# PRACTICAL 1: INTRODUCTION TO SQL AND INSTALLATION OF ORACLE.

## Introduction to SQL:

SQL (Structured Query Language) is a powerful programming language used for managing and manipulating relational databases. SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987. SQL can execute queries against a database. It is the standard language for interacting with relational database management systems (RDBMS) such as Oracle, MySQL, Microsoft SQL Server, and PostgreSQL. SQL allows you to create, update, retrieve, and manage data in a database. It's used for tasks like creating tables, inserting data, querying data, and modifying data, making it an essential skill for database administrators, data analysts, and software developers.

## Installation of Oracle Database:

To install Oracle Database, you'll need to follow these general steps. Please note that Oracle Database installation and setup can be quite complex, so it's essential to refer to Oracle's official documentation for specific details and system requirements:

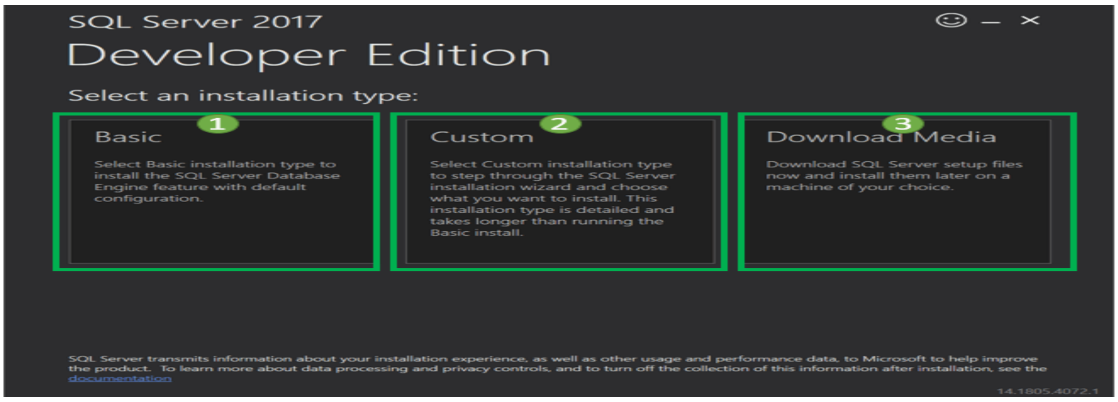
1. **System Requirements:** Ensure that your system meets the Oracle Database system requirements. These requirements can vary depending on the specific version of Oracle Database you intend to install.
2. **Download the Software:** Visit Oracle's official website and download the Oracle Database software. You may need to sign in or create an Oracle account to access the downloads.
3. **Install Oracle Database Software:** Run the installer and follow the installation wizard's instructions. You'll be prompted to provide information such as the installation location, system credentials, and database configurations.
4. **Choose Installation Type:** Oracle Database offers different installation options, including Express Edition (XE), Standard Edition, and Enterprise Edition. Choose the edition that suits your needs.
5. **Configuration:** During the installation, you'll be asked to configure various settings, such as the database name, passwords for administrative users (like SYS and SYSTEM), and other options related to your specific use case.

6. **Post-Installation Tasks:** After the installation is complete, you may need to perform post-installation tasks, such as creating user accounts, configuring network settings, and managing database parameters.
7. **Start and Access the Database:** Once the installation and configuration are complete, start the Oracle Database service. You can access the database using Oracle SQL*Plus or SQL Developer, which are Oracle's command-line and graphical user interface tools for managing databases.
8. **Connect and Use the Database:** You can connect to the Oracle Database and start writing SQL queries to create tables, insert data, and perform other database operations.

## Steps to install SQL:

1. **<u>Open the .exe file</u>**

   Double click on “SQLServer2017-SSEI-Dev.exe”. Below screen will appear with three options: Basic, Custom and Download files.



SQL Server 2017

Developer Edition

Select an installation type:

Basic

Select Basic installation type to install the SQL Server Database Engine feature with default configuration.

Custom

Select Custom installation type to step through the SQL Server installation wizard and choose what you want to install. This installation type is detailed and takes longer than running the Basic install.

Download Media

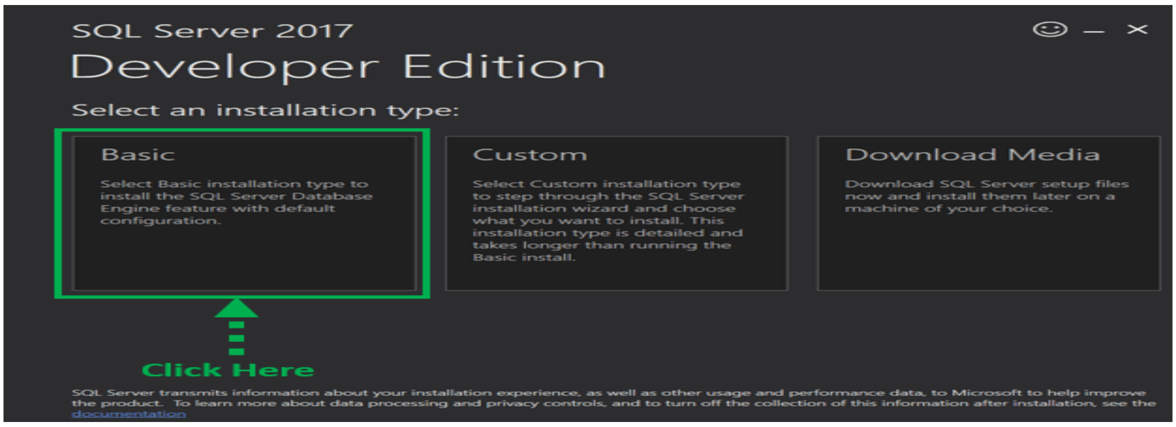
Download SQL Server setup files now and install them later on a machine of your choice.

SQL Server transmits information about your installation experience, as well as other usage and performance data, to Microsoft to help improve the product. To learn more about data processing and privacy controls, and to turn off the collection of this information after installation, see the documentation

14.1805.4072.1

2. **<u>Choose the version</u>**

   Choose the basic version by clicking on the ‘Basic’ option, as it has all default configuration required to learn MS SQL.



SQL Server 2017

Developer Edition

Select an installation type:

Basic

Select Basic installation type to install the SQL Server Database Engine feature with default configuration.

Custom

Select Custom installation type to step through the SQL Server installation wizard and choose what you want to install. This installation type is detailed and takes longer than running the Basic install.

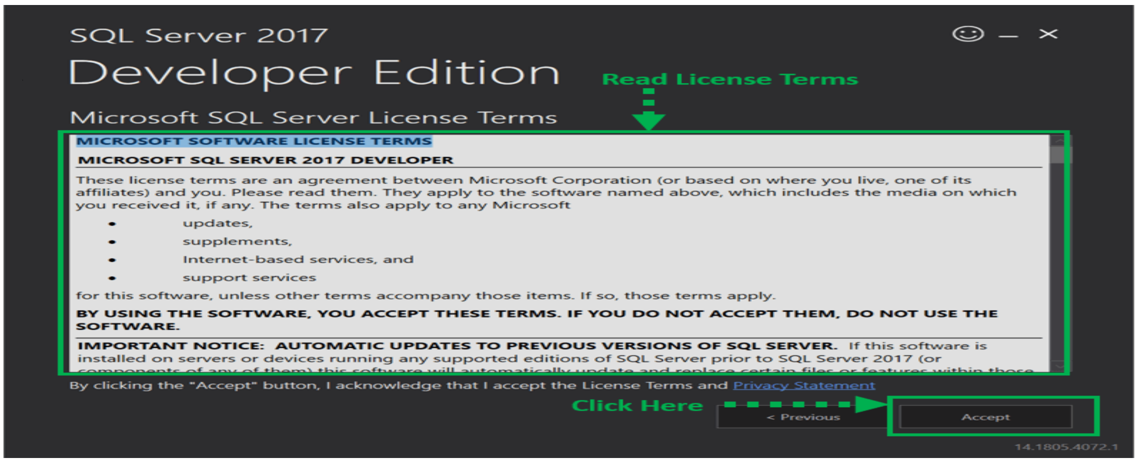
Download Media

Download SQL Server setup files now and install them later on a machine of your choice.

SQL Server transmits information about your installation experience, as well as other usage and performance data, to Microsoft to help improve the product. To learn more about data processing and privacy controls, and to turn off the collection of this information after installation, see the documentation

3. **Accept the terms**

   ‘Microsoft Server License Terms’ screen will appear. Read the License Terms and then click ‘Accept.’



MICROSOFT SOFTWARE LICENSE TERMS

MICROSOFT SQL SERVER 2017 DEVELOPER

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the software named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- updates,
- supplements,
- Internet-based services, and
- support services

for this software, unless other terms accompany those items. If so, those terms apply.

BY USING THE SOFTWARE, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT USE THE SOFTWARE.

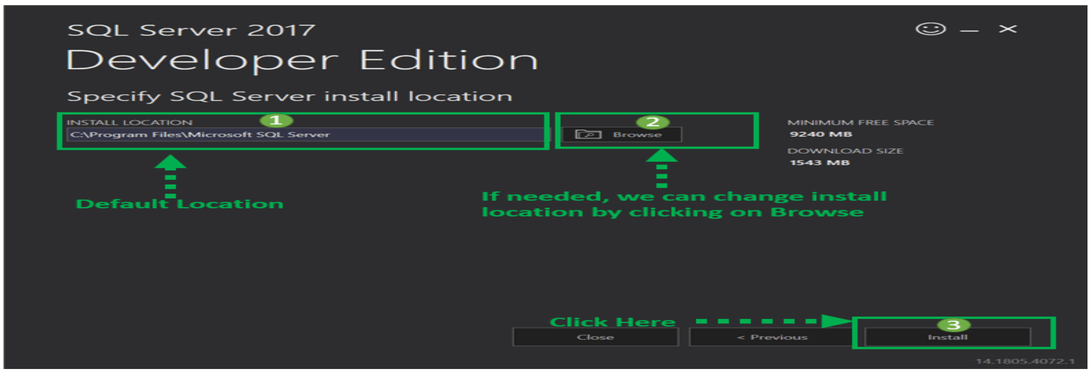
IMPORTANT NOTICE: AUTOMATIC UPDATES TO PREVIOUS VERSIONS OF SQL SERVER. If this software is installed on servers or devices running any supported editions of SQL Server prior to SQL Server 2017 (or

By clicking the "Accept" button, I acknowledge that I accept the License Terms and Privacy Statement

4. **Choose the location**

   Below ‘SQL server install location’ window will appear.

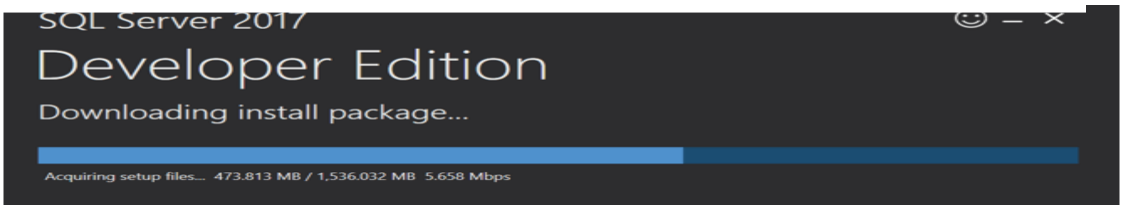
   - The Default location is C:\Program Files\Microsoft SQL Server.
   - Optionally, we canalso change the installation location by clicking on Browse.3. Once the location is selected, click the ‘Install’ button to start SQL installation Windows 10.



INSTALL LOCATION
C:\Program Files\Microsoft SQL Server

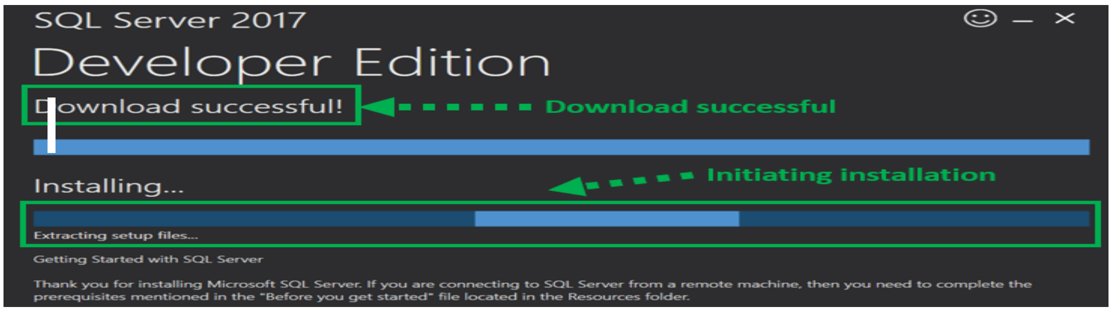
MINIMUM FREE SPACE
9240 MB
DOWNLOAD SIZE
1543 MB

Below ‘Downloading install package’ progress screen will be displayed. Wait until the SQL software download is complete.



Acquiring setup files... 473.813 MB / 1,536.032 MB 5.658 Mbps

Once, the download is complete; the system will initiate installing developer edition.

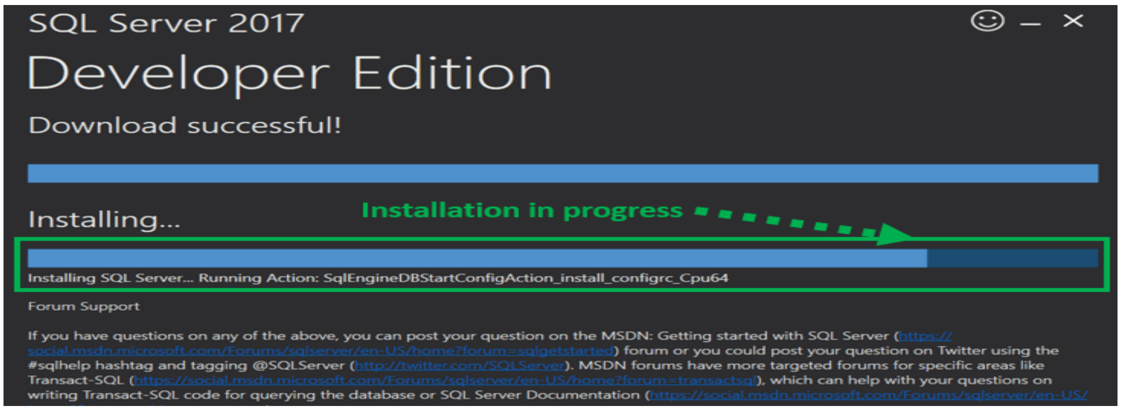


Extracting setup files...

Getting Started with SQL Server

Thank you for installing Microsoft SQL Server. If you are connecting to SQL Server from a remote machine, then you need to complete the prerequisites mentioned in the "Before you get started" file located in the Resources folder.

Below screen is show installation progress.



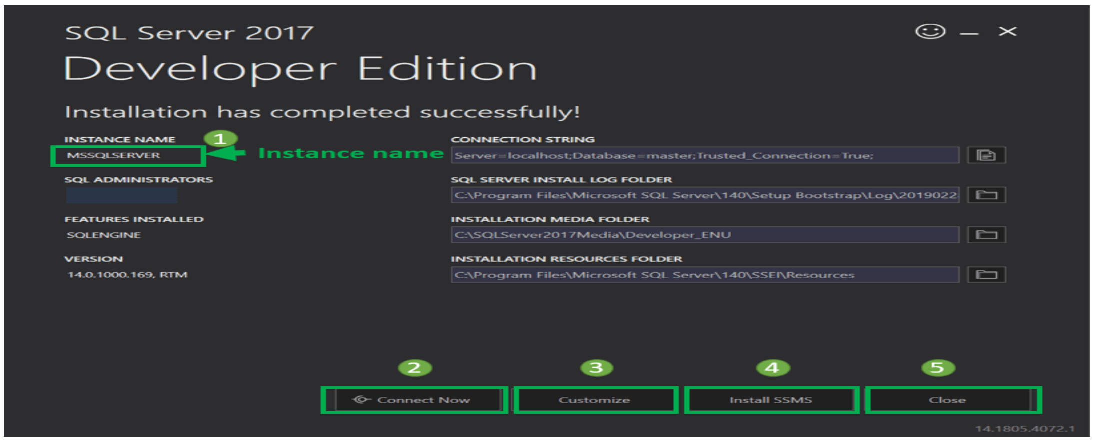
Installing SQL Server... Running Action: SqlEngineDBStartConfigAction_install_configrc_Cpu64

Forum Support

If you have questions on any of the above, you can post your question on the MSDN: Getting started with SQL Server (https://social.msdn.microsoft.com/Forums/sqlserver/en-US/home?forum=sqlgetstarted) forum or you could post your question on Twitter using the #sqlhelp hashtag and tagging @SQLServer (http://twitter.com/SQLServer). MSDN forums have more targeted forums for specific areas like Transact-SQL (https://social.msdn.microsoft.com/Forums/sqlserver/en-US/home?forum=transactsql), which can help with your questions on writing Transact-SQL code for querying the database or SQL Server Documentation (https://social.msdn.microsoft.com/Forums/sqlserver/en-US/

5. **Finish the installation process**

Once installation completed successfully, below screen will appear.



Installation has completed successfully!

INSTANCE NAME: MSSQLSERVER

SQL ADMINISTRATORS: [redacted]

FEATURES INSTALLED: SQLENGINE

VERSION: 14.0.1000.169, RTM

CONNECTION STRING: Server=localhost;Database=master;Trusted_Connection=True;

SQL SERVER INSTALL LOG FOLDER: C:\Program Files\Microsoft SQL Server\140\Setup Bootstrap\Log\2019022

INSTALLATION MEDIA FOLDER: C:\SQLServer2017Media\Developer_ENU

INSTALLATION RESOURCES FOLDER: C:\Program Files\Microsoft SQL Server\140\SSEI\Resources

## PRACTICAL 2: INTRODUCTION TO DDL STATEMENTS AND ITS IMPLEMENTATION IN ORACLE.

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.

Following are the various DDL commands:

1) **CREATE:**

The CREATE command is used to create new database objects such as tables, indexes, views, or even entire databases.

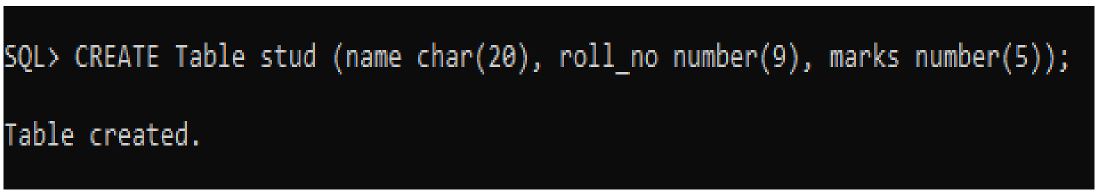
**Syntax:**

CREATE Table <tablename> (column1 datatype(specify), (column2 datatype(specify)……, (column N datatype(specify));

- CREATE TABLE: Specifies that you want to create a new table.
- table_name: Specifies the name of the table you want to create.
- (column1, column2, ..., columnN): Lists the columns you want to create in the table. Each column is followed by its name.
- datatype: Specifies the data type for each column.

**Example:**

CREATE Table student (name char(20),roll_no number(9), marks number(5));



```
SQL> CREATE Table stud (name char(20), roll_no number(9), marks number(5));

Table created.
```

2) **ALTER:**

The ALTER command is used to modify the structure of an existing database object, such as a table, column, or index.

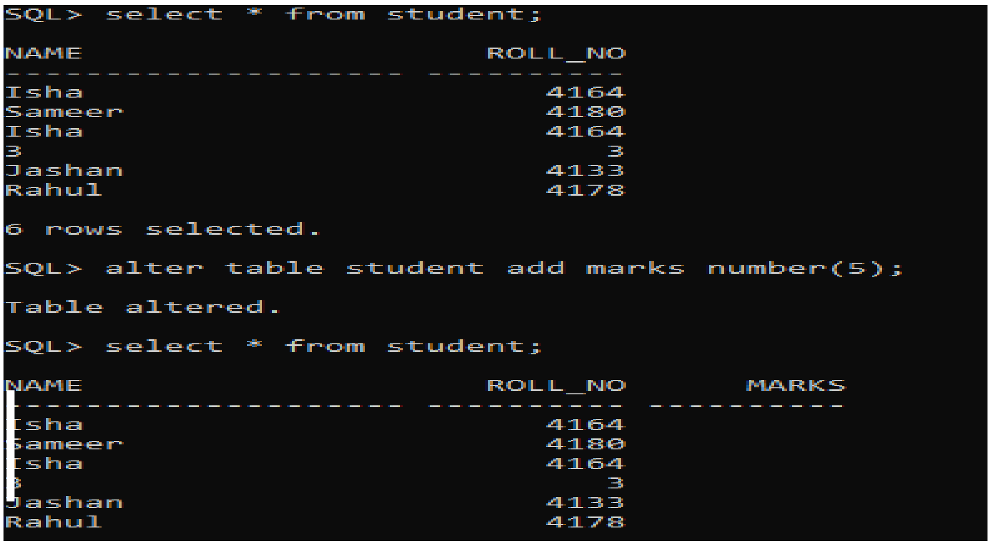
- When we want to add a new column in a table,

  **Syntax**:

  alter table table_name modify column_name new_data_type;

  **Example**:

  alter table student add marks number(5);



```
SQL> select * from student;

NAME                  ROLL_NO
-------------------- ----------
Isha                       4164
Sameer                     4180
Isha                       4164
3                             3
Jashan                     4133
Rahul                      4178

6 rows selected.

SQL> alter table student add marks number(5);

Table altered.

SQL> select * from student;

NAME                  ROLL_NO      MARKS
-------------------- ---------- ----------
Isha                       4164
Sameer                     4180
Isha                       4164
3                             3
Jashan                     4133
Rahul                      4178
```

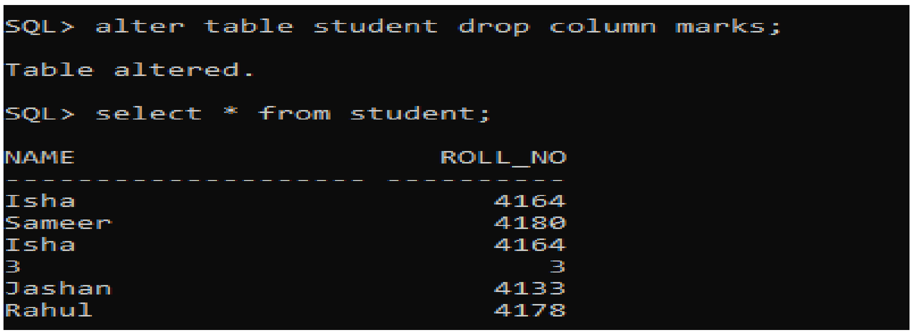
- When we want to delete a column in a table

  **Syntax:**

  alter table table_name drop column column_name;

  **Example:**

  alter table student drop column marks;



```
SQL> alter table student drop column marks;

Table altered.

SQL> select * from student;

NAME                  ROLL_NO
-------------------- ----------
Isha                       4164
Sameer                     4180
Isha                       4164
3                             3
Jashan                     4133
Rahul                      4178
```

**3) TRUNCATE:**

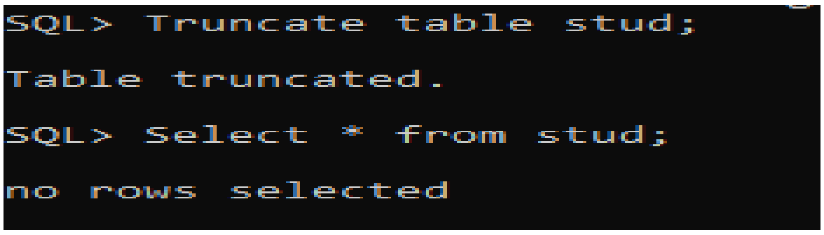
The TRUNCATE command is used to remove all the rows from an existing table while keeping the table's structure intact. This SQL statement is a quick way to delete all the data in a table without dropping and recreating the table itself.

**<u>Syntax:</u>**

Truncate table_name;

**<u>Example:</u>**

Truncate table stud;



```
SQL> Truncate table stud;

Table truncated.

SQL> Select * from stud;

no rows selected
```

**4) DROP:**

The DROP command is used to remove or delete existing database objects, such as tables, indexes, views, or even entire databases.

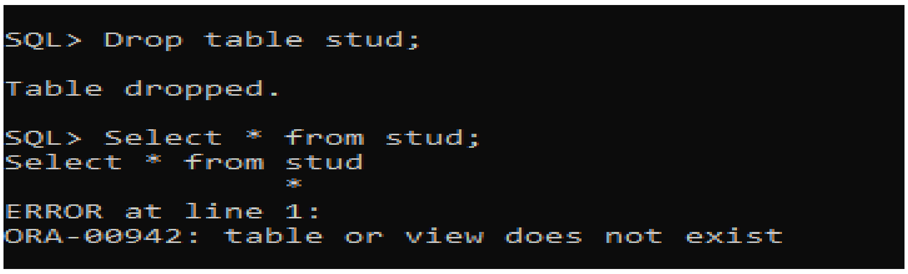
**<u>Syntax:</u>**

drop table tablename;

- DROP TABLE statement is used to remove an existing table from the database, along with all its data and associated objects.
- table_name: Specifies the name of the table you want to drop.

**<u>Example</u>:**

Drop table stud;



```
SQL> Drop table stud;

Table dropped.

SQL> Select * from stud;
Select * from stud
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

**5) RENAME:**

The RENAME command is used to change the name of an existing database object, or even a database itself. This SQL statement allows you to modify the name of an object without altering its structure or properties.

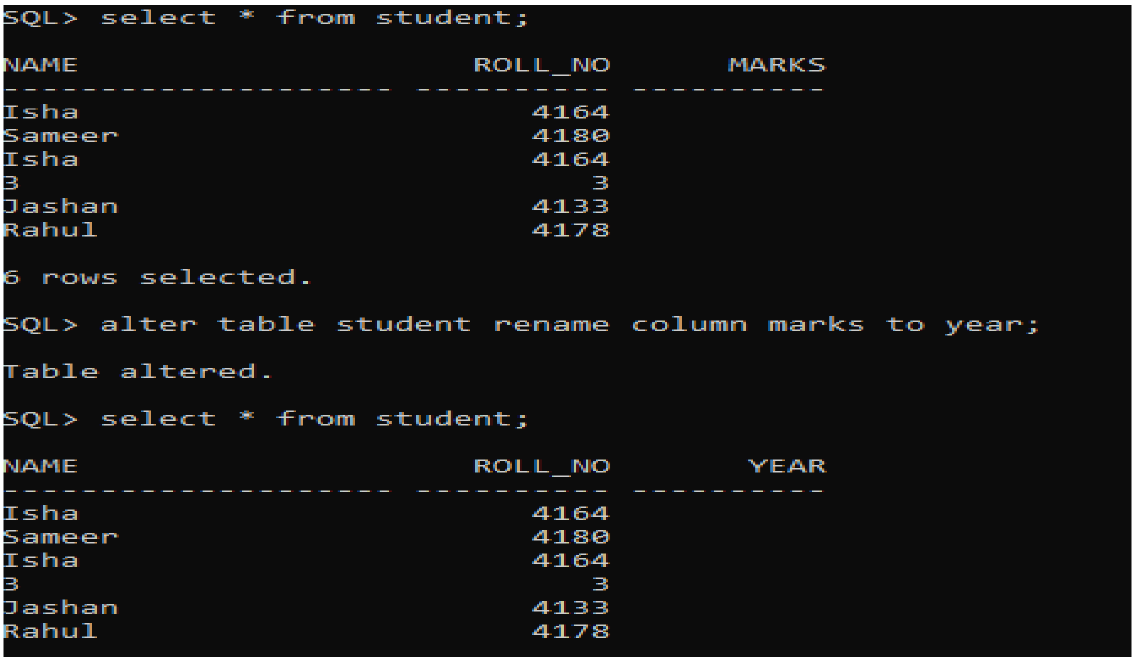
**Syntax:**

alter table table_name rename column old_ name to new_ name;

- old_name: Specifies the current name of the object you want to rename.
- new_name: Specifies the new name you want to assign to the object.

**Example:**

alter table student rename column marks to year;



```
SQL> select * from student;

NAME                   ROLL_NO      MARKS
-------------------- ---------- ----------
Isha                       4164
Sameer                     4180
Isha                       4164
3                             3
Jashan                     4133
Rahul                      4178

6 rows selected.

SQL> alter table student rename column marks to year;

Table altered.

SQL> select * from student;

NAME                   ROLL_NO       YEAR
-------------------- ---------- ----------
Isha                       4164
Sameer                     4180
Isha                       4164
3                             3
Jashan                     4133
Rahul                      4178
```

**PRACTICAL 3: INTRODUCTION TO DML AND TCL STATEMENTS AND ITS IMPLEMENTATION IN ORACLE.**

## Data Manipulation Language (DML):

The SQL commands that deal 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. DML performs the following operations:

1) **SELECT:**

The SELECT command is a fundamental SQL statement used for querying data from a database. It allows to retrieve specific information from one or more database tables based on certain conditions.

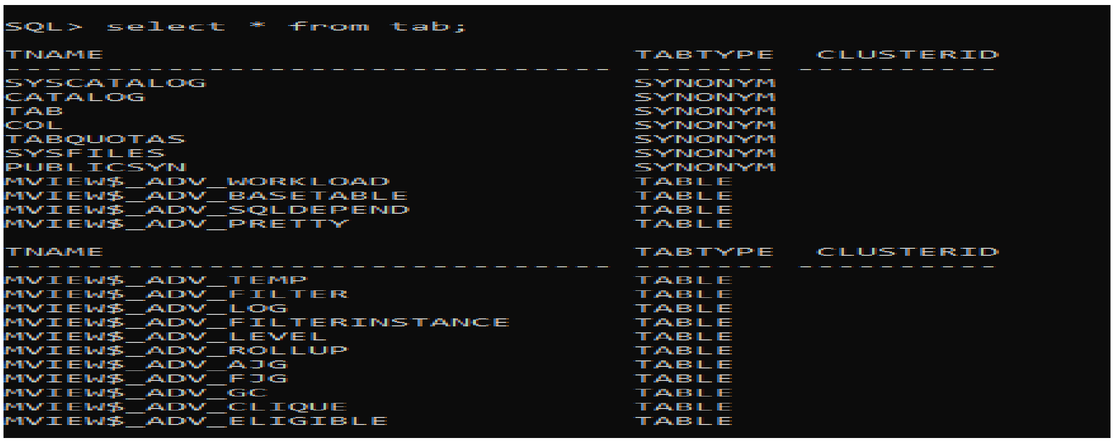
**Syntax:**

select * from tablename;

- SELECT: Specifies that you want to retrieve data from the database.
- FROM table_name: Specifies the table from which you want to retrieve data.

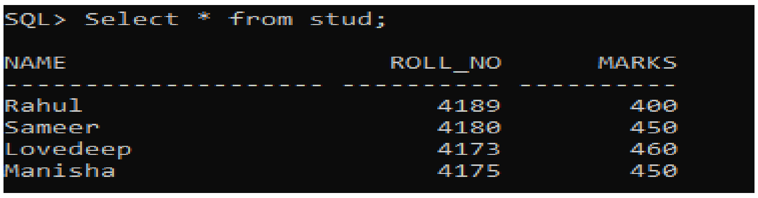
**Example:**

select * from tab;



```
SQL> select * from tab;
TNAME                          TABTYPE  CLUSTERID
------------------------------ ------- ----------
SYSCATALOG                     SYNONYM
CATALOG                        SYNONYM
TAB                            SYNONYM
COL                            SYNONYM
TABQUOTAS                      SYNONYM
SYSFILES                       SYNONYM
PUBLICSYN                      SYNONYM
MVIEW$_ADV_WORKLOAD            TABLE
MVIEW$_ADV_BASETABLE           TABLE
MVIEW$_ADV_SQLDEPEND           TABLE
MVIEW$_ADV_PRETTY              TABLE
TNAME                          TABTYPE  CLUSTERID
------------------------------ ------- ----------
MVIEW$_ADV_TEMP                TABLE
MVIEW$_ADV_FILTER              TABLE
MVIEW$_ADV_LOG                 TABLE
MVIEW$_ADV_FILTERINSTANCE      TABLE
MVIEW$_ADV_LEVEL               TABLE
MVIEW$_ADV_ROLLUP              TABLE
MVIEW$_ADV_AJG                 TABLE
MVIEW$_ADV_FJG                 TABLE
MVIEW$_ADV_GC                  TABLE
MVIEW$_ADV_CLIQUE              TABLE
MVIEW$_ADV_ELIGIBLE            TABLE
```

select * from stud;



```
SQL> Select * from stud;

NAME                    ROLL_NO      MARKS
-------------------- ---------- ----------
Rahul                      4189        400
Sameer                     4180        450
Lovedeep                   4173        460
Manisha                    4175        450
```

2) **INSERT:**

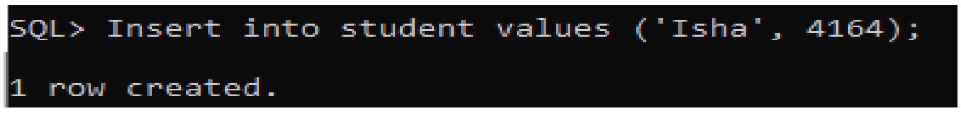
The INSERT command is used to add new rows or records to a table. This statement allows to provide values for each column in the table, which are then inserted into the specified table's rows. There are two ways to inset the data in the table.

**Syntax (way1) :**

Insert into <tablename> values ('value1, value2, value3,……);

**Example:**

Insert into student values ('Isha', 4164)



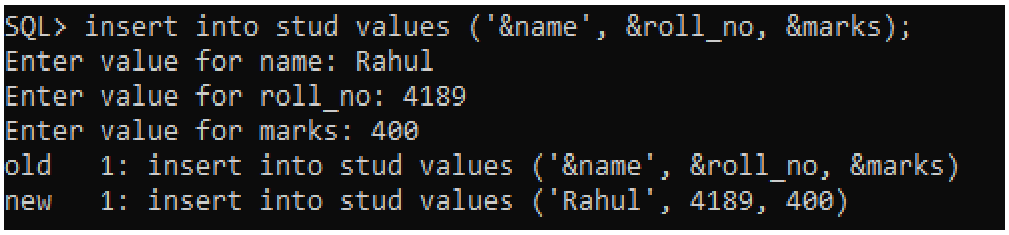
```
SQL> Insert into student values ('Isha', 4164);

1 row created.
```

**Syntax (way 2) :**

Insert into <tablename> values ('&column1, &column2, &colun3,…);

**Example:**

insert into stud values (‘&name’, &roll_no, &marks);



```
SQL> insert into stud values ('&name', &roll_no, &marks);
Enter value for name: Rahul
Enter value for roll_no: 4189
Enter value for marks: 400
old   1: insert into stud values ('&name', &roll_no, &marks)
new   1: insert into stud values ('Rahul', 4189, 400)
```

- INSERT INTO: Specifies that you want to insert data into a table.
- table_name: Specifies the name of the table into which you want to insert data.
- (column1, column2, column3, ...): Specifies the columns into which you want to insert values. This part is optional if you are inserting values into all columns in the order they are defined in the table.
- VALUES (value1, value2, value3, ...): Specifies the values you want to insert into the columns. The values must be provided in the same order as the columns listed in the previous part.

3) **DELETE:**

The DELETE command is used to remove rows or records from a table based on specified conditions. This SQL statement allows you to selectively remove data from a table while maintaining the integrity and structure of the database.

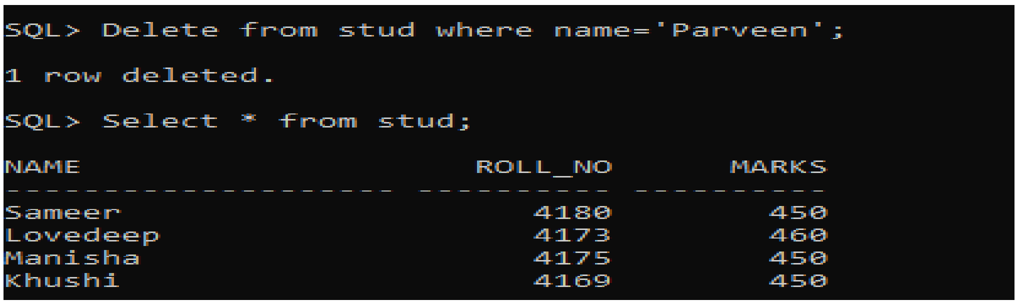
**Syntax:**

Delete from table_name where condition;

- DELETE FROM: Specifies that you want to delete data from a table.
- WHERE condition: Specifies the condition that determines which rows should be deleted. If no condition is provided, all rows in the table will be deleted.

**Example:**

Delete from stud where name='Parveen';



```
SQL> Delete from stud where name='Parveen';

1 row deleted.

SQL> Select * from stud;

NAME                    ROLL_NO      MARKS
-------------------- ---------- ----------
Sameer                     4180        450
Lovedeep                   4173        460
Manisha                    4175        450
Khushi                     4169        450
```

4) **UPDATE:**

This command is used to modify existing records within a table. This statement allows to change the values of specific columns in one or more rows based on certain conditions.

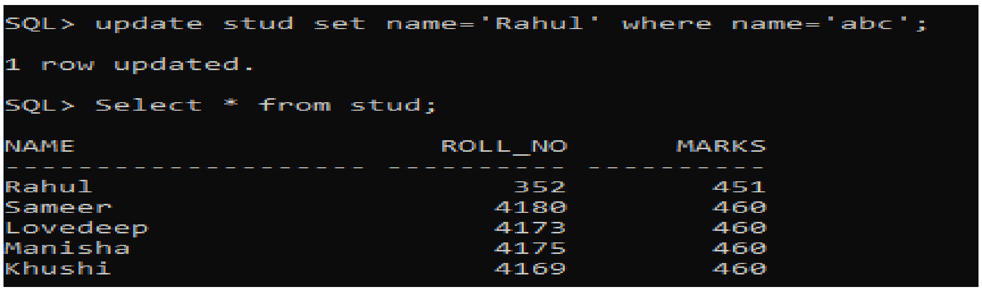
**Syntax:**

update table_name set column1=new_value1, column2=new_value2, …. where condition;

- UPDATE: Specifies that you want to modify data in a table.
- SET column1=new_value1, column2=new_value2, ...: Lists the columns you want to update along with the new values you want to assign to them.
- WHERE condition: Specifies the condition that determines which rows should be updated. If no condition is provided, all rows in the table will be updated.

**Example:**

update stud set name='Rahul' where name='abc';



```
SQL> update stud set name='Rahul' where name='abc';

1 row updated.

SQL> Select * from stud;

NAME                    ROLL_NO      MARKS
-------------------- ---------- ----------
Rahul                       352        451
Sameer                     4180        460
Lovedeep                   4173        460
Manisha                    4175        460
Khushi                     4169        460
```

## Transaction Control Language (TCL):

TCL statement is used to manage the changes made by DML statements. It allows statements to be grouped together into logical level for revoking the transaction and also commit to the database. TCL includes following commands:

1) **COMMIT:**

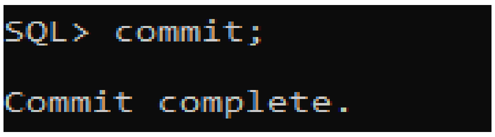
   It is used to permanently save all the modifications are done (all the transactions) by the DML commands in the database. Once issued, it cannot be undone.

   **Syntax:**

   commit;

   **Example:**

   commit;



```
SQL> commit;

Commit complete.
```

2) **SAVEPOINT:**

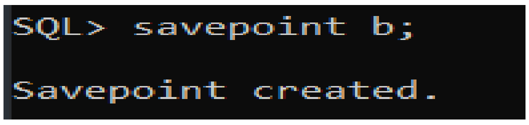
   It is used to create a point within the groups of transactions to save or roll back later. It is used to roll the transactions back to a certain point without the need to roll back the whole group of transactions

   **Syntax:**

   savepoint savepoint_name;

   **Example:**

   savepoint b;



```
SQL> savepoint b;

Savepoint created.
```

3) **ROLLBACK:**

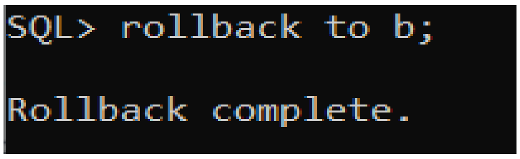
   It is used to undo the transactions that have not already been permanently saved (or committed) to the database. It restores the previously stored value, i.e., the data present before the execution of the transactions.

**Syntax:**

rollback to savepoint_name;

**Example:**

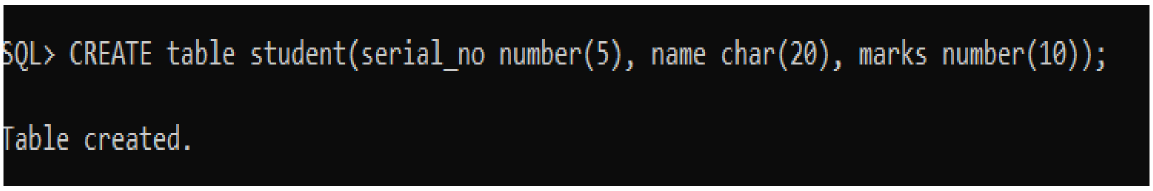
rollback to b;



```
SQL> rollback to b;

Rollback complete.
```

## PRACTICAL 4: IMPLEMENTATION OF AGGREGATE FUNCTIONS.

Aggregate functions in a Database Management System (DBMS) are functions that operate on sets of values within a database table and return a single value as a result. These functions perform calculations on groups of rows to generate summary information about the data. Aggregate functions are commonly used in SQL queries to perform calculations like sum, average, count, maximum, and minimum on columns of a table. Here are some common aggregate functions and their explanations:



```
SQL> CREATE table student(serial_no number(5), name char(20), marks number(10));

Table created.
```

1) **SUM( ) :**

   The SUM aggregate function in a Database Management System is used to calculate the total sum of numeric values within a specified column of a table. It is often used to obtain the sum of all values in a column, which can be particularly useful for financial, statistical, or analytical calculations.

   **Syntax:**

   Select SUM(column_name) from tablename;

   **OR**

   Select SUM(column_name) as sum_result_name from tablename;

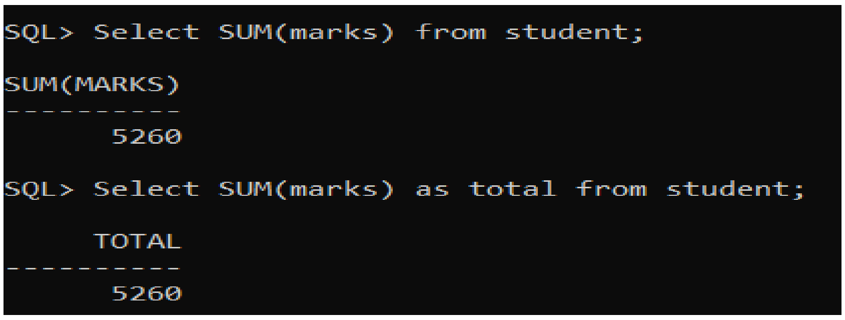
   - column_name: Specifies the name of the column from which you want to calculate the sum.

   **Example:**

   Select SUM (marks) from student;

   **OR**

   Select SUM(marks) as sum_total from student;



```
SQL> Select SUM(marks) from student;

SUM(MARKS)
----------
      5260

SQL> Select SUM(marks) as total from student;

     TOTAL
----------
      5260
```

**2) AVG :**

The AVG aggregate function in a Database Management System is used to calculate the average (mean) value of numeric values within a specified column of a table. It is used to obtain the average of all values in a column, which can be useful for various statistical and analytical calculations.

**Syntax:**

Select AVG(column_name) from tablename;

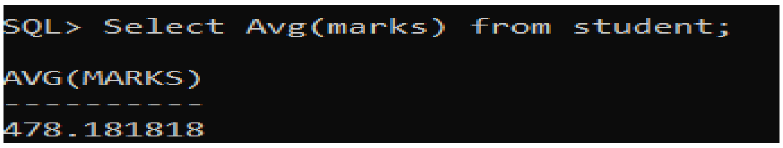
**OR**

Select AVG(column_name) as avg_result_name from tablename;

- column_name: Specifies the name of the column from which you want to calculate the sum.

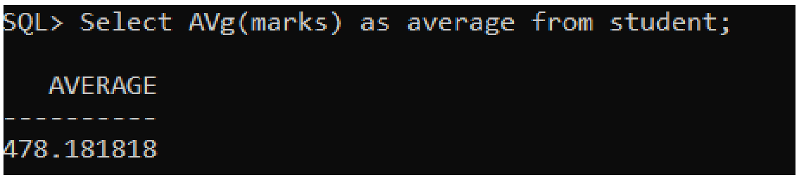
**Example:**

Select Avg (marks) from student;



```
SQL> Select Avg(marks) from student;

AVG(MARKS)
----------
478.181818
```

Select Avg (marks) as average from student;



```
SQL> Select AVg(marks) as average from student;

   AVERAGE
----------
478.181818
```

**3) MAX:**

The MAX aggregate function in a Database Management System (DBMS) is used to retrieve the maximum (largest) value from a specified column of a table. It helps you find the highest value among a set of numeric or alphanumeric values in a column.

**Syntax:**

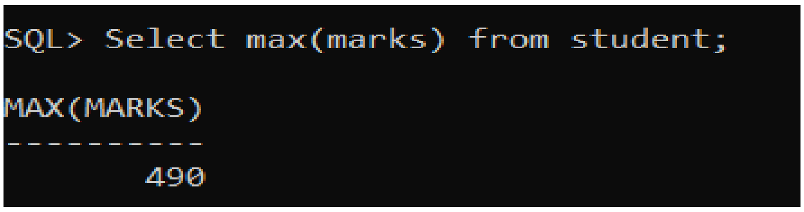
Select max(column_name) from tablename;

**OR**

Select max(column_name) as max_result_name from tablename;

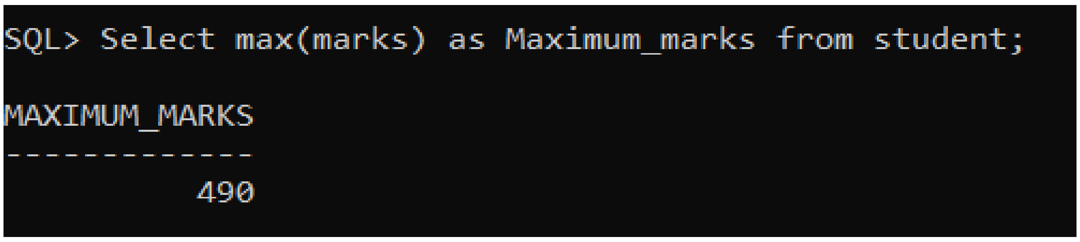
**Example:**

Select max (marks) from student;



```
SQL> Select max(marks) from student;

MAX(MARKS)
----------
       490
```

Select max (marks) as Maximum_marks from student;



```
SQL> Select max(marks) as Maximum_marks from student;

MAXIMUM_MARKS
-------------
          490
```

**4) MIN:**

The MIN aggregate function in a Database Management System (DBMS) is used to retrieve the minimum (smallest) value from a specified column of a table. It helps you find the least value among a set of numeric or alphanumeric values in a column.

**Syntax:**

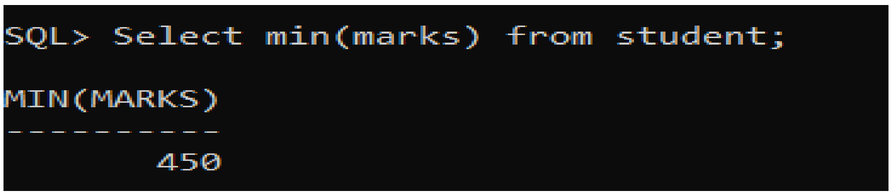
Select min(column_name) from tablename;

**OR**

Select min(column_name) as min_result_name from tablename;

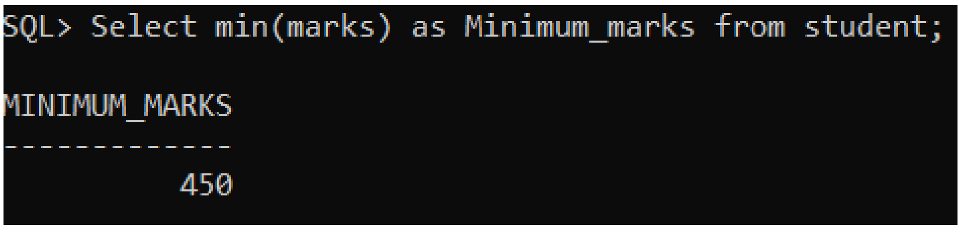
**Example:**

Select min (marks) from student;



```
SQL> Select min(marks) from student;

MIN(MARKS)
----------
       450
```

Select min (marks) as Minimun_marks from student;



```
SQL> Select min(marks) as Minimum_marks from student;

MINIMUM_MARKS
-------------
          450
```

**5) COUNT :**

The COUNT aggregate function in a Database Management System is used to calculate the number of rows in a result set or the number of non-null values in a specified column of a table. It helps you count the occurrences of records or values in a dataset.

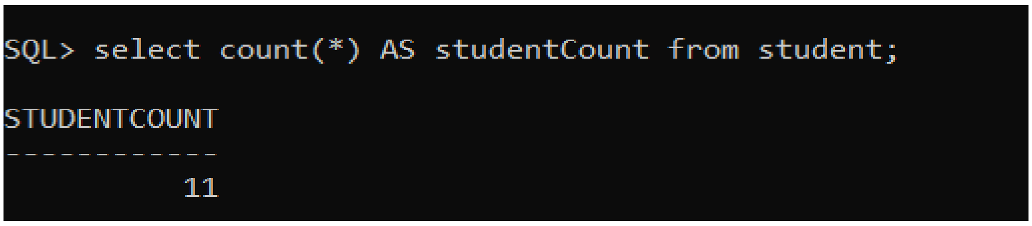
**<u>Syntax:</u>**

select count(*) AS count_result_name from table_name;

- *: Specifies that you want to count all rows in the result set.

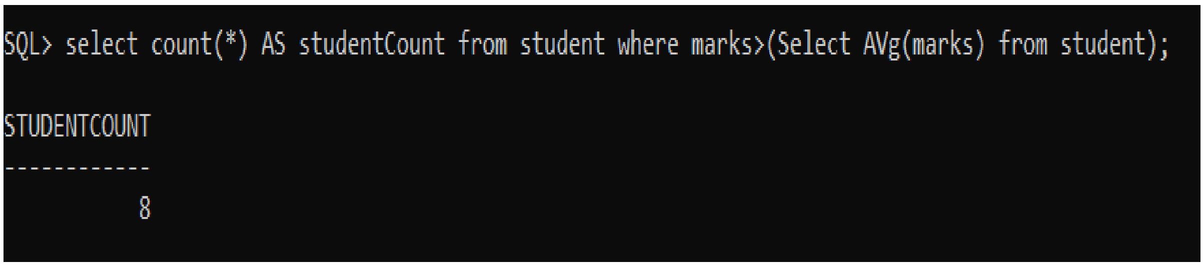
**<u>Example:</u>**

select count(*) AS studentCount from student;



```
SQL> select count(*) AS studentCount from student;

STUDENTCOUNT
------------
          11
```

select count(*) AS studentCount from student where marks>(Select AVG(marks) from student);



```
SQL> select count(*) AS studentCount from student where marks>(Select AVg(marks) from student);

STUDENTCOUNT
-----------
          8
```

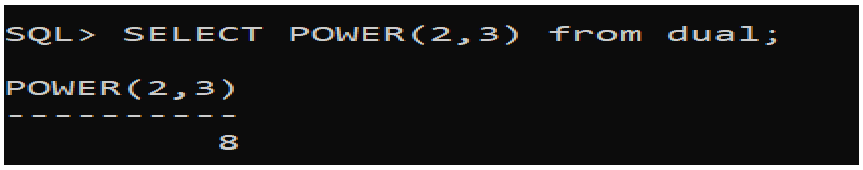
**6) POWER:**

The POWER() function returns the value of a number raised to the power of another number.

**<u>Syntax:</u>**

Select POWER(base,exponent) from dual;

**<u>Example:</u>**



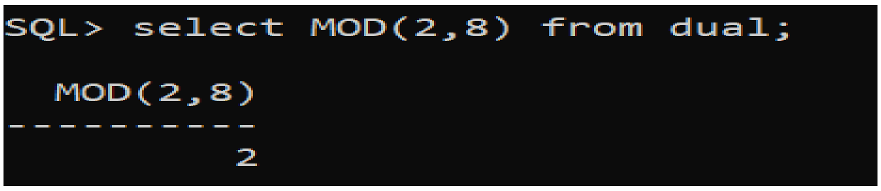
```
SQL> SELECT POWER(2,3) from dual;

POWER(2,3)
----------
         8
```

7) **MODE:**

**Syntax:**

Selcet MODE( ) from dual;

**Example:**



```
SQL> select MOD(2,8) from dual;

  MOD(2,8)
----------
         2
```

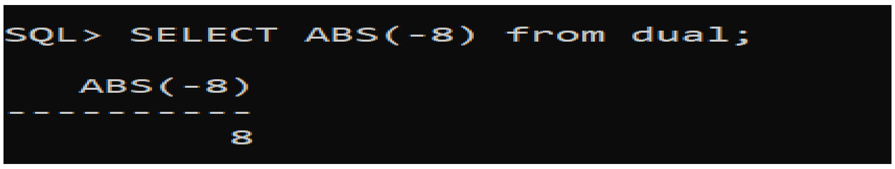
8) **ABSOLUTE:**

This function takes a single argument, which is the value you want to compute the absolute value of.

**Syntax:**

Selcet ABS(number) from dual;

**Example:**



```
SQL> SELECT ABS(-8) from dual;

   ABS(-8)
----------
         8
```

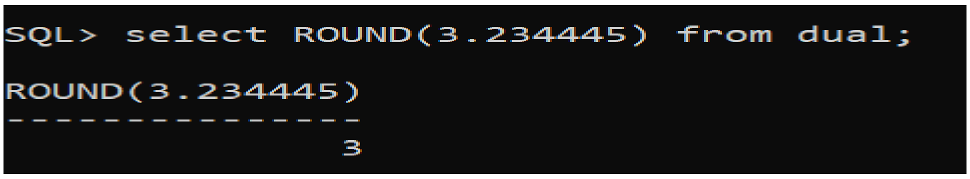
9) **ROUND OFF:**

Round off a number to a specified number of decimal places using the ROUND function.

**Syntax:**

Select ROUND(number) from dual;

**Example:**



```
SQL> select ROUND(3.234445) from dual;

ROUND(3.234445)
---------------
              3
```

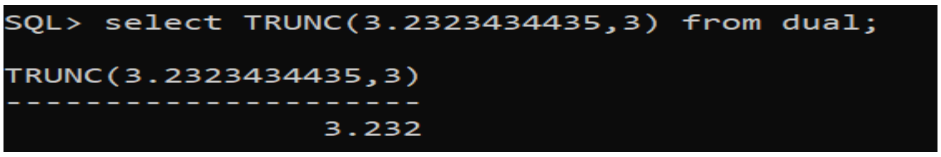
**10) TRUNCATE:**

The TRUNC function is used to truncate a number to a specified level of precision.

**Syntax:**

Select TRUNC(number, decimals) from dual;

**Example:**



```
SQL> select TRUNC(3.2323434435,3) from dual;

TRUNC(3.2323434435,3)
---------------------
                3.232
```

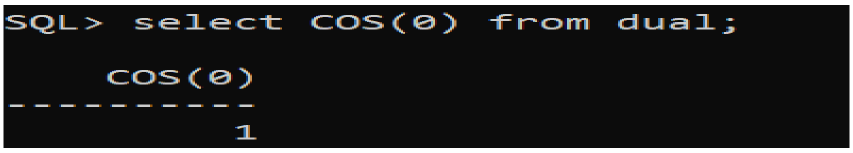
**11) COSINE:**

The SIN function to calculate the cosine of an angle in radians.

**Syntax:**

Select COS(angle) from dual;

**Example:**



```
SQL> select COS(0) from dual;

     COS(0)
----------
         1
```

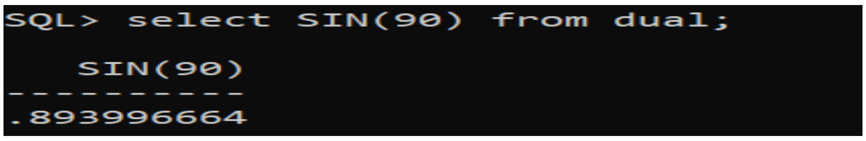
**12) SINE:**

The SIN function to calculate the sine of an angle in radians.

**Syntax:**

Select SIN(angle) from dual;

**Example:**



```
SQL> select SIN(90) from dual;

   SIN(90)
----------
.893996664
```

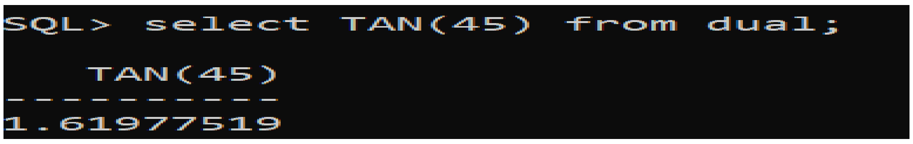
**13) TANGENT:**

The SIN function to calculate the tangent of an angle in radians.

**Syntax:**

Select TAN(angle) from dual;

**Example:**



```
SQL> select TAN(45) from dual;

   TAN(45)
----------
1.61977519
```

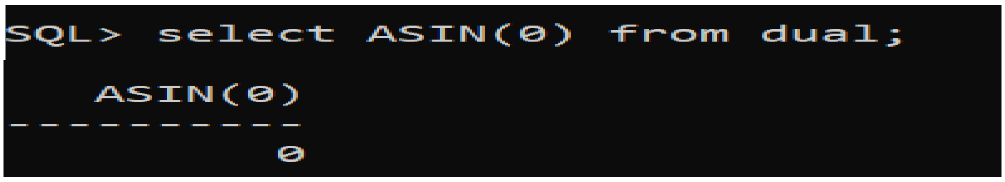
**14) ARCSINE:**

The ASIN() function returns the arc sine of a number. The specified number must be between -1 to 1, otherwise this function returns NULL.

**Syntax:**

select ASIN(angle) from dual;

**Example:**



```
SQL> select ASIN(0) from dual;

   ASIN(0)
----------
         0
```

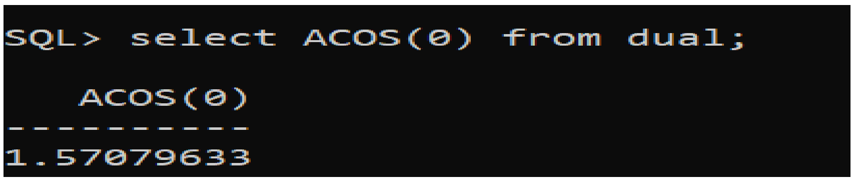
**15) ARCOSINE:**

The ACOS() function returns the arc cosine of a number. The specified number must be between -1 to 1, otherwise this function returns NULL.

**Syntax:**

select ACOS(angle) from dual;

**Example:**



```
SQL> select ACOS(0) from dual;

   ACOS(0)
----------
1.57079633
```

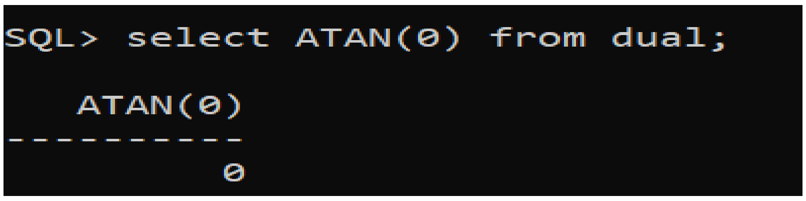
**16) ARCTANGENT:**

The ATAN() function in **SQL Server** is used to return the arc tangent or the inverse tangent of a specified value. This function accepts only one parameter which is a number and the range accepted for the argument number is unbounded.

**Syntax:**

select ATAN(angle) from dual;

**Example:**



```
SQL> select ATAN(0) from dual;

   ATAN(0)
----------
         0
```

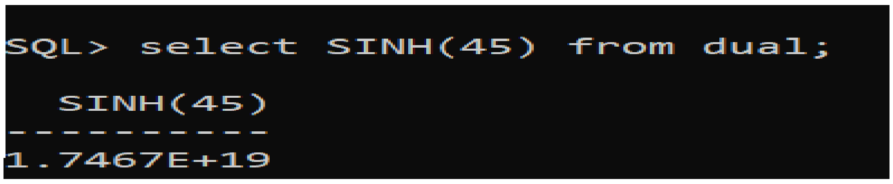
**17) SINH:**

SINH returns the hyperbolic sine of n . This function takes as an argument any numeric datatype or any nonnumeric datatype that can be implicitly converted to a numeric datatype.

**Syntax:**

select SINH(angle) from dual;

**Example:**



```
SQL> select SINH(45) from dual;

  SINH(45)
----------
1.7467E+19
```

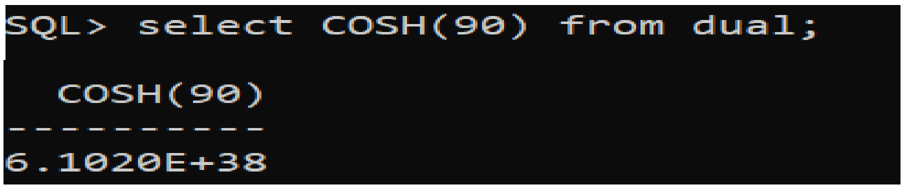
**18) COSH:**

COSH returns the hyperbolic cosine of n . This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type.

**Syntax:**

select COSH(angle) from dual;

**Example:**



```
SQL> select COSH(90) from dual;

  COSH(90)
----------
6.1020E+38
```

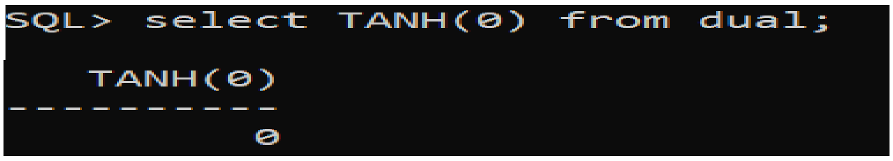
**19) TANH:**

The **TANH() method** in SQL returns the hyperbolic tangent of a parametric value. More precisely, it takes a double-precision number in arguments that specify the angle in radians. TANH() returns the hyperbolic tangent of this specific angle.

**Syntax:**

select TANH(angle) from dual;

**Example:**



```
SQL> select TANH(0) from dual;

   TANH(0)
----------
         0
```

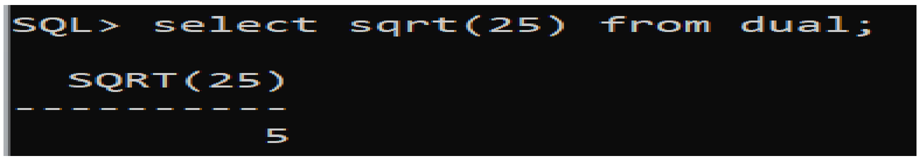
## 20) SQUARE ROOT:

This function in **SQL Server** is used to return the square root of a specified positive number.

**Syntax:**

select sqrt(number) from dual;

**Example:**



```
SQL> select sqrt(25) from dual;

  SQRT(25)
----------
         5
```

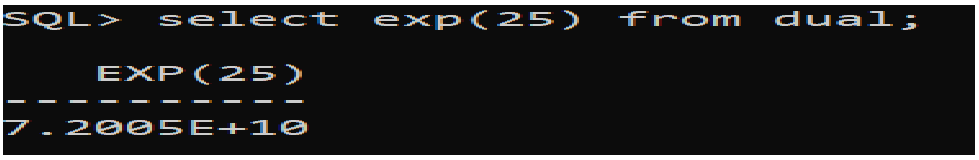
## 21) EXPONENTIAL:

This function in **SQL Server** is used to return a value which is **e** raised to the nth power, where n is a specified number.

**Syntax:**

select exp(number) from dual;

**Example:**



```
SQL> select exp(25) from dual;

   EXP(25)
----------
7.2005E+10
```

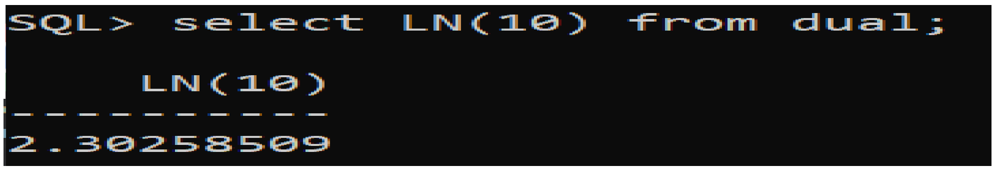
## 22) LOG:

The LOG() function returns the natural logarithm of a specified *number*, or the logarithm of the number to the specified base.

**Syntax:**

select LN(number) from dual;

**Example:**



```
SQL> select LN(10) from dual;

    LN(10)
----------
2.30258509
```

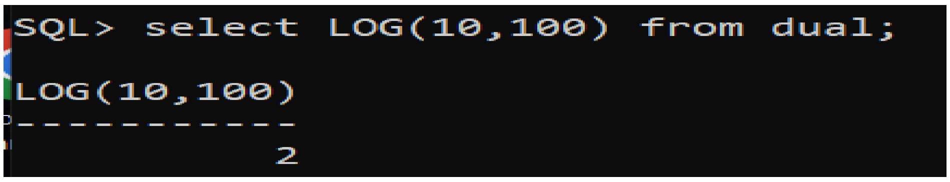
**23) NATURAL LOG:**

SQL LN() function returns the natural logarithm of n, where n is greater than 0 and its base is a number equal to approximately 2.71828183.

**Syntax:**

select LOG(Base, number) from dual;

**Example:**



```
SQL> select LOG(10,100) from dual;

LOG(10,100)
-----------
          2
```

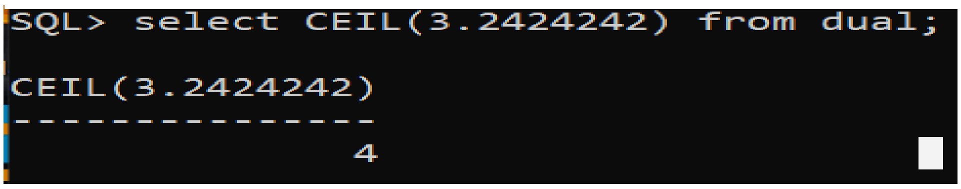
**24) CEILING:**

The CEILING() function returns the smallest integer value that is larger than or equal to a number.

**Syntax:**

select CEIL( number) from dual;

**Example:**



```
SQL> select CEIL(3.2424242) from dual;

CEIL(3.2424242)
---------------
              4
```

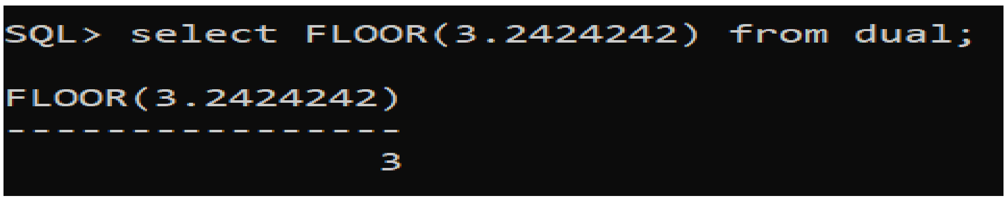
**25) FLOOR:**

The FLOOR() function returns the largest integer value that is smaller than or equal to a number.

**Syntax:**

select FLOOR( number) from dual;

**Example:**



```
SQL> select FLOOR(3.2424242) from dual;

FLOOR(3.2424242)
----------------
               3
```

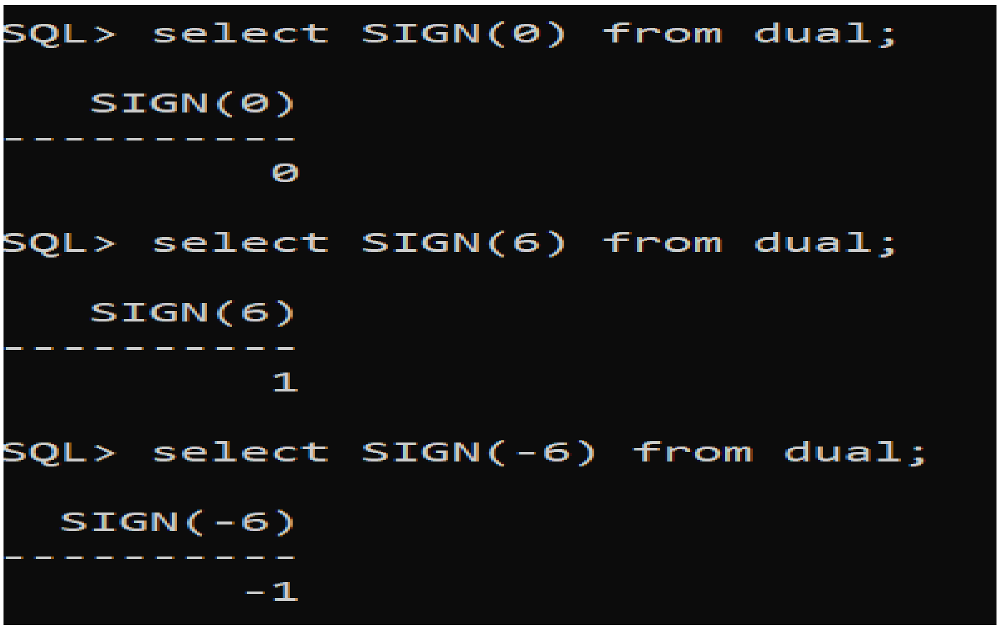
**26) SIGN:**

The SIGN() function returns the sign of a number. If number is 0 then sign(0) gives value 0. If number is positive then sign(positive_number) then gives value 1. If number is negative then sign(negative_number) then gives value -1.

**Syntax:**

select SIGN( number) from dual;

**Example:**



```
SQL> select SIGN(0) from dual;

   SIGN(0)
----------
         0

SQL> select SIGN(6) from dual;

   SIGN(6)
----------
         1

SQL> select SIGN(-6) from dual;

  SIGN(-6)
----------
        -1
```

# PRACTICAL 5: IMPLEMENTATION OF CONSTRAINTS IN SQL.

Constraints are guidelines or limitations imposed on database tables to maintain the integrity, correctness, and consistency of the data. Constraints can be used to enforce data linkages across tables, verify that data is unique, and stop the insertion of erroneous data. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraints and the data action, the action is aborted.

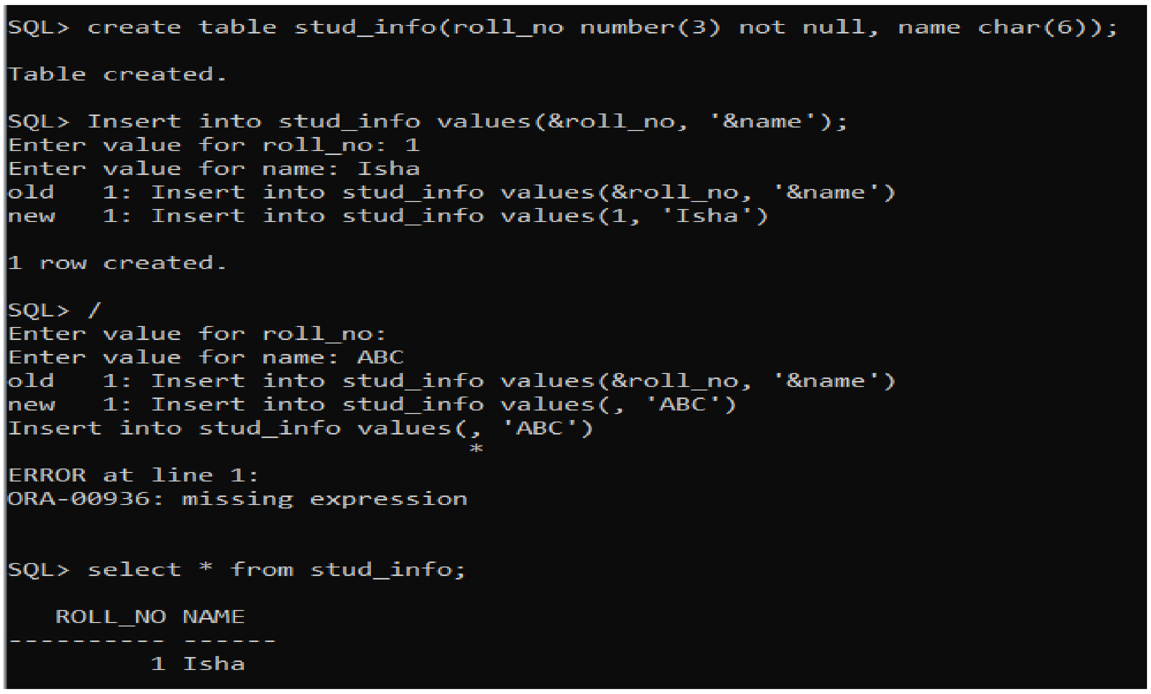
1) **NOT NULL:**

   This constraint tells that we cannot store a null value in a column. That is, if a column is specified as NOT NULL then we will not be able to store null in this particular column any more.

   **Syntax:**

   CREATE TABLE TableName (ColumnName1 datatype NOT NULL, ColumnName2 datatype,…., ColumnNameN datatype);

   **Example:**



```
SQL> create table stud_info(roll_no number(3) not null, name char(6));

Table created.

SQL> Insert into stud_info values(&roll_no, '&name');
Enter value for roll_no: 1
Enter value for name: Isha
old   1: Insert into stud_info values(&roll_no, '&name')
new   1: Insert into stud_info values(1, 'Isha')

1 row created.

SQL> /
Enter value for roll_no:
Enter value for name: ABC
old   1: Insert into stud_info values(&roll_no, '&name')
new   1: Insert into stud_info values(, 'ABC')
Insert into stud_info values(, 'ABC')
                             *
ERROR at line 1:
ORA-00936: missing expression


SQL> select * from stud_info;

   ROLL_NO NAME
---------- ------
         1 Isha
```

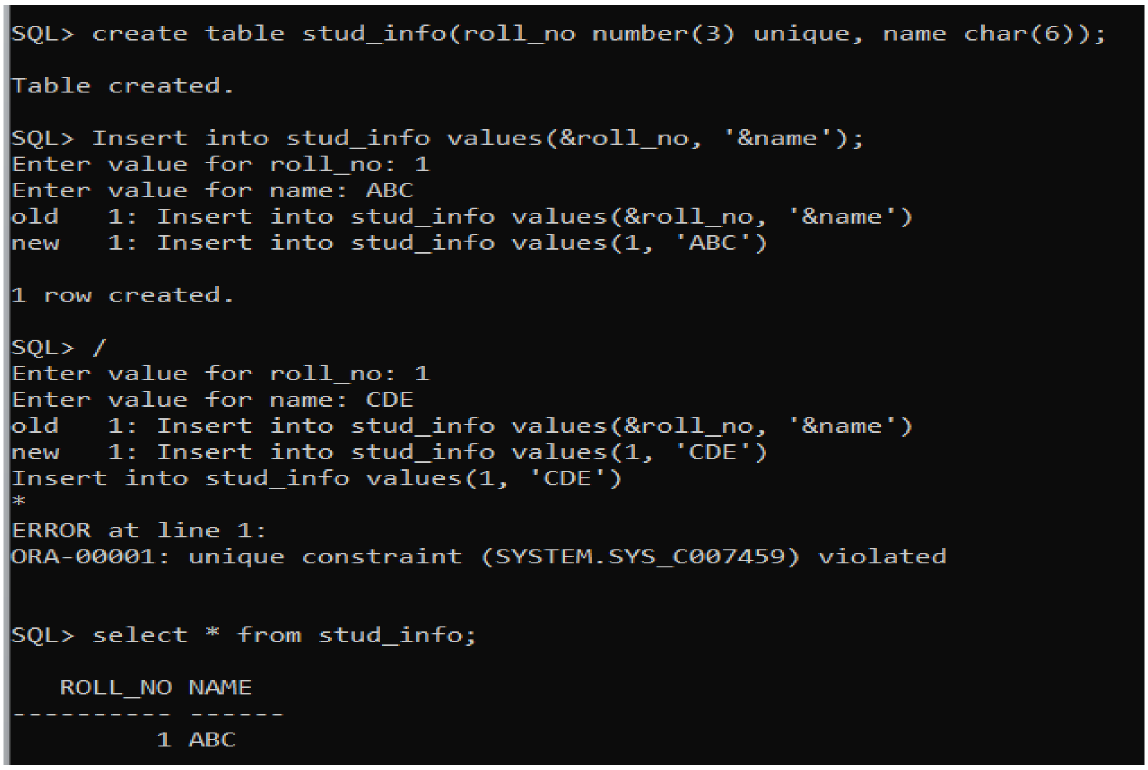
2) **UNIQUE:**

   This constraint helps to uniquely identify each row in the table. i.e. for a particular column, all the rows should have unique values. We can have more than one UNIQUE columns in a table.

**Syntax:**

CREATE TABLE TableName (ColumnName1 datatype UNIQUE, ColumnName2 datatype,…., ColumnNameN datatype);

**Example:**



```
SQL> create table stud_info(roll_no number(3) unique, name char(6));

Table created.

SQL> Insert into stud_info values(&roll_no, '&name');
Enter value for roll_no: 1
Enter value for name: ABC
old   1: Insert into stud_info values(&roll_no, '&name')
new   1: Insert into stud_info values(1, 'ABC')

1 row created.

SQL> /
Enter value for roll_no: 1
Enter value for name: CDE
old   1: Insert into stud_info values(&roll_no, '&name')
new   1: Insert into stud_info values(1, 'CDE')
Insert into stud_info values(1, 'CDE')
*
ERROR at line 1:
ORA-00001: unique constraint (SYSTEM.SYS_C007459) violated


SQL> select * from stud_info;

   ROLL_NO NAME
---------- ------
         1 ABC
```

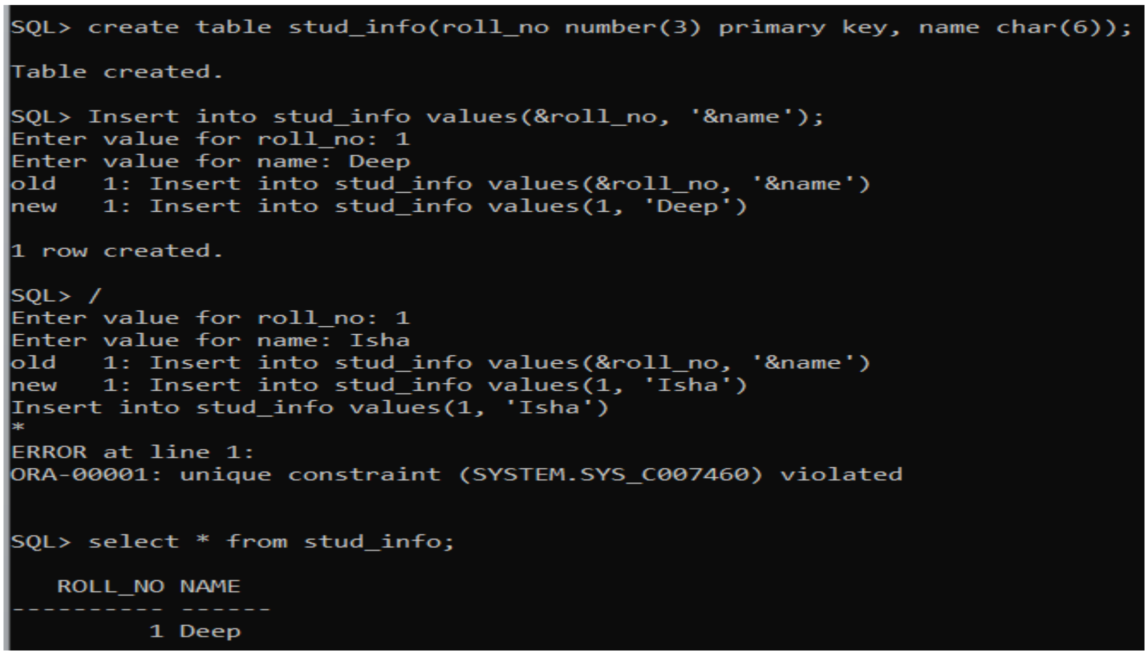
3) **PRIMARY KEY:**

Primary Key is a field which uniquely identifies each row in the table. If a field in a table as primary key, then the field will not be able to contain NULL values as well as all the rows should have unique values for this field. So, in other words we can say that this is combination of NOT NULL and UNIQUE constraints. A table can have only one field as primary key.

**SYNTAX:**

CREATE TABLE TableName (ColumnName1 datatype PRIMARY KEY, ColumnName2 datatype,…., ColumnNameN datatype);

**EXAMPLE:**



```
SQL> create table stud_info(roll_no number(3) primary key, name char(6));

Table created.

SQL> Insert into stud_info values(&roll_no, '&name');
Enter value for roll_no: 1
Enter value for name: Deep
old   1: Insert into stud_info values(&roll_no, '&name')
new   1: Insert into stud_info values(1, 'Deep')

1 row created.

SQL> /
Enter value for roll_no: 1
Enter value for name: Isha
old   1: Insert into stud_info values(&roll_no, '&name')
new   1: Insert into stud_info values(1, 'Isha')
Insert into stud_info values(1, 'Isha')
*
ERROR at line 1:
ORA-00001: unique constraint (SYSTEM.SYS_C007460) violated


SQL> select * from stud_info;

   ROLL_NO NAME
---------- ------
         1 Deep
```

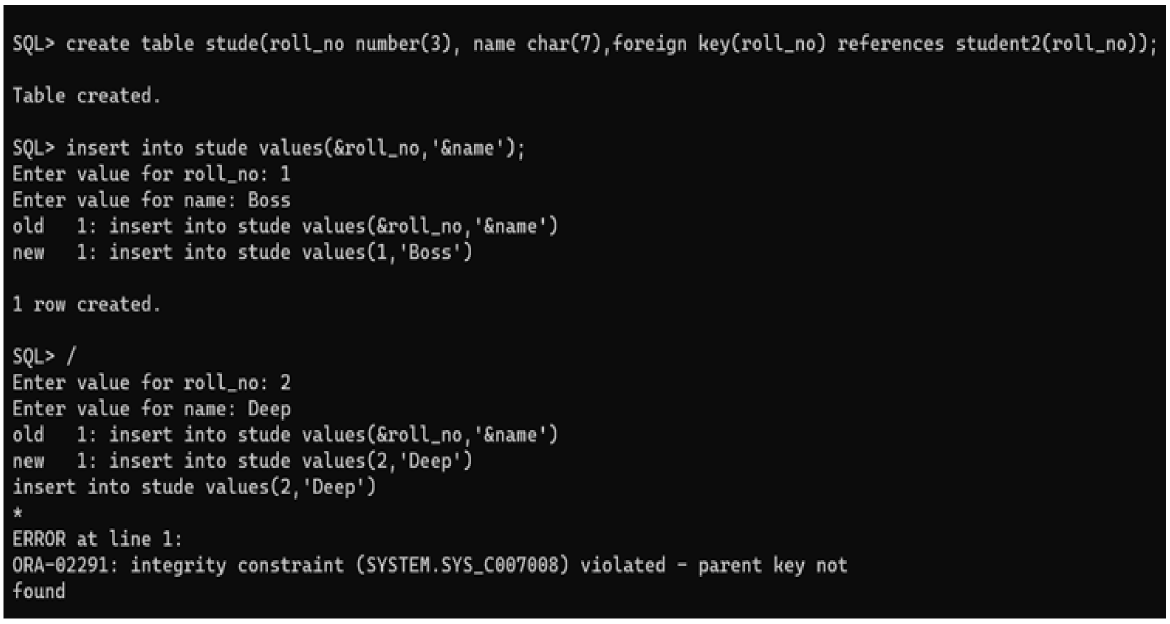
4) **FOREIGN KEY:**

Foreign Key is a field in a table which uniquely identifies each row of a another table. That is, this field points to primary key of another table. This usually creates a kind of link between the tables.

**Syntax:**

CREATE TABLE tablename(ColumnName1 Datatype(SIZE) PRIMARY KEY, ColumnNameN Datatype(SIZE), FOREIGN KEY( ColumnName ) REFERENCES PARENT_TABLE_NAME(Primary_Key_ColumnName));

**Example:**



```
SQL> create table stude(roll_no number(3), name char(7),foreign key(roll_no) references student2(roll_no));

Table created.

SQL> insert into stude values(&roll_no,'&name');
Enter value for roll_no: 1
Enter value for name: Boss
old   1: insert into stude values(&roll_no,'&name')
new   1: insert into stude values(1,'Boss')

1 row created.

SQL> /
Enter value for roll_no: 2
Enter value for name: Deep
old   1: insert into stude values(&roll_no,'&name')
new   1: insert into stude values(2,'Deep')
insert into stude values(2,'Deep')
*
ERROR at line 1:
ORA-02291: integrity constraint (SYSTEM.SYS_C007008) violated - parent key not
found
```

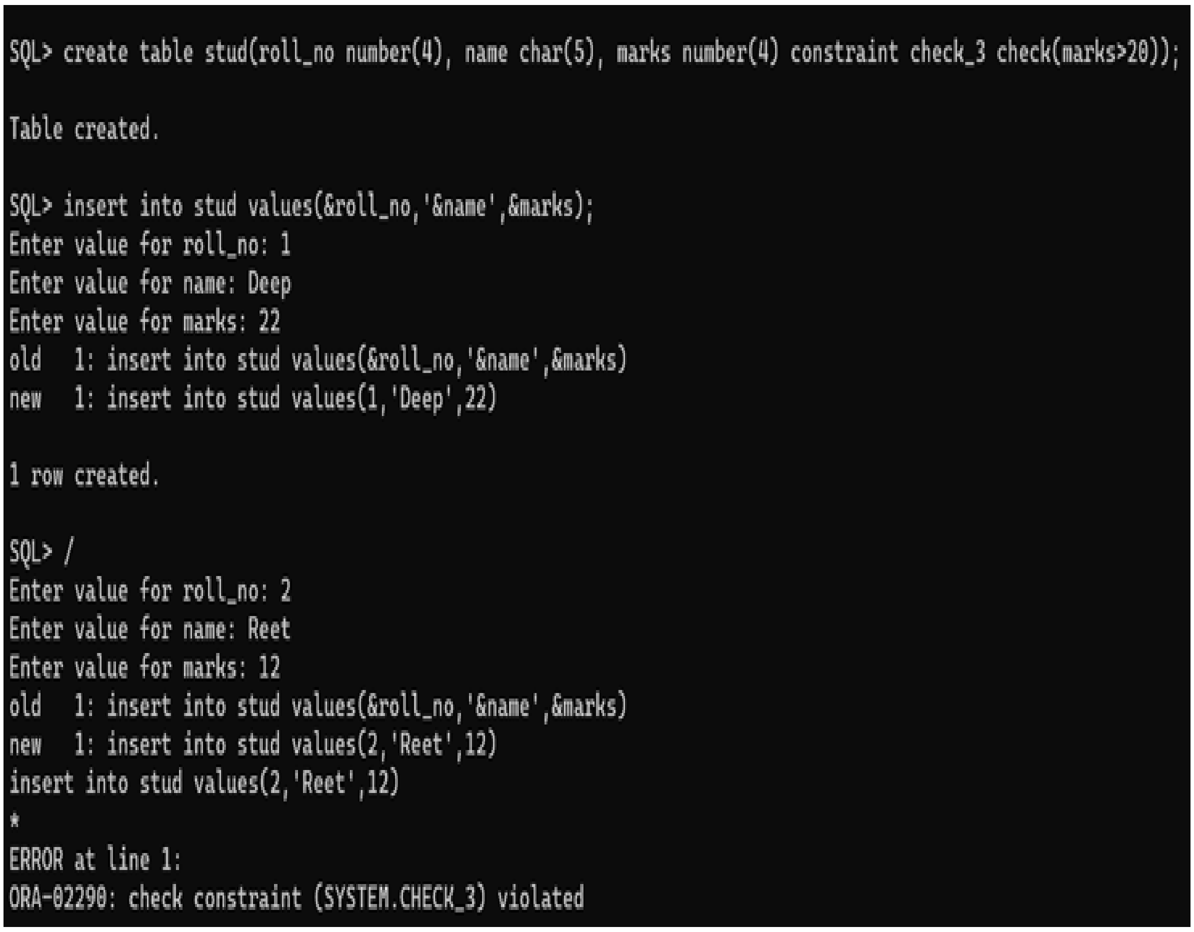
5) **CHECK:**

Using the CHECK constraint we can specify a condition for a field, which should be satisfied at the time of entering values for this field.

**Syntax:**

CREATE TABLE TableName (ColumnName1 datatype CHECK (ColumnName1 Condition), ColumnName2 datatype,…., ColumnNameN datatype);

**Example:**



```
SQL> create table stud(roll_no number(4), name char(5), marks number(4) constraint check_3 check(marks>20));

Table created.

SQL> insert into stud values(&roll_no,'&name',&marks);
Enter value for roll_no: 1
Enter value for name: Deep
Enter value for marks: 22
old   1: insert into stud values(&roll_no,'&name',&marks)
new   1: insert into stud values(1,'Deep',22)

1 row created.

SQL> /
Enter value for roll_no: 2
Enter value for name: Reet
Enter value for marks: 12
old   1: insert into stud values(&roll_no,'&name',&marks)
new   1: insert into stud values(2,'Reet',12)
insert into stud values(2,'Reet',12)
*
ERROR at line 1:
ORA-02290: check constraint (SYSTEM.CHECK_3) violated
```

# PRACTICAL 6: INTRODUCTION TO SET OPERATORS AND ITS IMPLEMENTATION IN ORACLE.

SQL set operators allow combining results from two or more SELECT statements. At first sight, this look similar to SQL joins although there is a big difference. SQL joins tends to combine columns.

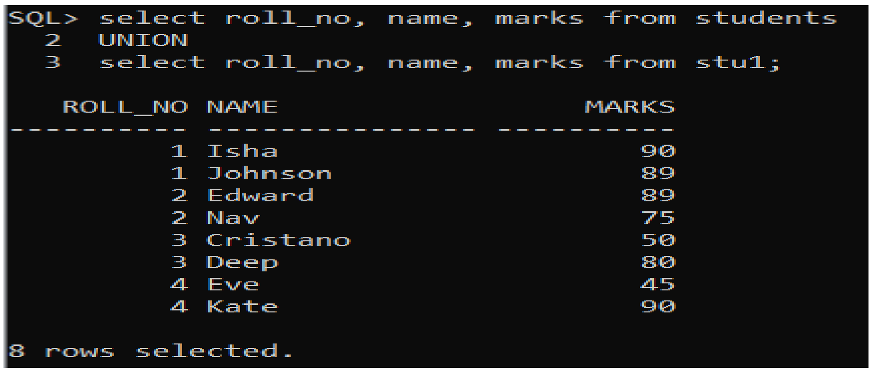
1) **UNION**

   It returns all distinct rows selected by either query.

   **Syntax:**

   SELECT column_name FROM table_name_1 UNION SELECT column_name FROM table_name_2;

   **Example:**



```
SQL> select roll_no, name, marks from students
  2  UNION
  3  select roll_no, name, marks from stu1;

   ROLL_NO NAME                  MARKS
---------- --------------- ----------
         1 Isha                     90
         1 Johnson                  89
         2 Edward                   89
         2 Nav                      75
         3 Cristano                 50
         3 Deep                     80
         4 Eve                      45
         4 Kate                     90

8 rows selected.
```

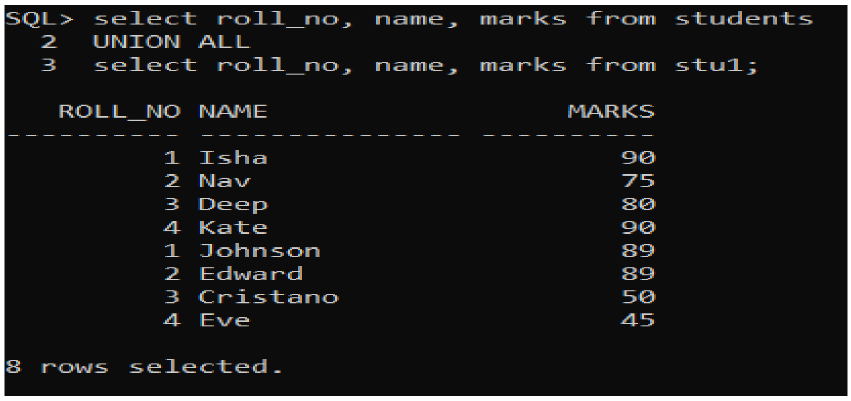
2) **UNION ALL**

   It returns all rows selected by either query, including all duplicates.

   **Syntax:**

   SELECT column_name FROM table_name_1 UNION ALL SELECT column_name FROM table_name_2;

   **Example:**



```
SQL> select roll_no, name, marks from students
  2  UNION ALL
  3  select roll_no, name, marks from stu1;

   ROLL_NO NAME                  MARKS
---------- --------------- ----------
         1 Isha                     90
         2 Nav                      75
         3 Deep                     80
         4 Kate                     90
         1 Johnson                  89
         2 Edward                   89
         3 Cristano                 50
         4 Eve                      45

8 rows selected.
```

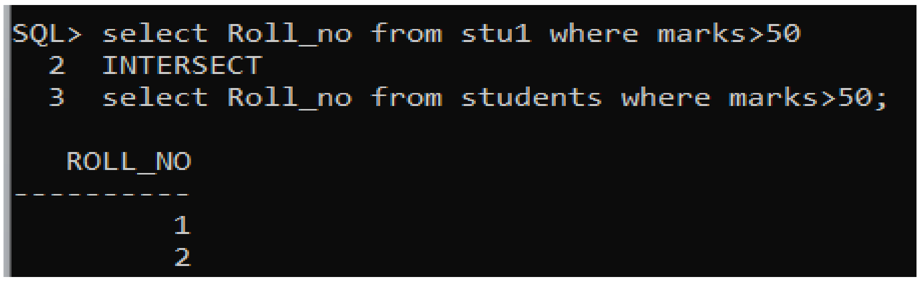
3) **INTERSECT**

It returns all distinct rows selected by both queries.

**Syntax:**

SELECT column_name FROM table_name_1 INTERSECT SELECT column_name FROM table_name_2;

**Example:**



```
SQL> select Roll_no from stu1 where marks>50
  2  INTERSECT
  3  select Roll_no from students where marks>50;

   ROLL_NO
----------
         1
         2
```

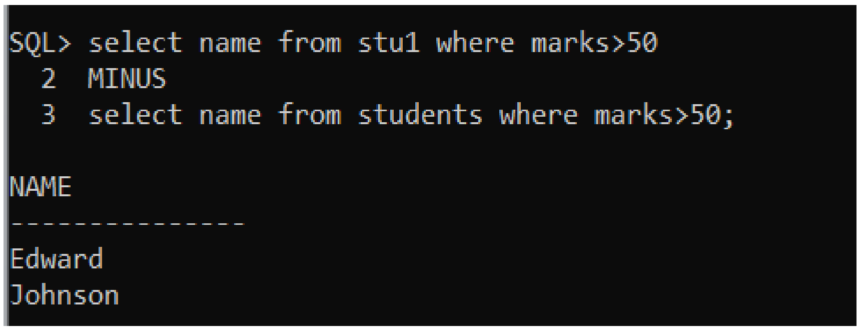
4) **MINUS**

It returns all distinct rows selected by the first query but not the second.

**Syntax:**

SELECT column_name FROM table_name_1 MINUS SELECT column_name FROM table_name_2;

**Example:**



```
SQL> select name from stu1 where marks>50
  2  MINUS
  3  select name from students where marks>50;

NAME
---------------
Edward
Johnson
```

## PRACTICAL 7: TO EXPLORE 'SELECT' STATEMENT USING WHERE, ORDER BY, BETWEEN, LIKE, GROUP BY, HAVING ETC.

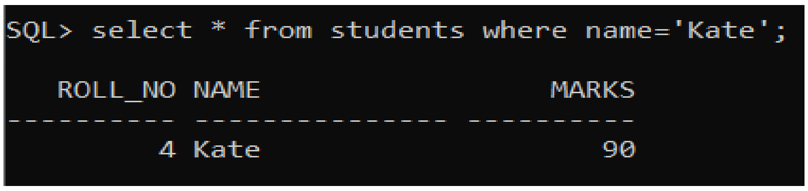
1) **WHERE**

The WHERE clause is used to filter the results obtained by the DML statements such as SELECT, UPDATE and DELETE etc. We can retrieve the data from a single table or multiple tables(after join operation) using the WHERE clause.

**Syntax:**

SELECT column1, column2,... column FROM table_name WHERE [condition];

**Example:**



```
SQL> select * from students where name='Kate';

   ROLL_NO NAME                                MARKS
---------- ---------------- -----------
         4 Kate                                   90
```

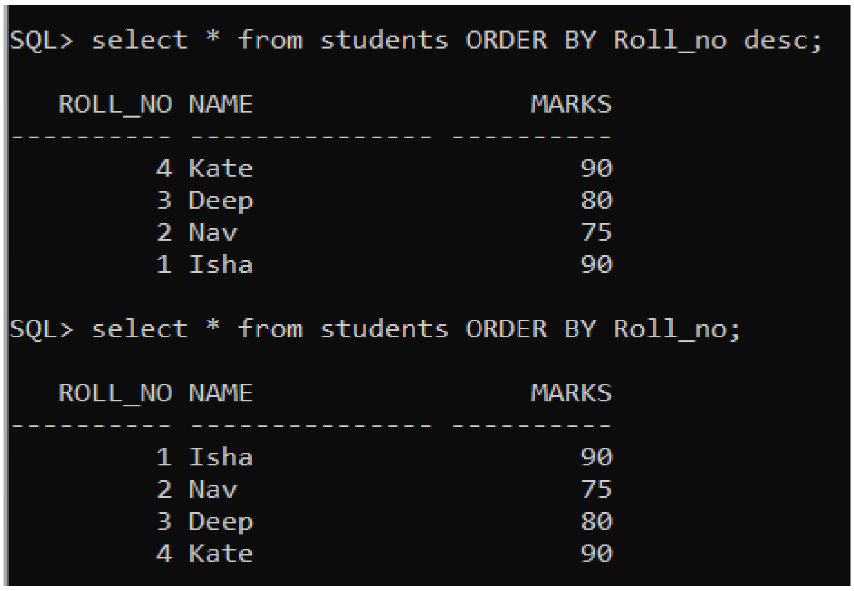
2) **ORDER BY**

The ORDER BY clause is used to sort the data in either ascending or descending order, based on one or more columns. This clause can sort data by a single column or by multiple columns.

**Syntax:**

SELECT column-list FROM table_name [ORDER BY column1, column2, .. columnN] [ASC | DESC];

**Example:**



```
SQL> select * from students ORDER BY Roll_no desc;

   ROLL_NO NAME                                MARKS
---------- --------------- ----------
         4 Kate                                   90
         3 Deep                                   80
         2 Nav                                    75
         1 Isha                                   90

SQL> select * from students ORDER BY Roll_no;

   ROLL_NO NAME                                MARKS
---------- --------------- ----------
         1 Isha                                   90
         2 Nav                                    75
         3 Deep                                   80
         4 Kate                                   90
```

### 3) GROUP BY

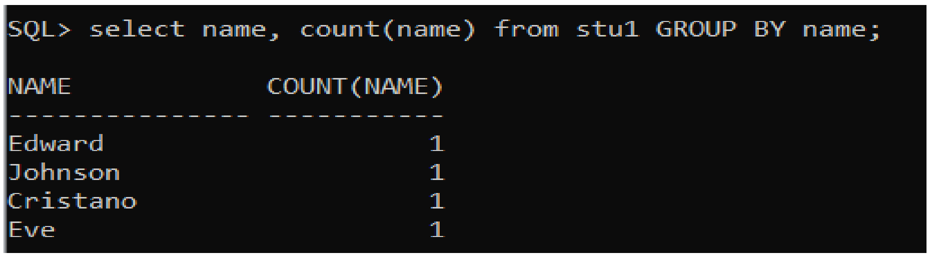
GROUP BY clause is used in conjunction with the SELECT statement to arrange identical data into groups. This clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY and HAVING clauses (if they exist).

The main purpose of grouping the records of a table based on particular columns is to perform calculations on these groups. Therefore, The GROUP BY clause is typically used with aggregate functions such as SUM(), COUNT(), AVG(), MAX(), MIN() etc.

**<u>Syntax:</u>**

SELECT column_name(s) FROM table_name GROUP BY column_name(s);

**<u>Example:</u>**



```
SQL> select name, count(name) from stu1 GROUP BY name;

NAME            COUNT(NAME)
--------------- -----------
Edward                    1
Johnson                   1
Cristano                  1
Eve                       1
```

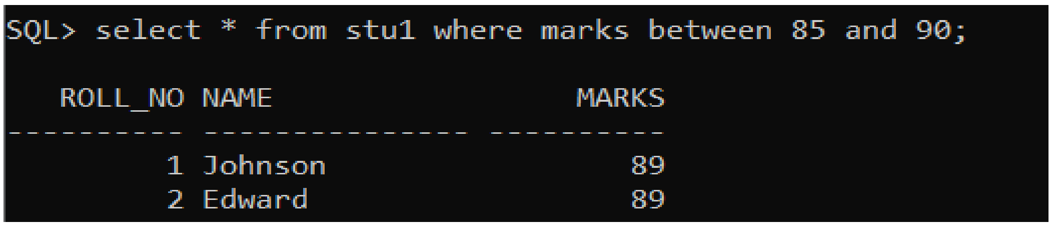
### 4) BETWEEN

The BETWEEN operator selects values within a given range. The values can be numbers, text or dates.

**<u>Syntax:</u>**

SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2;

**<u>Example:</u>**



```
SQL> select * from stu1 where marks between 85 and 90;

   ROLL_NO NAME                 MARKS
---------- --------------- ----------
         1 Johnson                 89
         2 Edward                  89
```

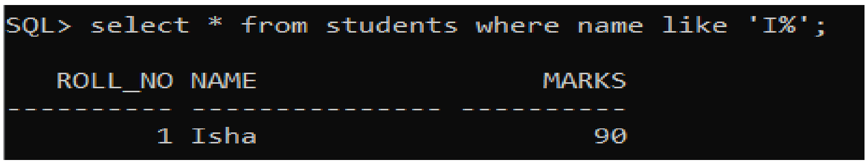
### 5) LIKE

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

**<u>Syntax:</u>**

SELECT col_name(s) FROM table_name WHERE col_name LIKE specifiedPattern;

**Example:**



```
SQL> select * from students where name like 'I%';

   ROLL_NO NAME                      MARKS
---------- --------------- ----------
         1 Isha                         90
```

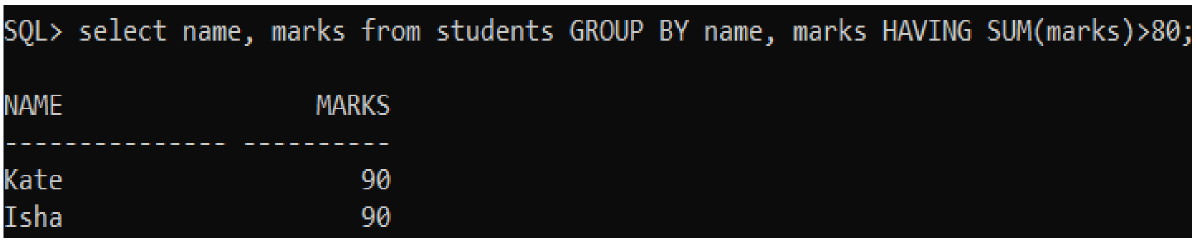
6) **HAVING**

The HAVING clause is then applied to the rows in the result set. Only the groups that meet the HAVING conditions appear in the query output. You can apply a HAVING clause only to columns that also appear in the GROUP BY clause or in an aggregate function.

**Syntax:**

SELECT column1, column2, aggregate_function(column) FROM table_name GROUP BY column1, column2 HAVING condition;

**Example:**



```
SQL> select name, marks from students GROUP BY name, marks HAVING SUM(marks)>80;

NAME                 MARKS
--------------- ----------
Kate                    90
Isha                    90
```

## PRACTICAL 8: CREATE AND USING SEQUENCES IN SQL.

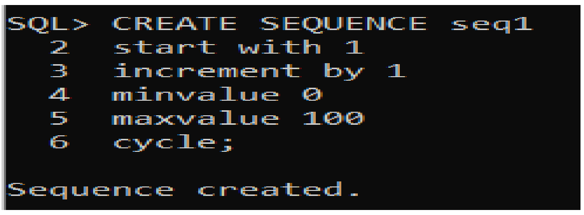
Sequences in SQL are database objects that generate a sequence of unique integer values. They are frequently used in databases because many applications require that each row in a table must contain unique values and sequences provide an easy way to generate them.

**Syntax:**

CREATE SEQUENCE sequence_name START WITH initial_value INCREMENT BY increment_value MINVALUE minimum value MAXVALUE maximum value CYCLE|NOCYCLE ;

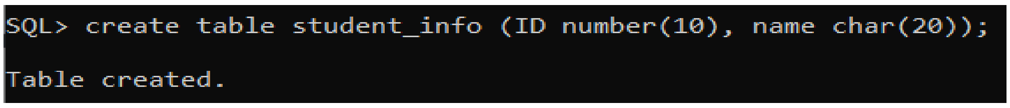
**Example:**

Create a sequence named seq1. The sequence will start from 1 and will be incremented by 1 having maximum value of 100. The sequence will repeat itself from the start value after exceeding 100.

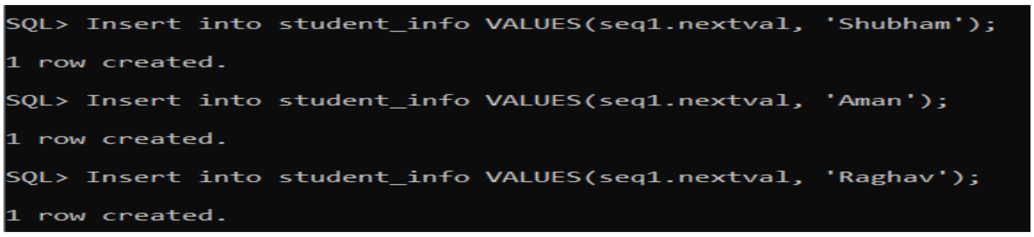


```
SQL> CREATE SEQUENCE seq1
  2  start with 1
  3  increment by 1
  4  minvalue 0
  5  maxvalue 100
  6  cycle;

Sequence created.
```

Create a table named student_info with columns as id and name and insert values in it.



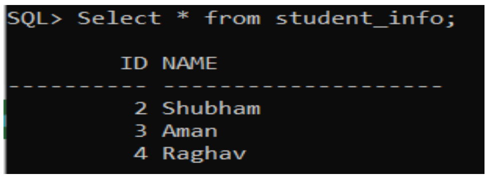
```
SQL> create table student_info (ID number(10), name char(20));

Table created.
```



```
SQL> Insert into student_info VALUES(seq1.nextval, 'Shubham');
1 row created.
SQL> Insert into student_info VALUES(seq1.nextval, 'Aman');
1 row created.
SQL> Insert into student_info VALUES(seq1.nextval, 'Raghav');
1 row created.
```

where seq1.nextval will insert id's in the id column in a sequence as defined in seq1.

**OUTPUT:**



```
SQL> Select * from student_info;

        ID NAME
---------- --------------------
         2 Shubham
         3 Aman
         4 Raghav
```

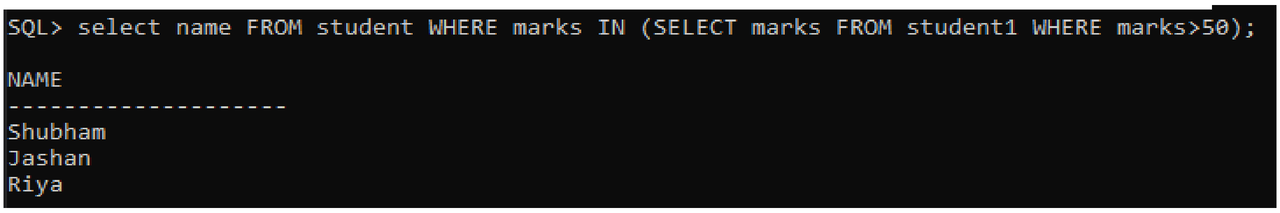
## PRACTICAL 9: IMPEMENTATION OF NESTED QUERIES IN SQL.

Nested queries are a way to perform more complex queries by embedding one query within another. A nested query is a query that appears inside another query, and it helps retrieve data from multiple tables or apply conditions based on the results of another query. The result of inner query is used in execution of outer query.

**Syntax:**

SELECT column1, column2, ... FROM table1 WHERE column1 IN ( SELECT column1 FROM table2 WHERE condition );

**Example:**



```
SQL> select name FROM student WHERE marks IN (SELECT marks FROM student1 WHERE marks>50);

NAME
--------------------
Shubham
Jashan
Riya
```

**Types:**

1. **Non-correlated (or Independent) Nested Queries**: Non-correlated subqueries are executed independently of the outer query. Their results are passed to the outer query.

   **Execution Order in Independent Nested Queries**

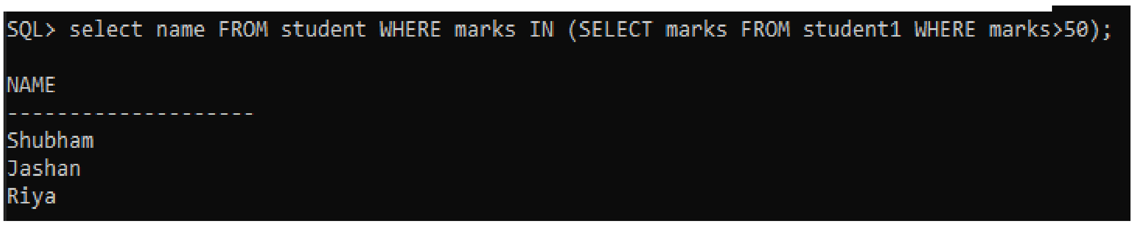
   In independent nested queries, the execution order is from the innermost query to the outer query. An outer query won't be executed until its inner query completes its execution. The outer query uses the result of the inner query.

   **Operators Used in Independent Nested Queries**

   - **IN Operator**

     This operator checks if a column value in the outer query's result is present in the inner query's result. The final result will have rows that satisfy the IN condition.

     **Example:**

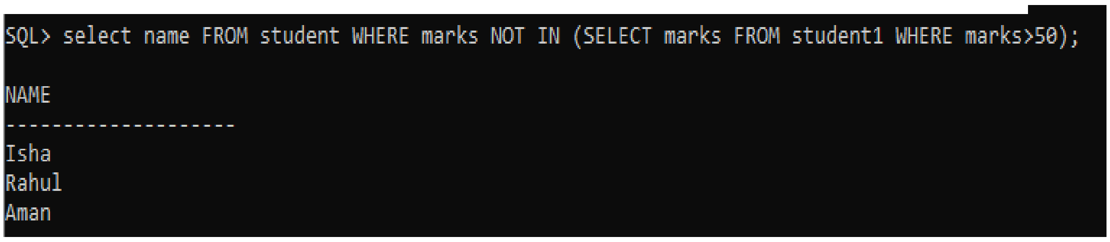


```
SQL> select name FROM student WHERE marks IN (SELECT marks FROM student1 WHERE marks>50);

NAME
--------------------
Shubham
Jashan
Riya
```

- **NOT IN Operator**

  This operator checks if a column value in the outer query's result is not present in the inner query's result. The final result will have rows that satisfy the NOT IN condition.

  **Example:**

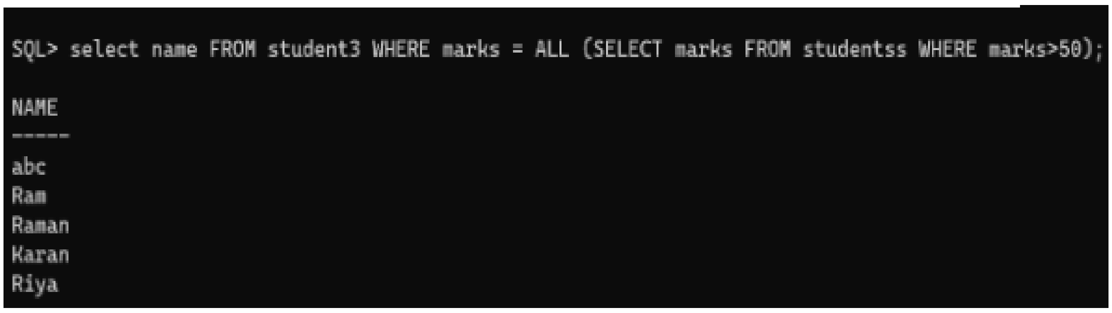


```
SQL> select name FROM student WHERE marks NOT IN (SELECT marks FROM student1 WHERE marks>50);

NAME
--------------------
Isha
Rahul
Aman
```

- **ALL Operator**

  This operator compares a value of the outer query's result with all the values of the inner query's result and returns the row if it matches all the values.

  **Example:**

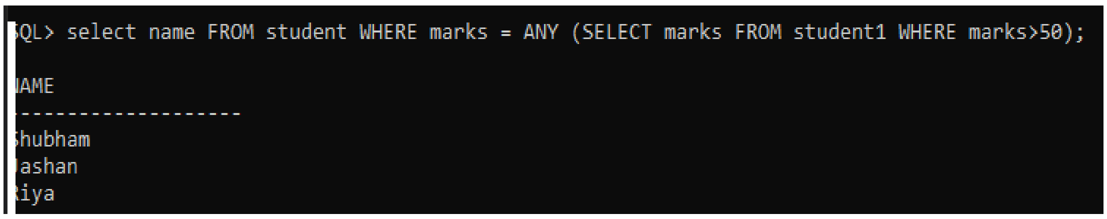


```
SQL> select name FROM student3 WHERE marks = ALL (SELECT marks FROM studentss WHERE marks>50);

NAME
-----
abc
Ram
Raman
Karan
Riya
```

- **ANY Operator**

  This operator compares a value of the outer query's result with all the inner query's result values and returns the row if there is a match with any value.

  **Example:**



```
QL> select name FROM student WHERE marks = ANY (SELECT marks FROM student1 WHERE marks>50);

AME
--------------------
hubham
ashan
iya
```

2. **Correlated Nested Queries:** Correlated subqueries are executed once for each row of the outer query. They use values from the outer query to return results.

   **Execution Order in Co-related Nested Queries**

   - In correlated nested queries, the inner query uses values from the outer query, and the execution order is different from that of independent nested queries.
   - First, the outer query selects the first row.
   - Inner query uses the value of the selected row. It executes its query and returns a result set.
   - Outer query uses the result set returned by the inner query. It determines whether the selected row should be included in the final output.
   - Steps 2 and 3 are repeated for each row in the outer query's result set.
   - This process can be resource-intensive. It may lead to performance issues if the query is not optimized properly.

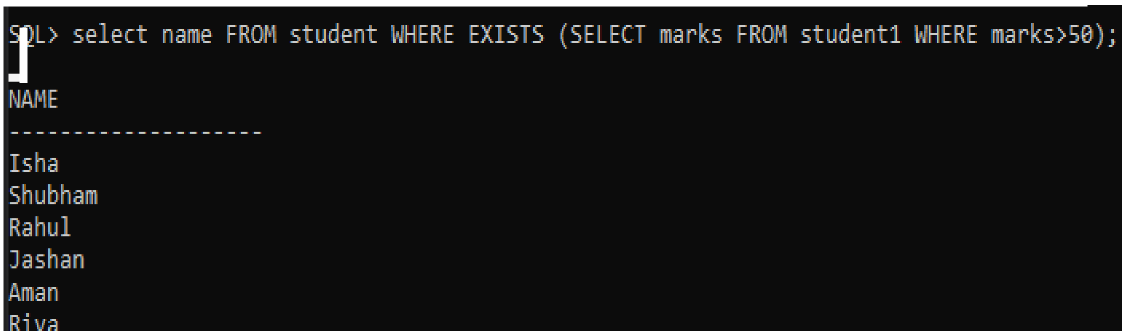
   **Operators Used in Co-related Nested Queries**

   In co-related nested queries, the following operators can be used

   - **EXISTS Operator**

     This operator checks whether a subquery returns any row. If it returns at least one row. EXISTS operator returns true, and the outer query continues to execute. If the subquery returns no row, the EXISTS operator returns false, and the outer query stops execution.

     **Example:**

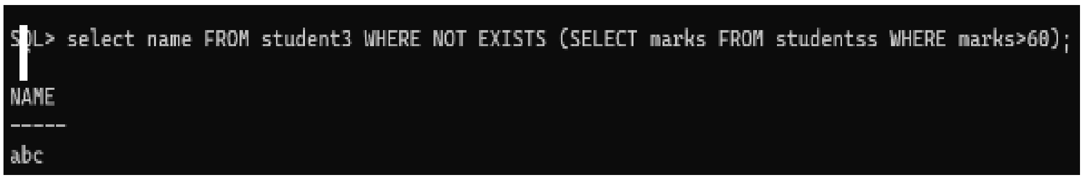


```
SQL> select name FROM student WHERE EXISTS (SELECT marks FROM student1 WHERE marks>50);

NAME
--------------------
Isha
Shubham
Rahul
Jashan
Aman
Riya
```

   - **NOT EXISTS Operator**

     This operator checks whether a subquery returns no rows. If the subquery returns no row, the NOT EXISTS operator returns true, and the outer query continues to execute. If the subquery returns at least one row, the NOT EXISTS operator returns false, and the outer query stops execution.

**Example:**



```
SQL> select name FROM student3 WHERE NOT EXISTS (SELECT marks FROM studentss WHERE marks>60);

NAME
-----
abc
```

These operators are used to create co-related nested queries that depend on values from the outer query for execution.

## PRACTICAL 10: IMPLEMENTATION THE CONCEPT OF JOINS.

SQL join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of joins are as follow :

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- NATURAL JOIN

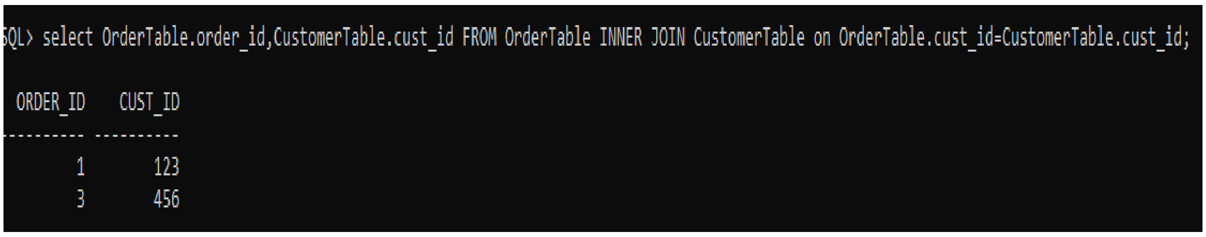
1) **INNER JOIN**

The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 INNER JOIN table2 ON table1.matching_column = table2.matching_column;

**Example:**



```
SQL> select OrderTable.order_id,CustomerTable.cust_id FROM OrderTable INNER JOIN CustomerTable on OrderTable.cust_id=CustomerTable.cust_id;

  ORDER_ID    CUST_ID
---------- ----------
         1        123
         3        456
```

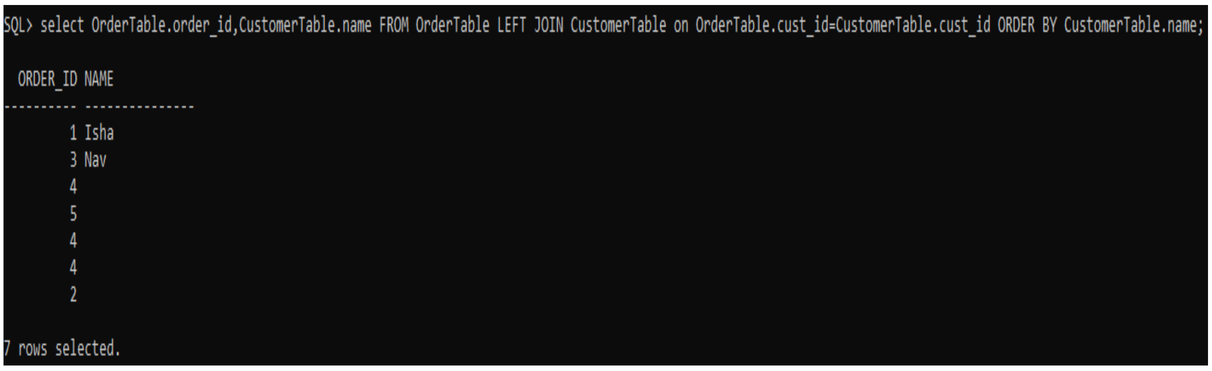
2) **LEFT JOIN**

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 LEFT JOIN table2 ON table1.matching_column = table2.matching_column;

**Example:**



```
SQL> select OrderTable.order_id,CustomerTable.name FROM OrderTable LEFT JOIN CustomerTable on OrderTable.cust_id=CustomerTable.cust_id ORDER BY CustomerTable.name;

  ORDER_ID NAME
---------- --------------
         1 Isha
         3 Nav
         4
         5
         4
         4
         2

7 rows selected.
```

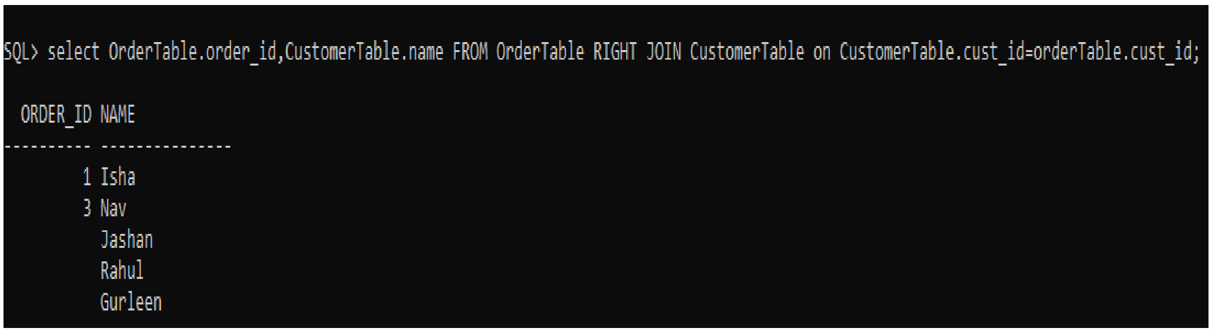
3) **RIGHT JOIN**

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 RIGHT JOIN table2 ON table1.matching_column = table2.matching_column;

**Example:**



```
SQL> select OrderTable.order_id,CustomerTable.name FROM OrderTable RIGHT JOIN CustomerTable on CustomerTable.cust_id=orderTable.cust_id;

  ORDER_ID NAME
---------- --------------
         1 Isha
         3 Nav
           Jashan
           Rahul
           Gurleen
```

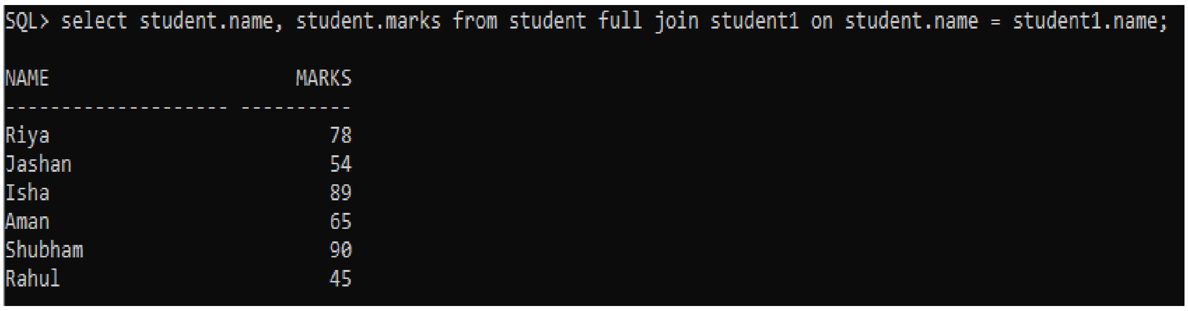
4) **FULL JOIN**

FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain NULL values.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 FULL JOIN table2 ON table1.matching_column = table2.matching_column;

**Example:**



```
SQL> select student.name, student.marks from student full join student1 on student.name = student1.name;

NAME                      MARKS
-------------------- ----------
Riya                         78
Jashan                       54
Isha                         89
Aman                         65
Shubham                      90
Rahul                        45
```

5) **NATURAL JOIN**

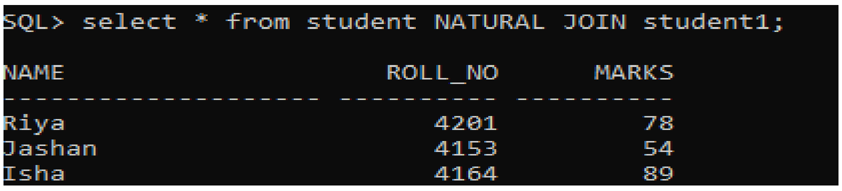
Natural join can join tables based on the common columns in the tables being joined. A natural join returns all rows by matching values in common columns having same name and data type of columns and that column should be present in both tables. Both table must have at list one common column with same column name and same data type. The two table are joined using Cross join.

DBMS will look for a common column with same name and data type Tuples having exactly same values in common columns are kept in result.

**Syntax:**

SELECT * FROM TABLE1 NATURAL JOIN TABLE2;

**Example:**



```
SQL> select * from student NATURAL JOIN student1;

NAME                    ROLL_NO      MARKS
-------------------- ---------- ----------
Riya                       4201         78
Jashan                     4153         54
Isha                       4164         89
```

# PRACTICAL 11: IMPLEMENTATION TO VIEWS AND INDEXES.

## 1. Views

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A view can either have all the rows of a table or specific rows based on certain condition.

**Creating view from single table:**

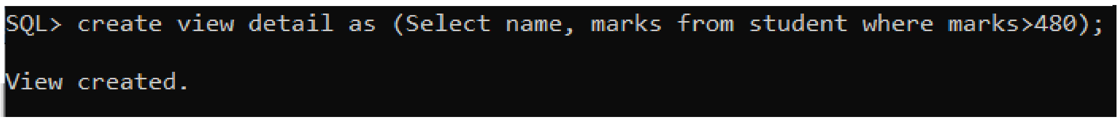
We can create View using CREATE VIEW statement. A View can be created from a single table.

**Syntax:**

CREATE VIEW view_name AS SELECT column1, column2..... FROM table_name WHERE condition;

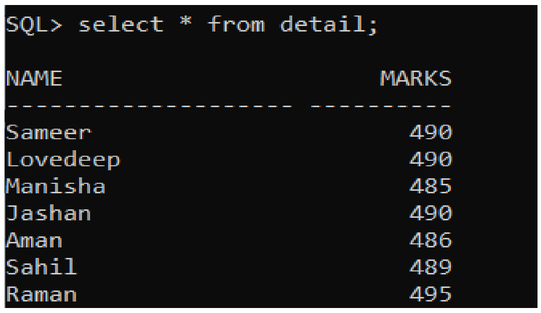
**Example:**

In this example we will create a View named detail from the table student.



```
SQL> create view detail as (Select name, marks from student where marks>480);

View created.
```

To see the data in the View, we can query the view in the same manner as we query a table.



```
SQL> select * from detail;

NAME                      MARKS
-------------------- ----------
Sameer                      490
Lovedeep                    490
Manisha                     485
Jashan                      490
Aman                        486
Sahil                       489
Raman                       495
```

**Creating view from multiple tables:**

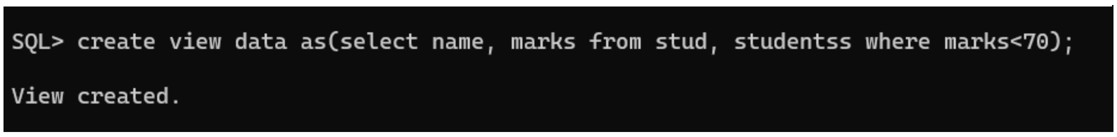
We can create View using CREATE VIEW statement. A View can be created from a multiple tables.

**Syntax:**

CREATE VIEW view_name AS SELECT column1, column2..... FROM table_name1, stable_name2,….. WHERE condition;

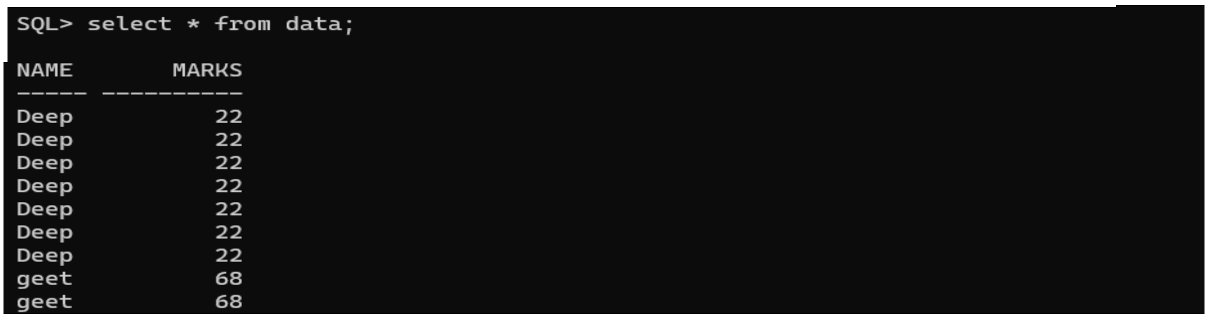
**Example:**

In this example we will create a View named data from two tables stud and studentss. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.



```
SQL> create view data as(select name, marks from stud, studentss where marks<70);

View created.
```

To see the data in the View, we can query the view in the same manner as we query a table.



```
SQL> select * from data;

NAME       MARKS
----- ----------
Deep          22
Deep          22
Deep          22
Deep          22
Deep          22
Deep          22
Deep          22
geet          68
geet          68
```

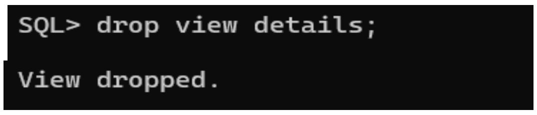
**Deleting a View:**

When a view no longer useful you may drop the view permanently. Also if a view needs change within it, it would be dropped and then created again with changes in appropriate places.

**Syntax:**

DROP VIEW view_name;

**Example:**



```
SQL> drop view details;

View dropped.
```

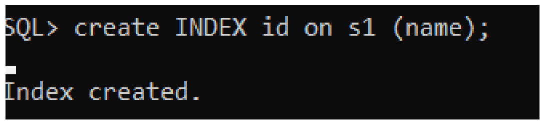
2. **Indexes**

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries. An index is a schema object. It is used by the server to speed up the retrieval of rows by using a pointer. It can reduce disk I/O(input/output) by using a rapid path access method to locate data quickly. An index helps to speed up select queries and where clauses, but it slows down data input, with the update and the insert statements. Indexes can be created or dropped with no effect on the data.

**Syntax:**

CREATE INDEX index_name ON table_name;

**Example:**



```
SQL> create INDEX id on s1 (name);

Index created.
```

**Types of Indexes:**

There are various types of indexes that can be created using the CREATE INDEX statement. They are:

- Unique Index
- Single-Column Index
- Composite Index

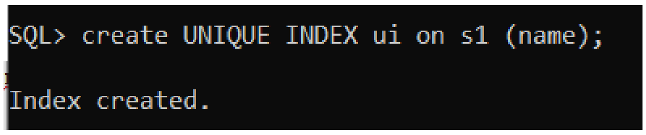
1. **Unique Indexes**

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. It is automatically created by PRIMARY and UNIQUE constraints when they are applied on a database table, in order to prevent the user from inserting duplicate values into the indexed table column(s). The basic syntax is as follows.

**Syntax:**

CREATE UNIQUE INDEX index_name on table_name (column_name);

**Example:**



```
SQL> create UNIQUE INDEX ui on s1 (name);

Index created.
```

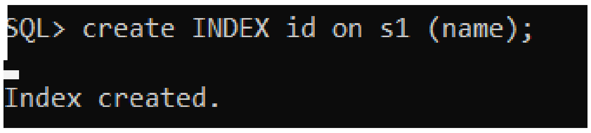
2. **Single-Column Indexes**

A single-column index is created only on one table column. The syntax is as follows.

**Syntax:**

CREATE INDEX index_name ON table_name (column_name);

**Example:**



```
SQL> create INDEX id on s1 (name);

Index created.
```

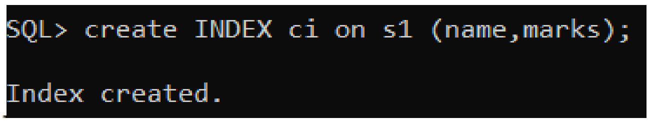
## 3. Composite Indexes

A composite index is an index that can be created on two or more columns of a table. Its basic syntax is as follows.

**Syntax:**

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

**Example:**



```
SQL> create INDEX ci on s1 (name,marks);

Index created.
```

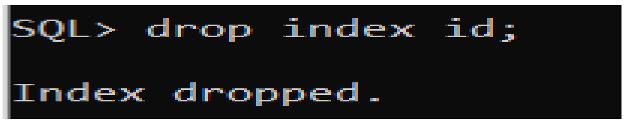
**DROP INDEX :**

An index can be dropped using SQL DROP command. Dropping an index can effect the query performance in a database. Thus, an index needs to be dropped only when it is absolutely necessary. The basic syntax is as follows.

**Syntax:**

DROP INDEX index_name;

**Example:**



```
SQL> drop index id;

Index dropped.
```

## PRACTICAL 12: IMPLEMENTATION TO DATABASE SECURITY AND PRIVILEGES. ALSO IMPLEMENT GRANT AND REVOKE COMMANDS IN ORACLE.

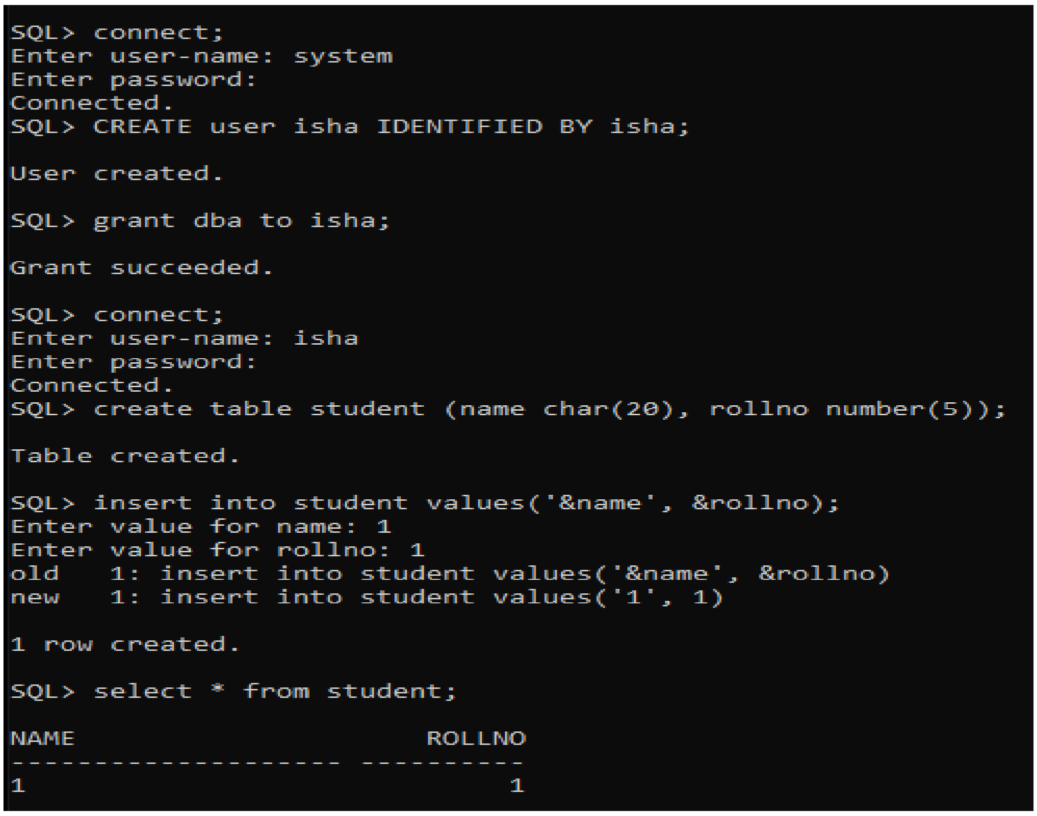
### GRANT:

This command allows the administrator to provide privileges or permissions over a database object, such as a table, view, or procedure. It can provide user access to perform certain database or component operations. In simple language, the GRANT command allows the user to implement other SQL commands on the database or its objects. The primary function of the GRANT command in SQL is to provide administrators with the ability to ensure the security and integrity of the data is maintained in the database.

**Syntax:**

grant privilege_name on object_name to {user_name | public | role_name}
Here privilege_name is which permission has to be granted, object_name is the name of the database object, user_name is the user to which access should be provided, the public is used to permit access to all the users.

**Example:**



```
SQL> connect;
Enter user-name: system
Enter password:
Connected.
SQL> CREATE user isha IDENTIFIED BY isha;

User created.

SQL> grant dba to isha;

Grant succeeded.

SQL> connect;
Enter user-name: isha
Enter password:
Connected.
SQL> create table student (name char(20), rollno number(5));

Table created.

SQL> insert into student values('&name', &rollno);
Enter value for name: 1
Enter value for rollno: 1
old   1: insert into student values('&name', &rollno)
new   1: insert into student values('1', 1)

1 row created.

SQL> select * from student;

NAME                     ROLLNO
-------------------- ----------
1                             1
```

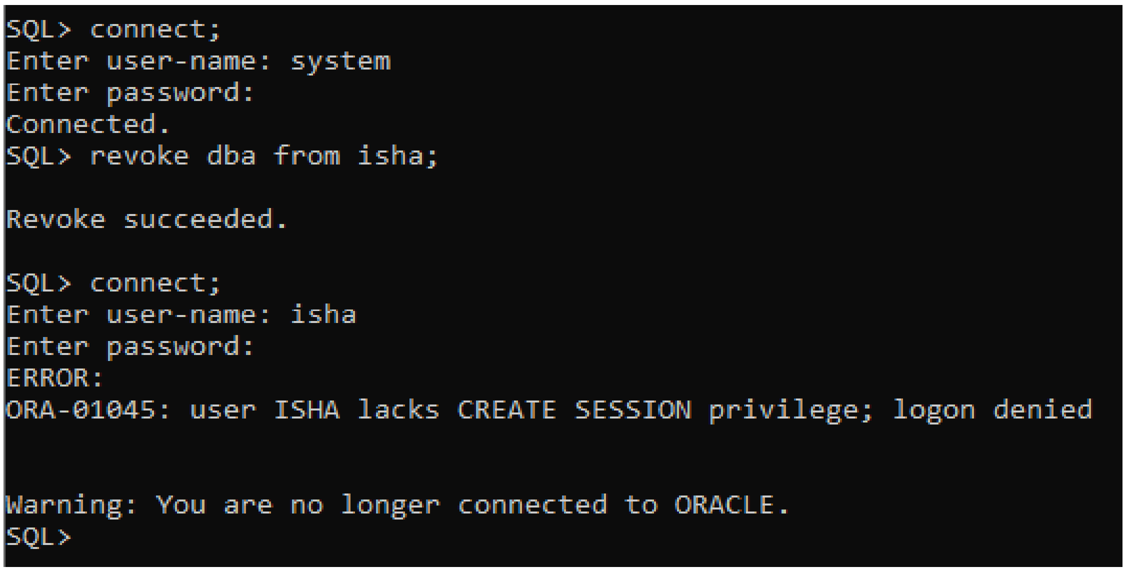
**REVOKE:**

As the name suggests, revoke is to take away. The REVOKE command enables the database administrator to remove the previously provided privileges or permissions from a user over a database or database object, such as a table, view, or procedure. The REVOKE commands prevent the user from accessing or performing a specific operation on an element in the database. In simple language, the REVOKE command terminates the ability of the user to perform the mentioned SQL command in the REVOKE query on the database or its component. The primary reason for implementing the REVOKE query in the database is to ensure the data's security and integrity.

**Syntax:**

revoke privilege_name on object_name from {user_name | public | role_name}

**Example:**



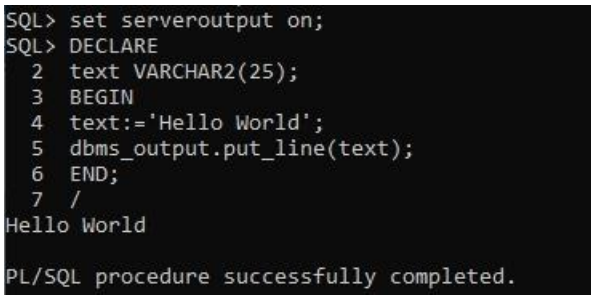
```
SQL> connect;
Enter user-name: system
Enter password:
Connected.
SQL> revoke dba from isha;

Revoke succeeded.

SQL> connect;
Enter user-name: isha
Enter password:
ERROR:
ORA-01045: user ISHA lacks CREATE SESSION privilege; logon denied


Warning: You are no longer connected to ORACLE.
SQL>
```

## PRACTICAL 13: IMPLEMENTATION TO PL/SQL USING A PL/SQL PROGRAM.

PL/SQL stands for Procedural Language extensions to the Structured Query Language (SQL).PL/SQL is a combination of SQL along with the procedural features of programming languages. Oracle uses a PL/SQL engine to processes the PL/SQL statements. PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements. All the statements of a block are passed to oracle engine all at once which increases processing speed and decreases the traffic.

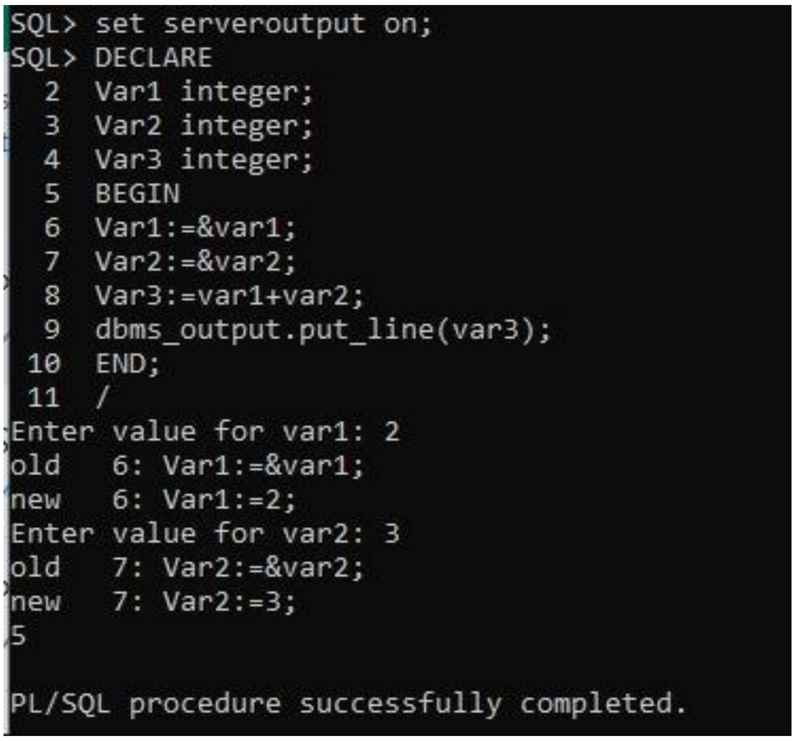
**Example:**

1) **Write a program to print hello worl using PL/SQL.**



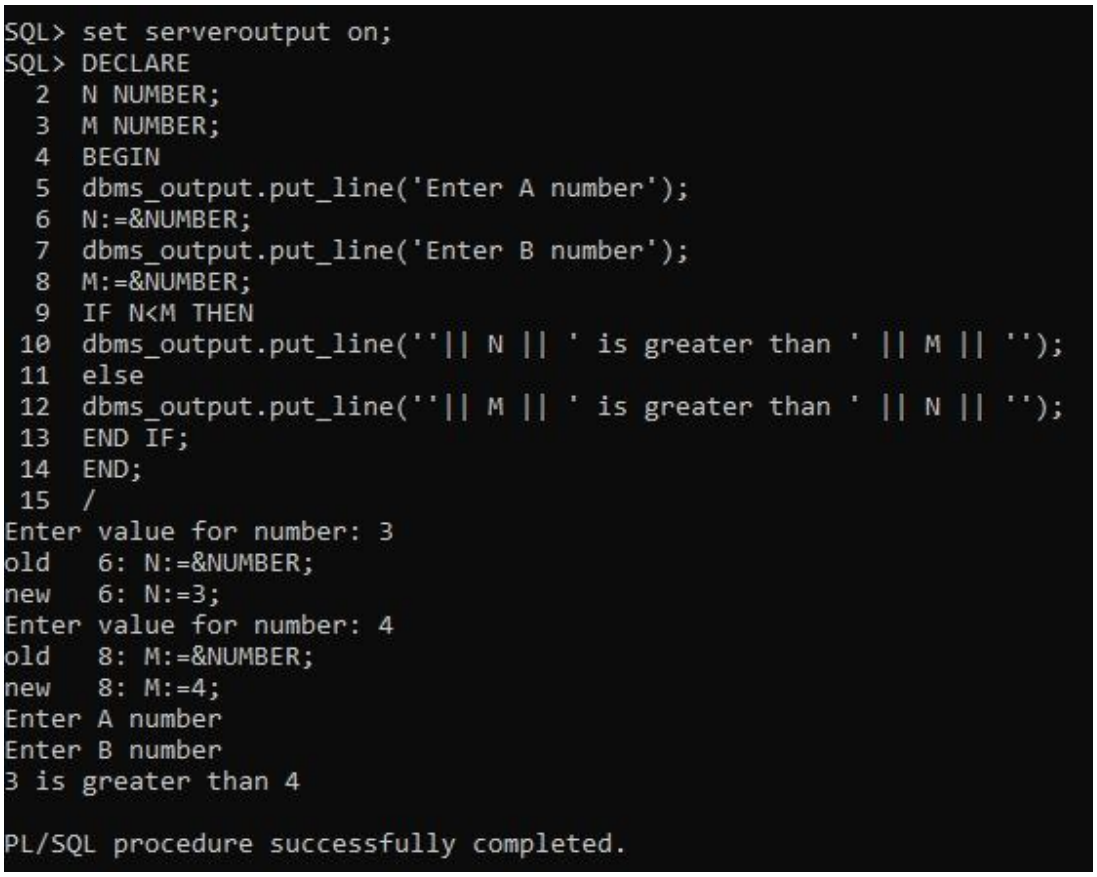
```
SQL> set serveroutput on;
SQL> DECLARE
  2  text VARCHAR2(25);
  3  BEGIN
  4  text:='Hello World';
  5  dbms_output.put_line(text);
  6  END;
  7  /
Hello World

PL/SQL procedure successfully completed.
```

2) **Write a program to add two numbers using PL/SQL.**



```
SQL> set serveroutput on;
SQL> DECLARE
  2  Var1 integer;
  3  Var2 integer;
  4  Var3 integer;
  5  BEGIN
  6  Var1:=&var1;
  7  Var2:=&var2;
  8  Var3:=var1+var2;
  9  dbms_output.put_line(var3);
 10  END;
 11  /
Enter value for var1: 2
old   6: Var1:=&var1;
new   6: Var1:=2;
Enter value for var2: 3
old   7: Var2:=&var2;
new   7: Var2:=3;
5

PL/SQL procedure successfully completed.
```

3) **Write a program to find largest of two numbers using PL/SQL.**



```
SQL> set serveroutput on;
SQL> DECLARE
  2  N NUMBER;
  3  M NUMBER;
  4  BEGIN
  5  dbms_output.put_line('Enter A number');
  6  N:=&NUMBER;
  7  dbms_output.put_line('Enter B number');
  8  M:=&NUMBER;
  9  IF N<M THEN
 10  dbms_output.put_line(''|| N || ' is greater than ' || M || '');
 11  else
 12  dbms_output.put_line(''|| M || ' is greater than ' || N || '');
 13  END IF;
 14  END;
 15  /
Enter value for number: 3
old   6: N:=&NUMBER;
new   6: N:=3;
Enter value for number: 4
old   8: M:=&NUMBER;
new   8: M:=4;
Enter A number
Enter B number
3 is greater than 4

PL/SQL procedure successfully completed.
```

## PRACTICAL 14: IMPLEMENTATION OF PROCEDURES AND TRIGGERS IN PL/SQL.