Sunita	20	Delhi	35000

2) PL/SQL Explicit Cursors

The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors should be defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Following is the syntax to create an explicit cursor:

CURSOR cursor_name **IS** select_statement;

Steps:

You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

1) Declare the cursor:

It defines the cursor with a name and the associated SELECT statement.

Syntax for explicit cursor decleration

CURSOR name IS
SELECT statement;

2) Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

Syntax for cursor open:

OPEN cursor_name;

3) Fetch the cursor:

It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:

Syntax for cursor Fetch:

FETCH cursor_name **INTO** variable_list;

4) Close the cursor:

It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

Syntax for cursor Close:

Close cursor_name;

PL/SQL Explicit Cursor Example

Explicit cursors are defined by programmers to gain more control over the context area. It is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

In this example, we are using the already created CUSTOMERS table.

Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create procedure:

Execute the following program to retrieve the customer name and address.

```
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
    FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
```

```
END;
/
```

Output:

```
1 Ramesh Allahabad
2 Suresh Kanpur
3 Mahesh Ghaziabad
4 Chandan Noida
5 Alex Paris
6 Sunita Delhi

PL/SQL procedure successfully completed.
```

PL/SQL Concepts- Exceptional Handling

What is Exception

An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two type of exceptions:

- System-defined Exceptions
- User-defined Exceptions

PL/SQL Exception Handling
Syntax for exception handling:

Following is a general syntax for exception handling:

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
```

6. <exception handling goes here >
7. **WHEN** exception1 **THEN**
8. exception1-handling-statements
9. **WHEN** exception2 **THEN**
10. exception2-handling-statements
11. **WHEN** exception3 **THEN**
12. exception3-handling-statements
13. ....
14. **WHEN** others **THEN**
15. exception3-handling-statements
16. **END**;

Example of exception handling

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT* FROM COUSTOMERS;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

1. **DECLARE**
2. c_id customers.id%type := 8;

```
   c_name  customers.name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
 DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

After the execution of above code at SQL Prompt, it produces the following result:

```
No such customer!
PL/SQL procedure successfully completed.
```

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

PL/SQL Concepts- Triggers

Trigger is invoked by Oracle engine automatically whenever a specified event occurs. Trigger is stored into database and invoked repeatedly, when specific condition match. Triggers are stored programs, which are automatically executed or fired when some event occurs.

Triggers are written to be executed in response to any of the following events.

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Advantages of Triggers

These are the following advantages of Triggers:

- Trigger generates some derived column values automatically
- Enforces referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Syntax for creating trigger:

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

Here,

- o CREATE [OR REPLACE] TRIGGER trigger_name: It creates or replaces an existing trigger with the trigger_name.
- o {BEFORE | AFTER | INSTEAD OF}: This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- o {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- o [OF col_name]: This specifies the column name that would be updated.
- o [ON table_name]: This specifies the name of the table associated with the trigger.
- o [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- o [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- o WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

PL/SQL Trigger Example

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

Create table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000

5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create trigger:

Following trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

After the execution of the above code at SQL Prompt, it produces the following result.

```
Trigger created.
```

Check the salary difference by procedure:

Use the following code to get the old salary, new salary and salary difference after the trigger created.

```
DECLARE
```

```
  Total_rows number (2);
BEGIN
  UPDATE customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line ('no customers updated');
  ELSIF sql%found THEN
    total_rows:= sql%rowcount;
    dbms_output.put_line (total_rows || ' customers updated ');
  END IF;
END;
/
```

NOTE: sql%notfound is an attribute has a Boolean value that returns TRUE if no rows were affected by an INSERT, UPDATE, or DELETE statement, or if a SELECT INTO statement did not retrieve a row.

Output:

```
Old salary: 20000
New salary: 25000
Salary difference: 5000
Old salary: 22000
New salary: 27000
Salary difference: 5000
Old salary: 24000
New salary: 29000
Salary difference: 5000
Old salary: 26000
New salary: 31000
Salary difference: 5000
Old salary: 28000
New salary: 33000
Salary difference: 5000
```

```
Old salary: 30000
New salary: 35000
Salary difference: 5000
6 customers updated
```

Note: As many times you executed this code, the old and new salary is incremented by 5000 and hence the salary difference is always 5000.

The following table specifies the status of the cursor with each of its attribute.

Attribute	Description
%FOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE.
%NOTFOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND.
%ISOPEN	It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.
%ROWCOUNT	It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement.

PL/SQL Exception Handling

What is Exception?

An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two types of exceptions:

- o System-defined Exceptions
- o User-defined Exceptions

Syntax for exception handling:

Following is a general syntax for exception handling:

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
     exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN exception3 THEN
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

Example of exception handling

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT* FROM COUSTOMERS;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000

4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

```
DECLARE
   c_id customers.id%type:= 8;
   c_name customers.name%type;
   c_addr customers.address%type;
BEGIN
   SELECT name, address INTO  c_name, c_addr
   FROM customers
  WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
 DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line ('No such customer!');
   WHEN others THEN
      dbms_output.put_line ('Error!');
END;
/
```

After the execution of above code at SQL Prompt, it produces the following result:

```
No such customer!
PL/SQL procedure successfully completed.
```

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

Following is a general syntax for exception handling:

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
     exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN exception3 THEN
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

Example of exception handling

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT* FROM CUSTOMERS;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000

2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

```
DECLARE
  c_id customers.id%type:= 8;
  c_name customers.name%type;
  c_addr customers.address%type;
BEGIN
  SELECT  name, address INTO  c_name, c_addr
  FROM customers
  WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
 DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line ('No such customer!');
  WHEN others THEN
    dbms_output.put_line ('Error!');
END;
/
```

After the execution of above code at SQL Prompt, it produces the following result:

```
No such customer!
PL/SQL procedure successfully completed
```

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

If you use the id defined in the above table (i.e. 1 to 6), you will get a certain result. For a demo example: here, we are using the id 5.

```
DECLARE
  c_id customers.id%type := 5;
  c_name  customers.name%type;
  c_addr customers.address%type;
BEGIN
  SELECT  name, address INTO  c_name, c_addr
  FROM customers
  WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
 DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

After the execution of above code at SQL prompt, you will get the following result:

```
Name: alex
Address: paris
PL/SQL procedure successfully completed.
```

Raising Exceptions

In the case of any internal database error, exceptions are raised by the database server automatically. But it can also be raised explicitly by programmer by using command RAISE.

Syntax for raising an exception:

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

PL/SQL User-defined Exceptions

PL/SQL facilitates their users to define their own exceptions according to the need of the program. A user-defined exception can be raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

Syntax for user define exceptions

```
DECLARE
my-exception EXCEPTION;
```

PL/SQL Pre-defined Exceptions

There are many pre-defined exception in PL/SQL which are executed when any database rule is violated by the programs.

For example: NO_DATA_FOUND is a pre-defined exception which is raised when a SELECT INTO statement returns no rows.

Following is a list of some important pre-defined exceptions:

Exception	**Oracle Error**	**SQL Code**	**Description**
ACCESS_INTO_NULL	06530	-6530	It is raised when a NULL object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the "WHEN" clauses of a CASE statement is selected, and there is no else clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than exists to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when s program attempts to log on to the database with an invalid username or password.

NO_DATA_FOUND	01403	+100	It is raised when a select into statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.