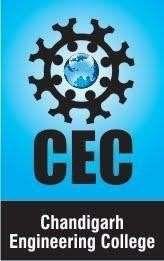
**Department of Computer Science and Engineering**

**Practical File**

**Database Management System**

**(BTCS 501-18)**

**Semester: - 5th**



**Submitted To: Submitted By:**

Dr. Gagandeep Jindal Piyush Kumar

1803187

CSE (5Z2)

CHANDIGARH ENGINEERING COLLEGE

Landran, Mohali-140307

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Practical Name** | **Page No** | **Remarks** |
| 1 | Introduction to SQL and installation of Oracle  Server | 2-14 |  |
| 2 | To study Data Types of SQL | 15-20 |  |
| 3 | To create, alter and drop a table | 21-23 |  |
| 4 | To perform DML commands on a table | 24-25 |  |
| 5 | To make use of different clauses viz. where, group by, having, order by | 26-29 |  |
| 6 | To perform aggregate functions like SUM, MAX,  MIN, AVG, COUNT on a table | 30-31 |  |
| 7 | To perform arithematic and logical operators | 32-35 |  |
| 8 | To perform single row functions: String, Numeric, Date and Conversion Functions. | 36-39 |  |
| 9 | To study set operators. | 40-41 |  |
| 10 | To study and implement constraints on a table. | 42-45 |  |
| 11 | To understand the use and working of joins. | 46-49 |  |
| 12 | To understand the use and working of Sub-Queries | 50-51 |  |
| 13 | To perform data security and privileges command: Grant and Revoke, Rollback and Commit on a table. | 52-54 |  |
| 14 | To make Views of a table and to perform Insert, Update, Select and Delete operations on View. | 55-56 |  |
| 15 | To make indexes and sequences for a table. | 57-58 |  |
| 16 | Introduction to PL/SQL Structure. | 60-61 |  |
| 17 | Compute Addition, Subtraction, Product and Division of given two numbers. | 62-63 |  |
| 18 | To Print Multiplication Table of a number. | 64 |  |
| 19 | For Loop – To Find Reverse of a given number. | 65 |  |
| 20 |  |  |  |
| 21 |  |  |  |
| 22 |  |  |  |

# INDEX PRACTICAL NO.1

**OBJECTIVE:**

Introduction to SQL and installation of Oracle Server.

**DESCRIPTION:**

D BMS

* A DBMS (Database Management System) is a software program used to manage a database. These programs enable users to access and modify database
* A DBMS is a complex set of software programs that controls the organization, storage, management, and retrieval of data in a database.
* A DBMS includes four main components, which are: Modeling Language, Data Structures, DB Query Language and Report Writer, and Transaction Mechanism.

Each of these components can be further broken down into smaller and more specific pieces, but it is the sum of these parts which are combined to create the management system around the particular database to be utilized.

* A database management system, or DBMS, gives the user access to their data and helps them transform the data into information.

A database is a structured collection of data. Data refers to the characteristics of people, things, and events.

ORACLE :

Oracle is one of the powerful RDBMS product that provide efficient solutions for database applications. Oracle is the product of Oracle Corporation which was founded by LAWRENCE ELLISIONin 1977. The first commercial product of oracle was delivered in 1970. The first version of oracle 2.0 was written in assembly language. Nowadays commonly used versions of oracle are ORACLE 8, 8i & 9i Oracle 8 and onwards provide tremendous increase in performance, features and functionality.

F EATURES OF ORACLE :

* Client/Server Architecture
* Large database and Space Management
* Concurrent Processing
* High transaction processing performance
* High Availability
* Many concurrent database users
* Compatibility

**Installing Oracle Database 10*g* on Windows**

To install the Oracle software, you must use the Oracle Universal installer.

**1.** For this installation you need either the DVDs or a downloaded version of the DVDs. In this tutorial, you install from the downloaded version. From the directory where the DVD files were unzipped, double-click setup.exe.

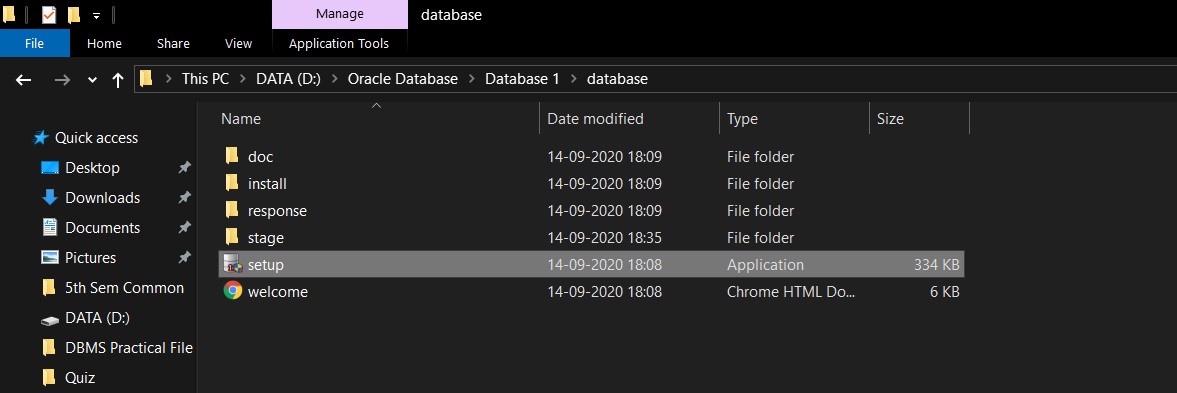


Fig. 1.1 Installation window **2.** The Oracle Universal Installer starts

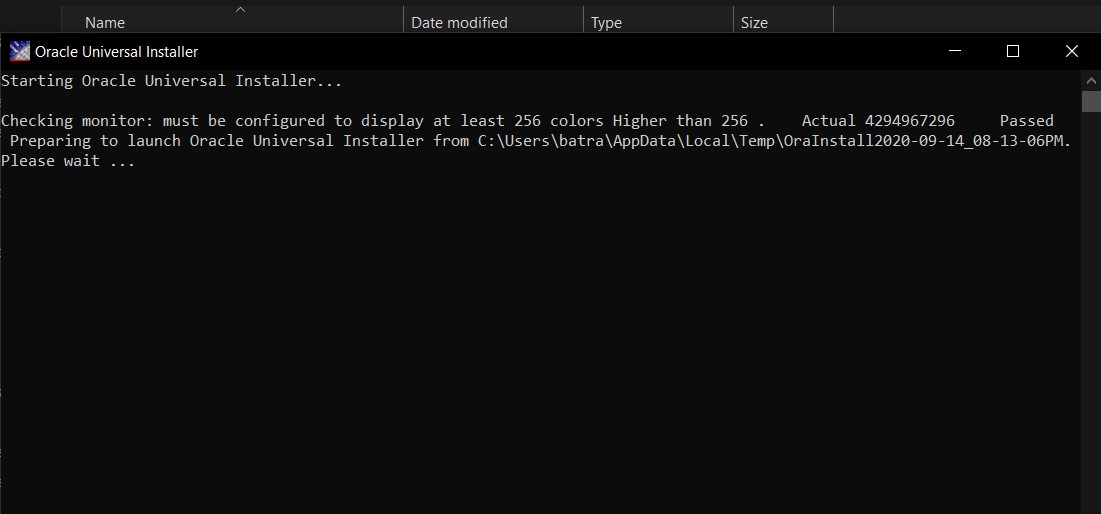


Figure 1.2 Command Prompt

1. You will perform a basic installation with a starter database.Enterorcl for the Global Database and oracle for the Database Password and Confirm Password. Then click **Next**.

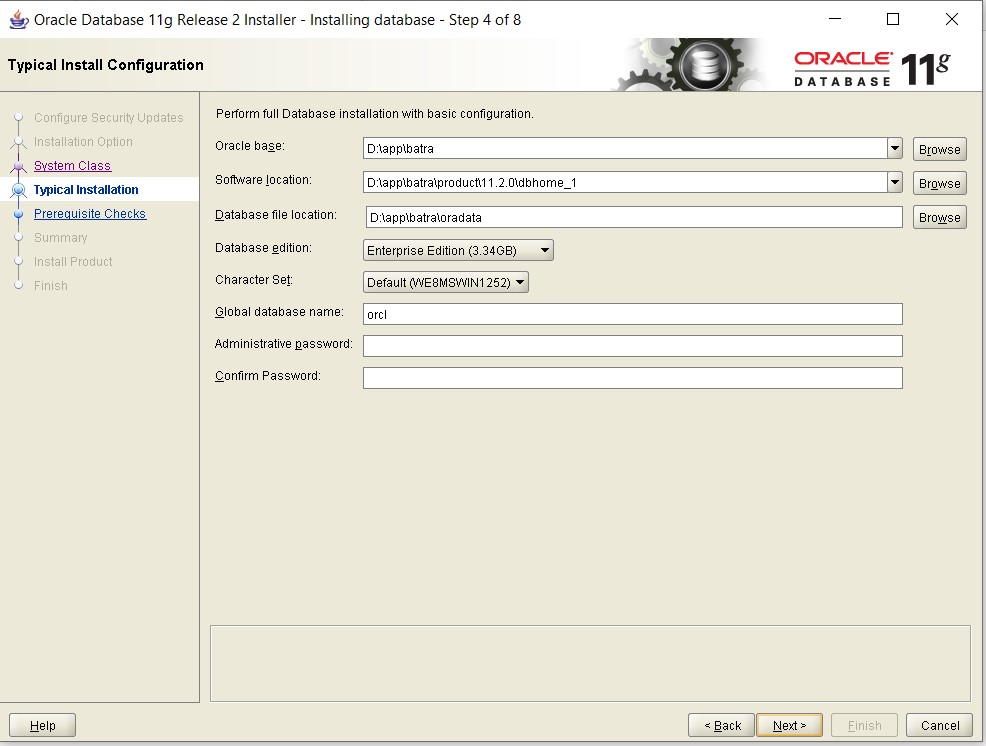
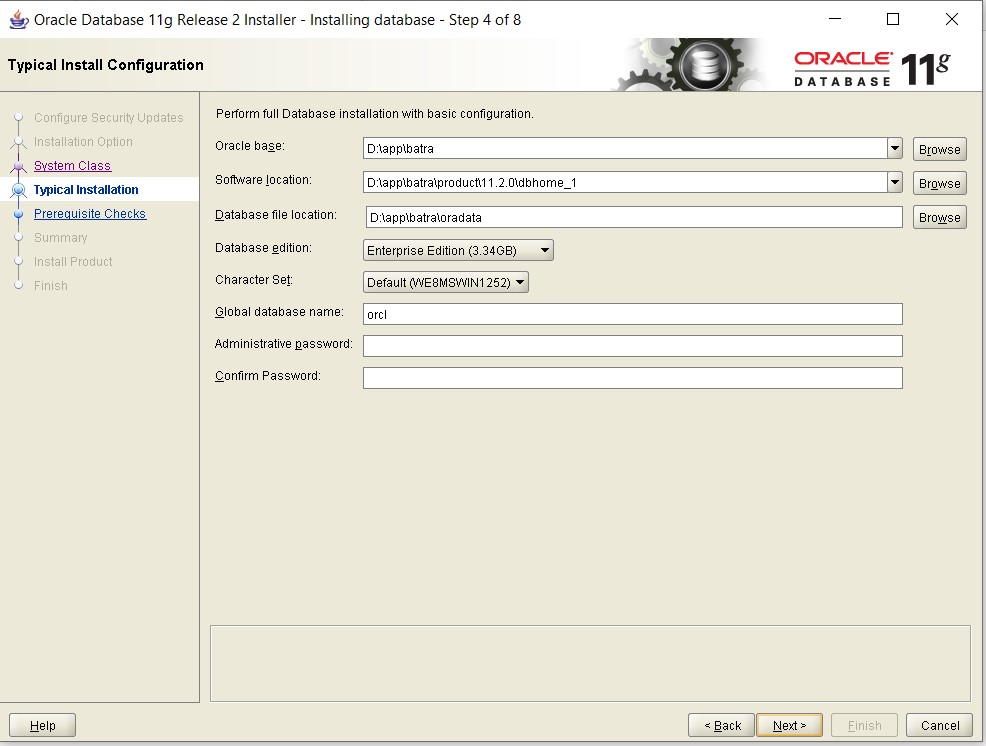
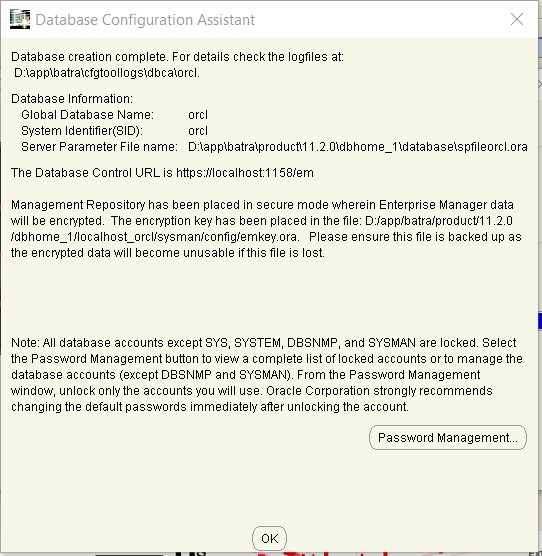


Fig. 1.3 Installation screen

1. The installer now verifies that the system meets all the minimum requirements for installing and configuring the chosen product. Please correct any reported errors (warnings are OK) before continuing. When the check successfully completes (with or without warnings), click **Next**.
2. Review the Summary window to verify what is to be installed. Then, click Install
3. When the database has been created, you can unlock the users you want to use. Click Password Management.
4. Unlock **SH**, **OE** and **HR** users by clicking on the check mark in the Lock Account? column. Enter the same name as the user in the New Password and Confirm Password fields. For example,to unlock user, enter SH in the New Password and Confirm Password fields. Then, click **OK.**



**13.** Click **Exit**.

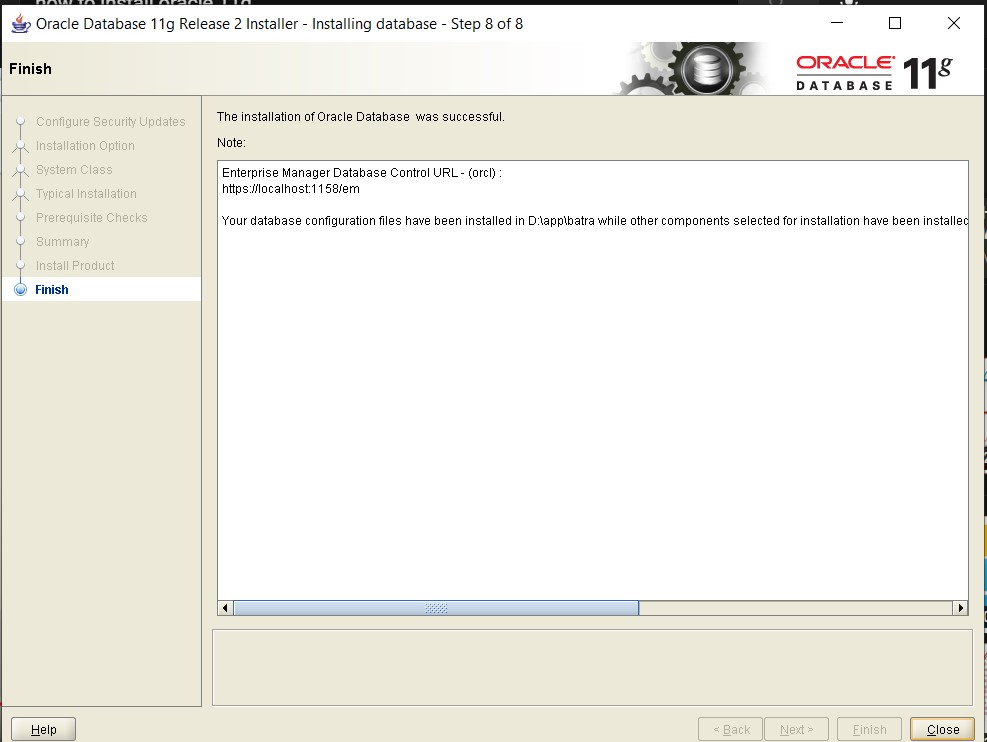


Fig. 1.13 End of Installation Screen

# PRACTICAL NO. 2

**OBJECTIVE:**

To study Datatypes of SQL

**THEORY:**

* CHAR

This data type is used to store character strings values of fixed length. The size in brackets determines the number of characters the cell can hold. The maximum number of characters (i.e. the size) this data type can hold is 255 characters. Syntax is CHAR(SIZE)

* VARCHAR

This data type is used to store variable length alphanumeric data. The maximum this data type can hold is 2000 characters. One difference between this data type and the CHAR data type is ORACLE compares. Syntax is VARCHAR(SIZE)

* NUMBER

The NUMBER data type is used to store numbers (fixed or floating point). Numbers of virtually any magnitude maybe stored up to 38 digits of precision. Numbers as large as 9.99 \* 10 to the power of 124, i.e. followed by 125 zeros can be stored.

* LONG

This data type is used to store variable length character strings containing up to 2GB. LONG data can be used to store arrays of binary data in ASCII format. LONG values cannot be indexed.

* DATE

This data type is used to represent data and time. The standard format id DD-MM-YY as in 13- JUL-85. To enter dates other than the standard format, use the appropriate functions. Date Time stores date in the 24-hour format. Syntax is DATE.

* LONG RAW

LONG RAW data types are used to store binary data, such as Digitized picture or image. LONG RAW data type can contain up to 2GB. Syntax is LONGRAW (SIZE).

* RAW

It is used to hold strings of byte oriented data. Data type can have a maximum length of 255 bytes. Syntax is RAW(SIZE)

**The following is a list of datatypes available in Oracle/PLSQL which includes**

**Character Datatypes**

Following are the **Numeric Datatypes** in Oracle/PLSQl:

|  |  |  |
| --- | --- | --- |
| Data Type  Syntax | Oracle 11g | Explanation |
| number(p,s) | Precision can range from 1 to 38.  Scale can range from -84 to 127. | Where ***p*** is the precision and ***s*** is the scale. |
| numeric(p,s) | Precision can range from 1 to 38. | Where ***p*** is the precision and ***s*** is the scale. |
| dec(p,s) | Precision can range from 1 to 38. | Where ***p*** is the precision and ***s*** is the scale. |
| decimal(p,s) | Precision can range from 1 to 38. | Where ***p*** is the precision and ***s*** is the scale. |

**Date/Time Datatypes**

Following are the **Date/Time Datatypes** in Oracle/PLSQl:

|  |  |  |
| --- | --- | --- |
| Data Type Syntax | Oracle 11g | Explanation |
|  | A date between Jan 1, |  |
| Date | 4712 BC and Dec 31, 9999 AD. |  |
| timestamp (*fractional seconds precision*) | ***fractional seconds precision*** must be a number between 0 and 9.  (default is  6) | Includes year, month, day, hour, minute, and seconds. |
| timestamp (*fractional seconds precision*) with time zone | ***fractional seconds precision*** must be a number between 0 and 9.  (default is  6) | Includes year, month, day, hour, minute, and seconds; with a time zone displacement value. |
| timestamp (*fractional seconds precision*) with local time zone | ***fractional seconds precision*** must be a number between 0 and 9.  (default is  6) | Includes year, month, day, hour, minute, and seconds; with a time zone expressed as the session time zone. |
| interval year  (*year precision*) to month | ***year precision*** is the number of digits in the  year. (default is 2) | Time period stored in years and months. |

# PRACTICAL NO. 3

**OBJECTIVE:**

To Create, Alter and Drop a Table.

**THEORY:**

1. CREATE: The command is used too create a table. It is used to specify a new relation by giving it a name and specifying attributes and initial constraints. This command is a part of DDL(Data Definition Language) of SQL. The column names must be specified along the data types. Each table must have atleast one column. Tables are the basic structure where data is stored in the database. Tables are divided into rows and columns.

Syntax of CREATE command is:

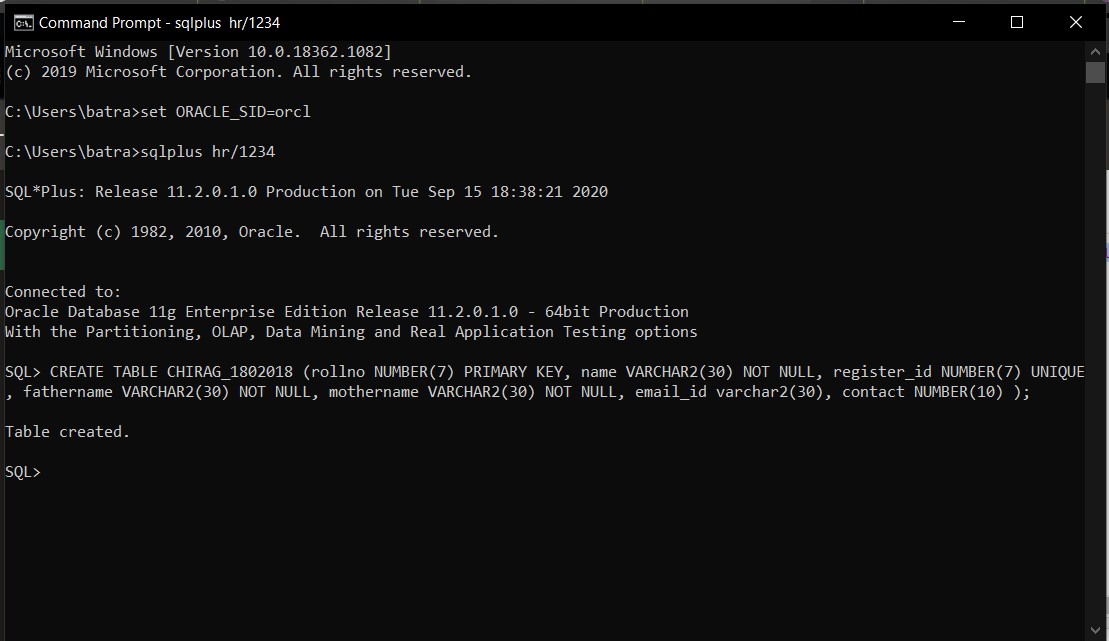
Create Table<Table Name>(

<column Name><datatype>(<size>),………….

)

;

**OUTPUT:**



1. ALTER: The definition of a base table or of other named schema elements can be changed using the ALTER command. After creating a table we may need to change the table structures because we omit a column or our column definition needs to be changed. For this purpose, we use Alter Command.

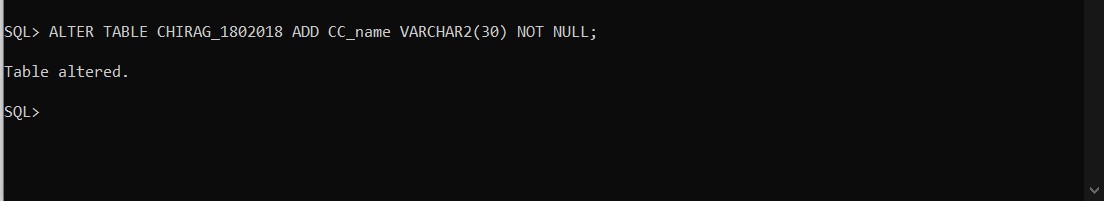
Tables can be altered in one of the three ways:

* 1. By adding a column to existing table.
  2. By changing a column’s definition.
  3. By dropping a column.

Syntax:

Alter Table <Table Name> Add ( <Column Name> <Datatype> (<size>));

**OUTPUT:**



3. DROP: The DROP command is used to drop tables. This command not only deletes all the records in the table, but also removes the definition of a table. When we drop a table, the database loses all the data in the table and all the indexes associated with it.

All data is deleted from the table.

Syntax of DROP Command:

DROP Table <Table Name>;

**OUTPUT:**



# PRACTICAL NO.4

**OBJECTIVE:**

To perform DML Commands on a table.

**THEORY:**

*INSERT Statement*

The INSERT statement is used to add new rows to a table.

Syntax for INSERT statement:

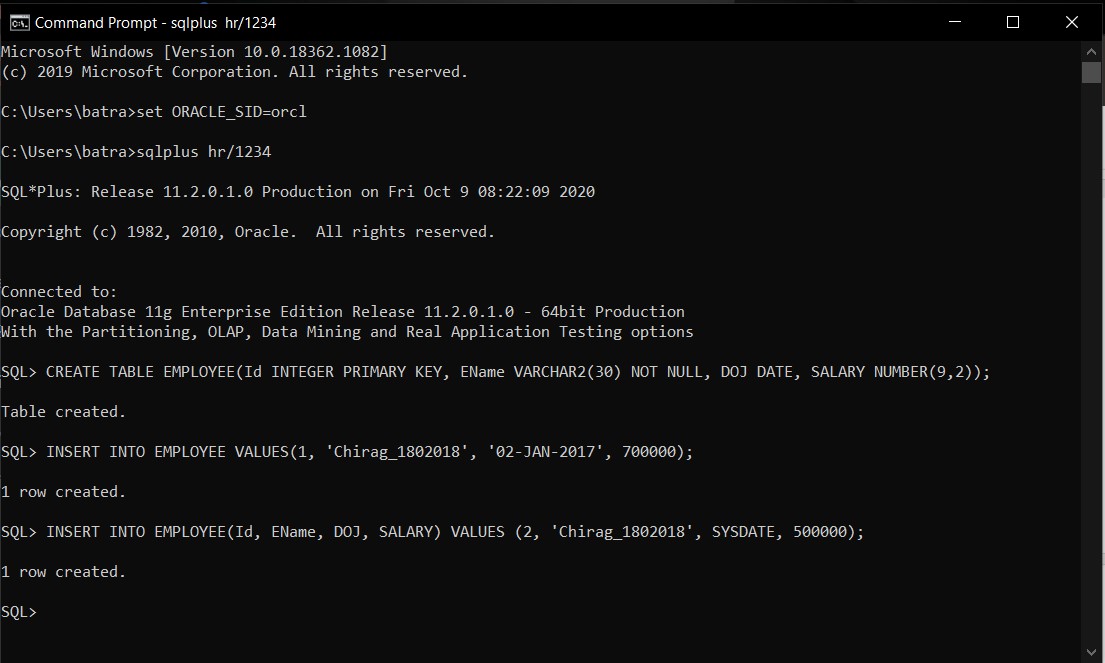
INSERT INTO <table\_name> VALUES (value1, value2, value3, …………..);

OR

INSERT INTO <table\_name>(column\_name, column\_name, column\_name, ………)

VALUES(value, value, value, ………….);

**OUTPUT:**



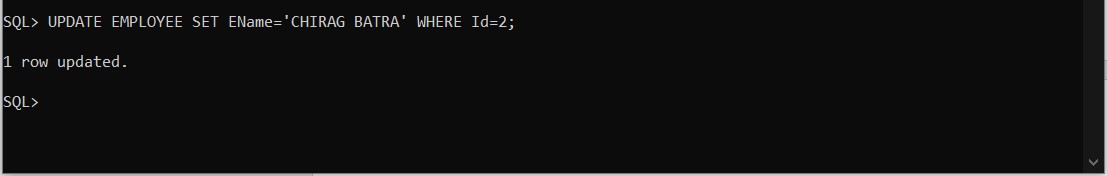
*UPDATE Statement*

The UPDATE Statement is used to modify existing rows.

Syntax for UPDATE Statement:

UPDATE <table\_name> SET column=value, column=value WHERE condition;

**OUTPUT:**



*DELETE Statement*

The DELETE statement is used to delete existing rows from a table.

Syntax for DELETE Statement:

DELETE FROM <table\_name> WHERE condition;

**OUTPUT:**



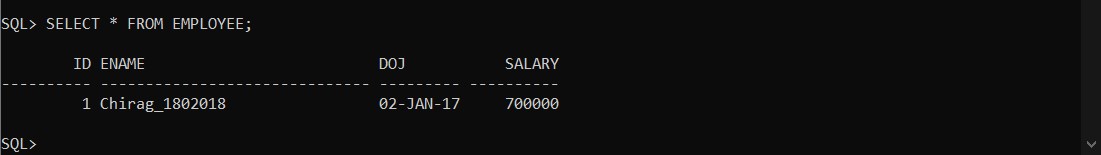
*SELECT Statement*

The SELECT statement is used to retrieve existing rows from a table.

Syntax for SELECT Statement

SELECT column, column, column, ….. FROM <table\_name> WHERE condition;

**OUTPUT:**



# PRACTICAL NO.5

**OBJECTIVE:**

To make use of different clauses viz Where, Group By. Having, Order By.

**THEORY:**

*WHERE Clause*

The WHERE Clause is used to extract only those records that fulfill a specified criterion.

*WHERE Clause includes operators as:*

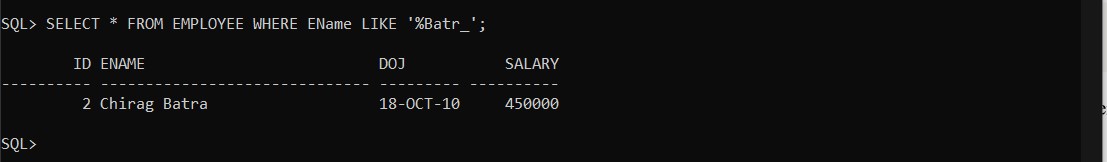
**LIKE Operator:** The LIKE Operator is used to search for a specified pattern in a column.

Syntax for LIKE Operator:

SELECT <column\_name, column\_name> FROM <table\_name>

WHERE <column\_name> LIKE pattern;

**OUTPUT:**



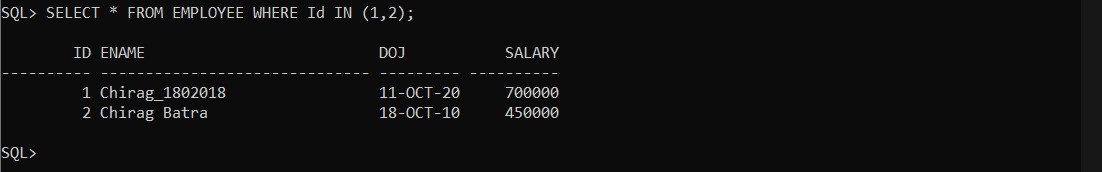
**IN Operator:** The IN operator allows us to specify multiple values in a WHERE clause.

Syntax for IN Operator:

SELECT <column\_name, column\_name> FROM <table\_name>

WHERE <column\_name> IN <value1, value2, ……>;

**OUTPUT:**

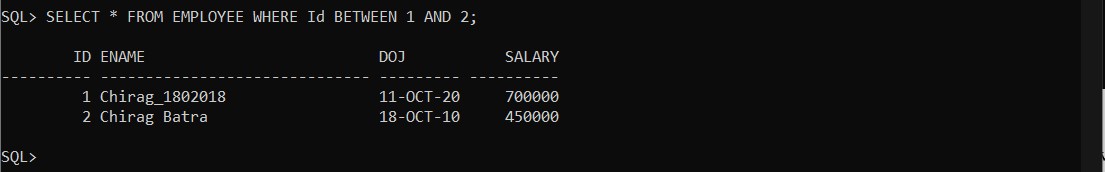


**BETWEEN Operator:** The BETWEEN operator selects values within a range. The values can be numbers, text or dates.

Syntax for BETWEEN Operator:

SELECT <column\_name, column\_name> FROM <table\_name> WHERE <column\_name> BETWEEN <value1> AND <value2>;

**OUTPUT:**



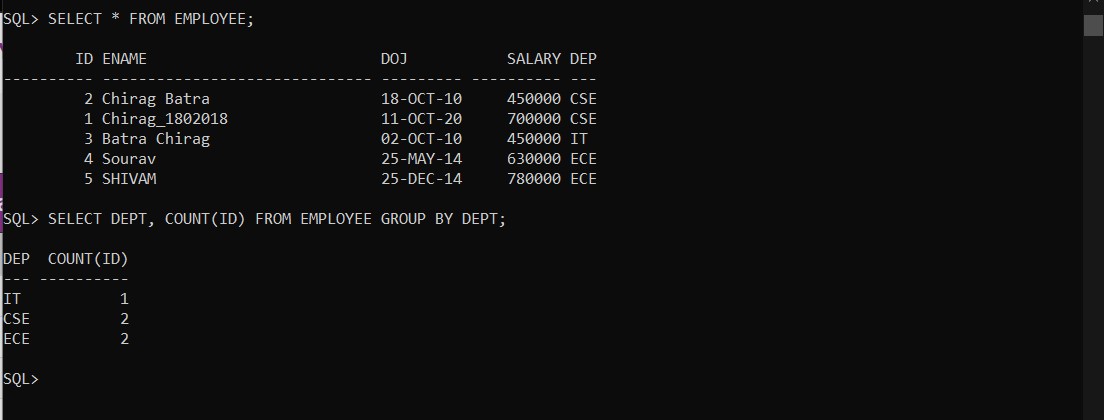
*GROUP BY Clause*

The GROUP BY statement is used in conjuction with the aggregate functions to group the result set by one or more columns.

Syntax for GROUP BY Clause:

|  |  |
| --- | --- |
| SELECT <column\_name, aggregate\_function(column\_name)> from  WHERE <column\_name operator value> GROUP BY <column\_name>; | <table\_name> |

**OUTPUT:**



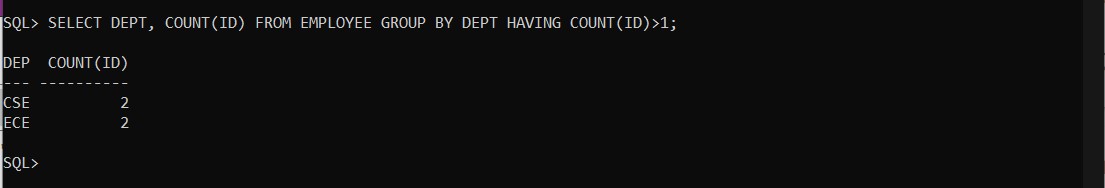
*HAVING Clause*

The HAVING Clause enables us to specify conditions that filter which group results appear in the final results.

Syntax for HAVING Clause:

|  |  |
| --- | --- |
| SELECT <column\_name, aggregate\_function(column\_name)> from  WHERE <column\_name operator value> GROUP BY <column\_name>  HAVING <aggregate\_function(column\_name) operator value>; | <table\_name> |

**OUTPUT:**



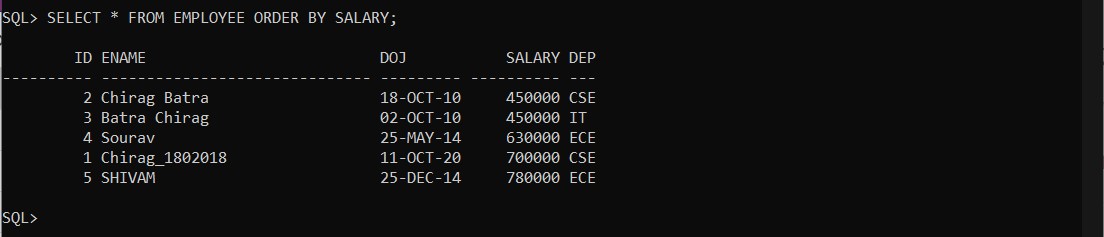
*ORDER BY Clause*

The ORDER BY clause is used to sort the result set by one or more columns.

Syntax for ORDER BY Clause

|  |  |  |
| --- | --- | --- |
| SELECT <column\_name, aggregate\_function(column\_name)>  WHERE <column\_name operator value>  ORDER BY <column\_name, column\_name, …..> ASC|DESC; | from | <table\_name> |

**OUTPUT:**



# PRACTICAL NO.6

**OBJECTIVE:**

To perform aggregate functions like SUM, MAX, MIN, AVG, COUNT on a table.

**THEORY:**

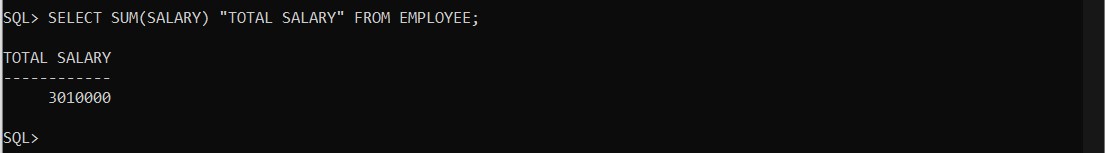
***Aggregate Functions:*** An aggregate function is a function where the values of multiple rows are groped together as input on certain criteria to form a single value of more significant measurement or meaning.

1. **SUM:** The SUM() function returns the total sum of a numeric column.

Syntax:

SELECT SUM(<column\_name>) FROM <table\_name>;

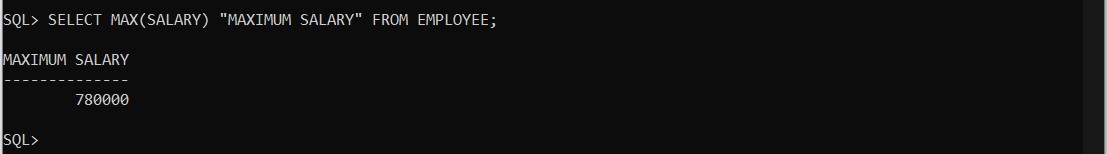
**OUTPUT:**



1. **MAX:** This function returns the maximum of values for any column of a table for all the rows Syntax:

SELECT MAX(<column\_name>) FROM <table\_name>;

**OUTPUT:**

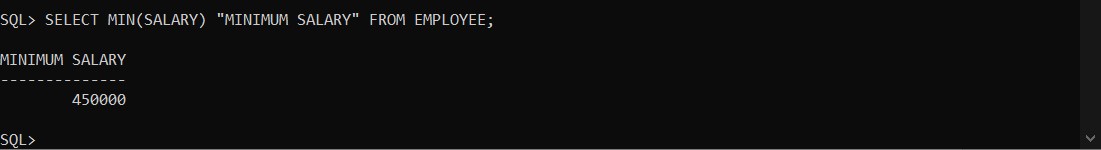


1. **MIN:** This function returns the minimum of values for any column of a table for all the rows.

Syntax:

SELECT MIN(<column\_name>) FROM <table\_name>;

**OUTPUT:**

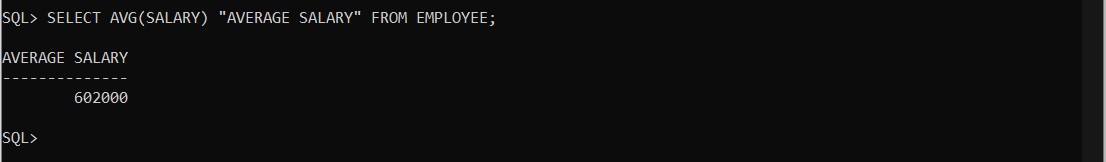


1. **AVG:** This function returns the average of values for any column of a table. It ignores the NULL values in the column.

Syntax:

SELECT AVG(<column\_name>) FROM <table\_name>;

**OUTPUT:**



1. **COUNT:** This function returns the number of rows or non-null values for a given column in a table.

Syntax:

SELECT COUNT(<column\_name>) FROM <table\_name>;

**OUTPUT:**



# PRACTICAL NO.7

**OBJECTIVE:**

To perform arithematic and logical operators.

**THEORY:**

**Arithematic operators:** We can use various Arithmetic Operators on the data stored in the tables. Arithmetic Operators are:

**+** [Addition]

**-** [Subtraction]

**/** [Division]

**\*** [Multiplication]

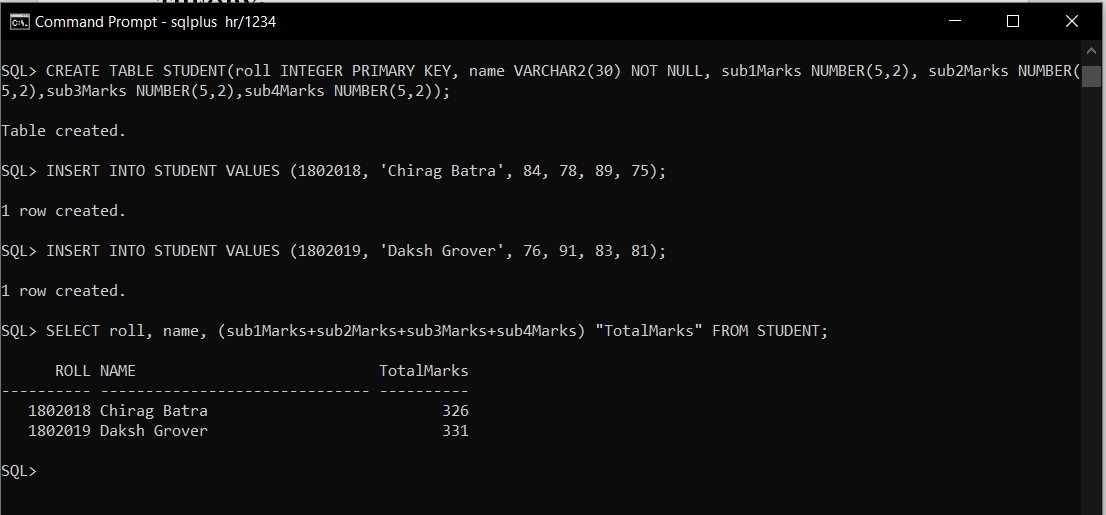
**1.Addition (+) :** It is used to perform **addition operation** on the data items, items include either single column or multiple columns

Syntax:

SELECT column1, column1+ column2,column3+100

AS " column1+ column2" FROM tablename;

**OUTPUT:**



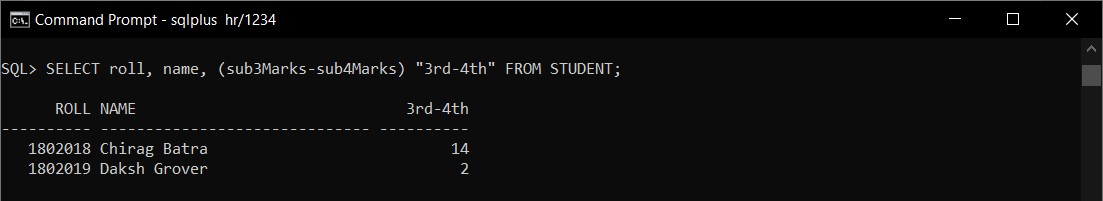
**2.Subtraction (-) :** It is use to perform **subtraction operation** on the data items, items include either single column or multiple colum ns.

Syntax:

SELECT column1, column1- column2,column3-100

AS " column1- column2" FROM tablename;

**OUTPUT:**



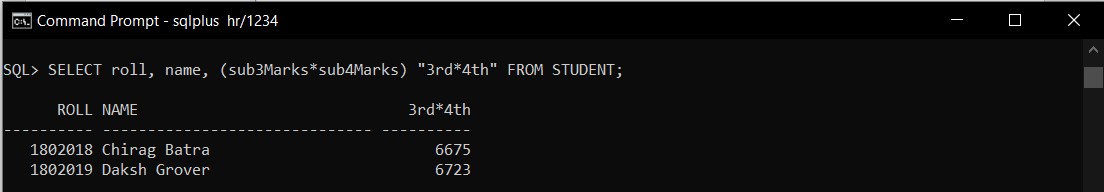
**3.Multiplication (\*) :** It is use to perform **multiplication** of data items.

Syntax:

SELECT column1, column1\* column2,column3\*2

AS " column1\* column2" FROM tablename;

**OUTPUT:**



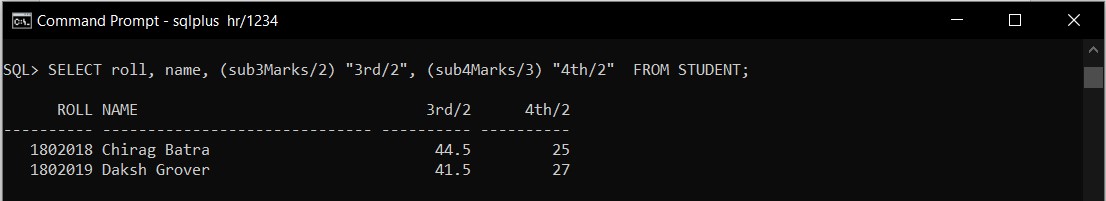
**4.Divide (/) :** It is use to get qoutient when one data is divided by another of data items.

Syntax:

SELECT column1, column1\* column2,(column3\*2)/100

AS " column1\* column2" FROM tablename;

**OUTPUT:**



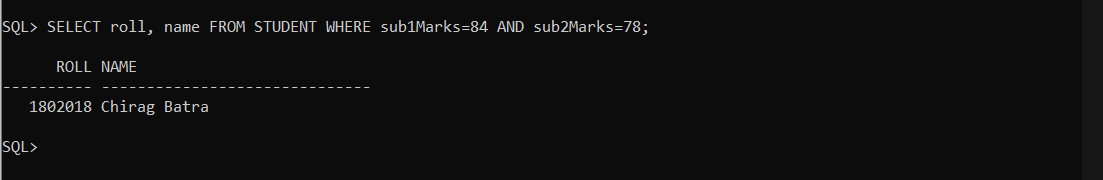
**Logical Operators:** The Logical operators are those that are true or false. They return a true or false values to combine one or more true or false values. The Logical operators are:

**1: AND:** Logical AND compares between two Booleans as expression and returns true when both expressions are true.

Synatx:

|  |
| --- |
| SELECT column1, column1 FROM tablename  where coloumn1…[condition] **and** column2…[condition] |

**OUTPUT:**

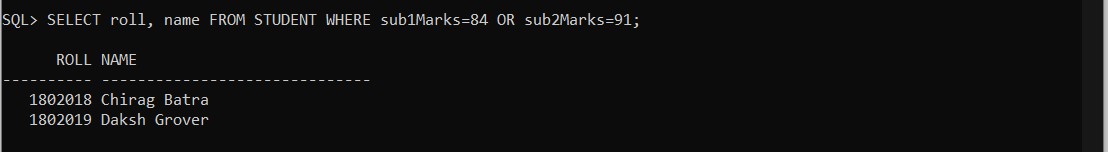


**2. OR:** Logical OR compares between two Booleans as expression and returns true when one of the expression is true.

Syntax:

|  |
| --- |
| SELECT column1, column1 FROM tablename  where coloumn1…[condition] **or** column2…[condition]; |

**OUTPUT:**

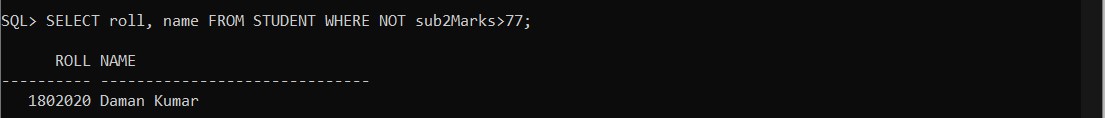


**3.NOT**:Not takes a single Boolean as an argument and changes its value from false to true or from true to false.

Syntax:

SELECT column1, column1 FROM tablename where **Not** coloumn1…[condition];

**OUTPUT:**



# PRACTICAL NO.8

**OBJECTIVE:**

To perform single row functions: String, Numeric, Date and Conversion Functions.

**THEORY:**

There are two types of functions in Oracle:

* **Single Row Functions:** Single row or scalar functions return a value for every row that is processed in a query.
* **Group Functions:** These functions group the rows of data based on the values returned by the query. This is discussed in SQL GROUP functions. The group functions are used to calculate aggregate values like total or average, which return just one total or one average value after processing a group of rows.

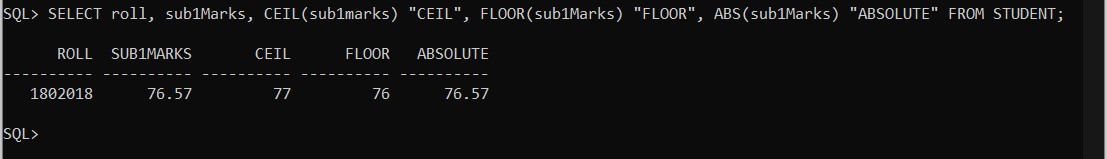
There are four type of single row functions:

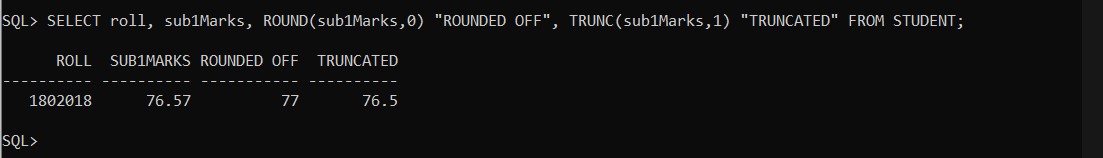
1. **Numeric Functions:** Numeric functions are used to perform operations on numbers.

They accept numeric values as input and return numeric values as output.

|  |  |
| --- | --- |
| **Function Name** | **Return Value** |
| ABS(x) | Absolute value of the number ‘x’. |
| CEIL(x) | Integer value that is greater than or equal to the number ‘x’. |
| FLOOR(x) | Integer value that is less than or equal to the number ‘x’. |
| TRUNC(x, y) | Truncates value of number ‘x’ up to ‘y’ decimal places. |
| ROUND(x, y) | Rounded off value of the number ‘x’ up to the number ‘y’ decimal places. |

**OUTPUT:**

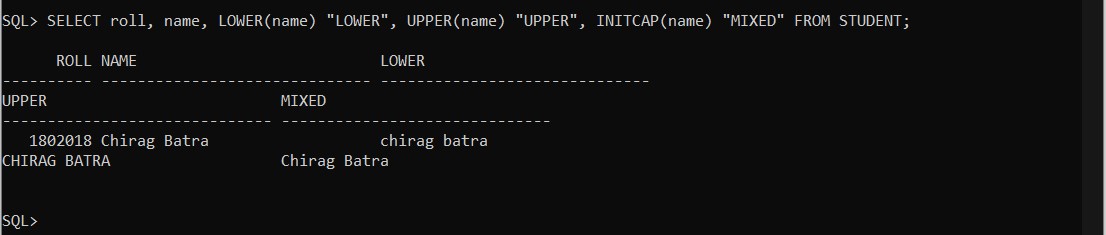


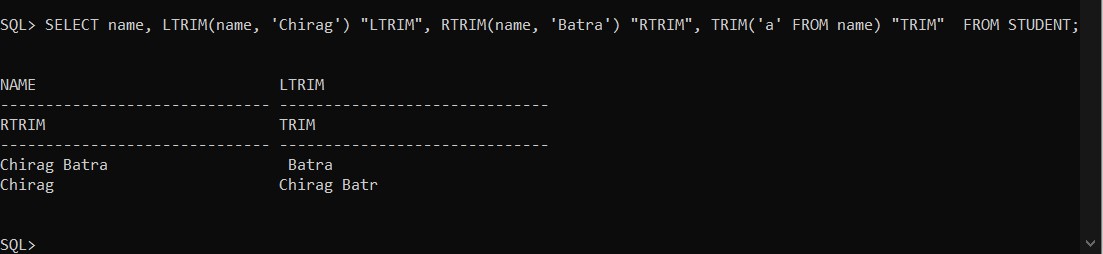


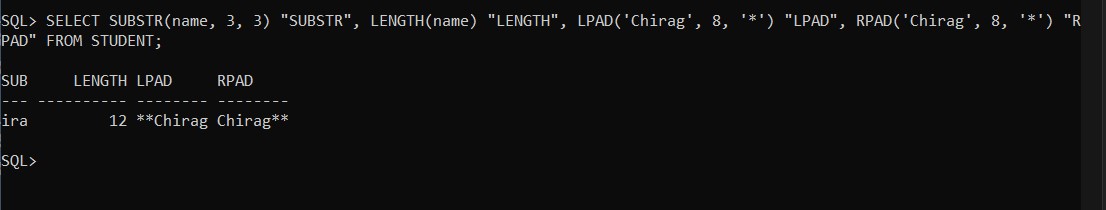
1. **Character or Text Functions:** Character or text functions are used to manipulate text strings. They accept strings or characters as input and can return both character or number values as output.

|  |  |
| --- | --- |
| **Function Name** | **Return Value** |
| LOWER(string\_value) | All the letters in ‘string\_value’ is converted to lower case. |
| UPPER(string\_value) | All the letters in ‘string\_value’ is converted to upper case. |
| INITCAP(string\_value) | All the letters in ‘string\_value’ is converted to mixed case. |
| LTRIM(string\_value, trim\_text) | All occurrences of ‘trim\_text’ is removed from the left of ‘string\_value’. |
| RTRIM(string\_value, Trim\_text) | All occurrences of ‘trim\_text’ is removed from the right of ‘string\_value’. |
| TRIM(trim\_text FROM string\_value) | All occurrences of ‘trim\_text’ from the left and right of ‘string\_value’, ‘trim\_text’ can also be only one character long |
| SUBSTR(string\_value, m, n) | Returns ‘n’ number of characters from ‘string\_value’ starting from the ‘m’ position. |
| LENGTH(string\_value) | Number of characters in ‘string\_value’ in returned. |
| LPAD(string\_value, n, pad\_value) | Returns ‘string\_value’ left padded with ‘pad\_value’. The length of the whole string will be of ‘n’ characters. |
| RPAD(string\_value, n, pad\_value) | Returns ‘string\_value’ right padded with ‘pad\_value’. The length of the whole string will be of ‘n’ characters. |

**OUTPUT:**



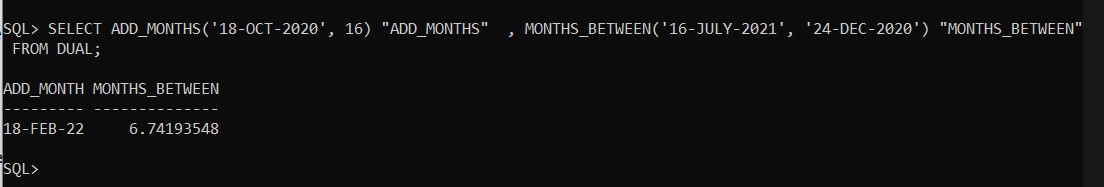


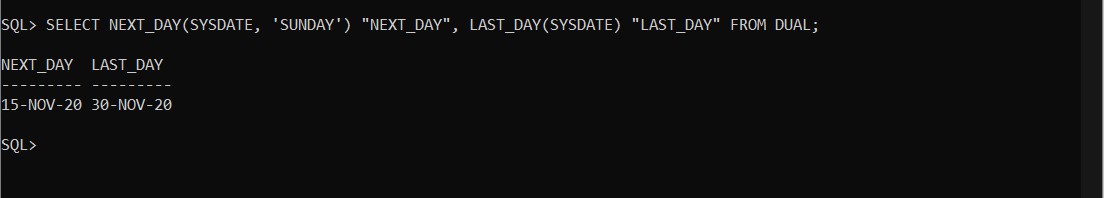


1. **Date Functions:** These are the functions that take values that are of datatype DATE as input and return values of datatypes DATE, except for the MONTHS\_BETWEEN function, which return a number as output.

|  |  |
| --- | --- |
| **Function Name** | **Return Value** |
| ADD\_MONTHS(date  , n) | Returns a date value after adding ‘n’ months to the date ‘x’. |
| MONTHS\_BETWEE  N  (x1, x2) | Returns the number of months between dates x1 and x2. |
| ROUND(x,  date\_format) | Returns the date ‘x’ rounded off to the nearest century, year, month, date,hour, minute, or second as specified by the ‘date\_format’. |
| TRUNC(x,  date\_format) | Returns the date ‘x’ lesser than or equal to the nearest century, year, month, date,hour, minute, or second as specified by the ‘date\_format’. |
| NEXT\_DAY(x, week\_day) | Returns the next date of the ‘week\_day’ on or after the date ‘x’ occurs. |
| LAST\_DAY(x) | It is used to determine the number of days remaining in a month from the date ‘x’ specified. |
| SYSDATE | Returns the systems current date and time. |
| NEW\_TIME(x,  zone1, zone2) | Returns the date and time in zone2 if date ‘x’ represents the time in zone1. |

**OUTPUT:**

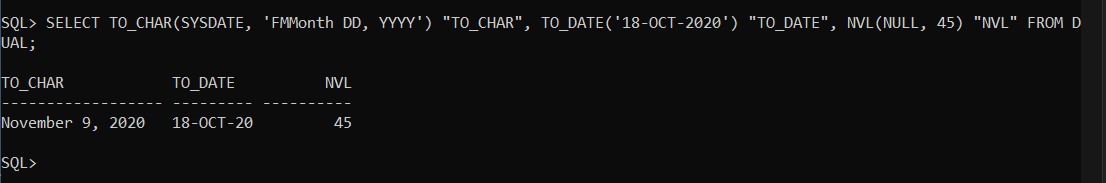




1. **Conversion functions:** These are functions that help us to convert a value in one form to another form. For ex: a null value into an actual value, or a value from one datatype to another datatype like NVL, TO\_CHAR, TO\_NUMBER, TO\_DATE.

|  |  |
| --- | --- |
| **Function Name** | **Return Value** |
| TO\_CHAR(x, y) | Converts numeric and data values to character string value. It cannot be used for calculations since it is a string value. |
| TO\_DATE(x, date\_format) | Converts a valid numeric and character values to a date value. Date is formatted to the format specified by ‘date\_format’. |
| NVL(x, y) | If ‘x’ is NULL, replace it with ‘y’. ‘x’ and ‘y’ must be of the same datatype. |

**OUTPUT:**



# PRACTICAL NO.9

**OBJECTIVE:**

To study set operators.

**THEORY:**

1. **Union:** The UNION operator is used to combine the result set of two or more SELECT statements. Each SELECT statement within the UNION must have the same number of columns, The columns must also have similar data types.

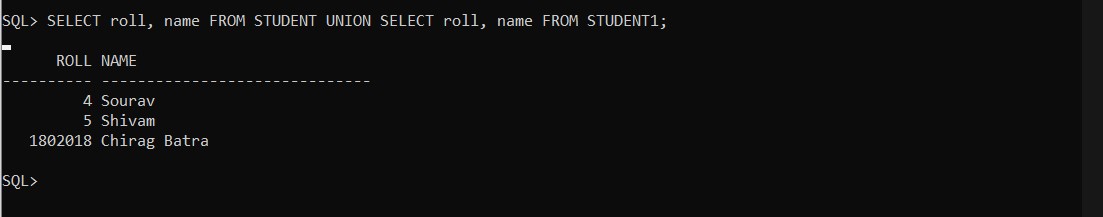
Syntax for Union:

SELECT column\_name(s) FROM table1

UNION

SELECT column\_name(s) FROM table2;

**OUTPUT:**



1. **Intersection:** The SQL INTERSECT clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identified to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.

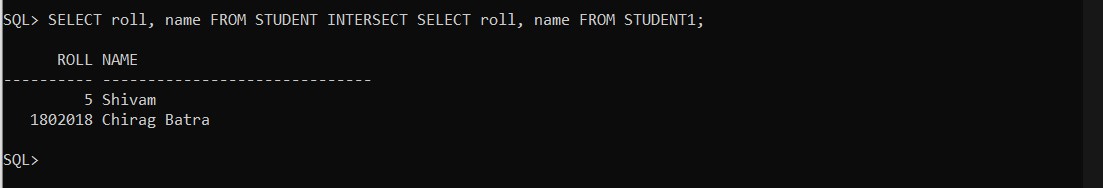
Syntax for Intersect:

SELECT column\_name(s) FROM table1

INTERSECT

SELECT column\_name(s) FROM table2;

**OUTPUT:**



1. **Set Difference:** The SQL MINUS query returns all rows in the first SQL SELECT statement that are not returned in the second SQL SELECT statement.

Each SQL SELECT statement within the SQL MINUS query must have the same number of fields in the result sets with similar data types.

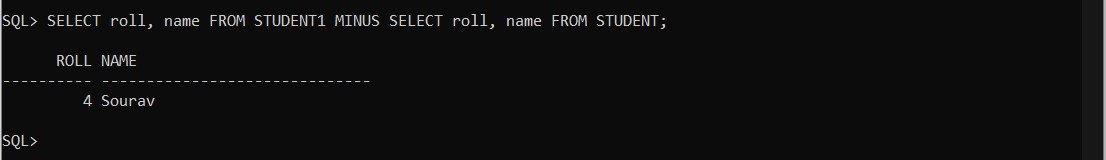
Syntax for MINUS:

SELECT column\_name(s) FROM table1

INTERSECT

SELECT column\_name(s) FROM table2;

**OUTPUT:**



# PRACTICAL NO.10

**OBJECTIVE:**

To study and implement constraints on a table.

**THEORY:**

The following constraint types are valid:

* NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK

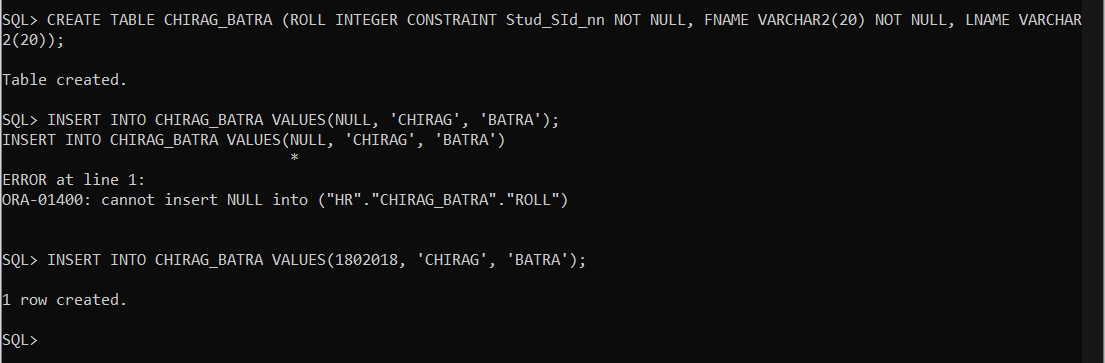
1. **NOT NULL:** NOT NULL constraint prevents a column from accepting NULL values. NOT NULL can only be applied as a column level constraint. Constraint name is optional and it can be specified by using CONSTRAINT keyword.

Syntax for NOT NULL:

CREATE TABLE table\_name(

column\_name datatype CONSTRAINT constraint\_name NOT NULL, column\_name datatype NOT NULL);

**OUTPUT:**



1. **UNIQUE:** UNIQUE constraint on a column ensures that two rows in a table cannot have some value in that column. Unlike primary key, UNIQUE constraint allows NULL values. A table can have many UNIQUE constraints.

Syntax for Unique:

CREATE TABLE table\_name(

column\_name datatype CONSTRAINT constraint\_name UNIQUE);

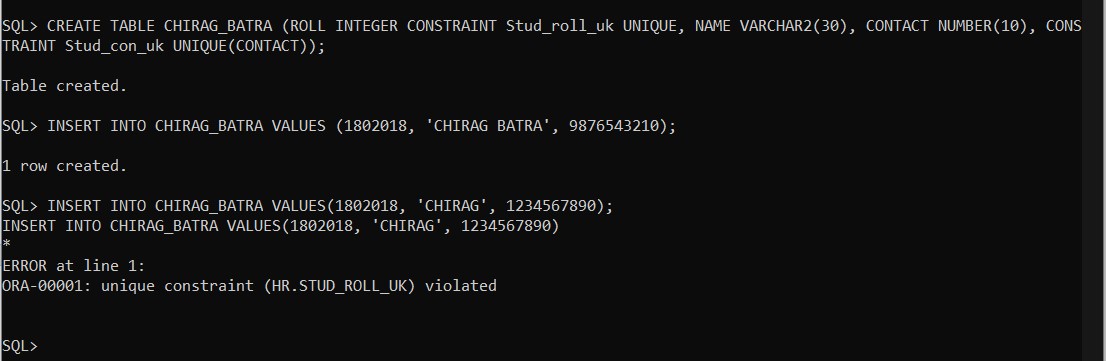
OR

CREATE TABLE table\_name(

Column\_name datatype ,

CONSTRAINT constraint\_name UNIQUE (column\_name));

**OUTPUT:**



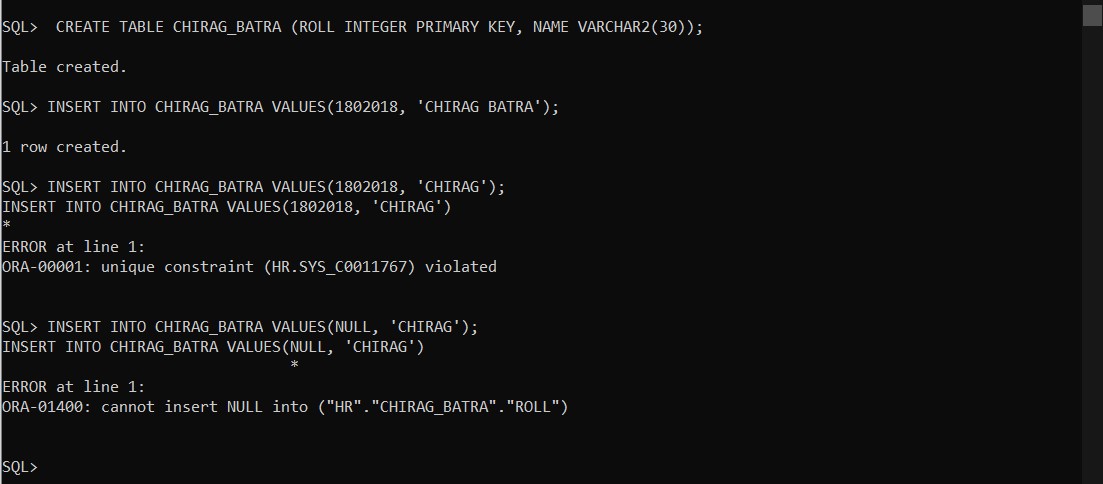
1. **PRIMARY KEY:** PRIMARY KEY constraint on a column ensures that the column cannot contain NULL and duplicate values. We can have only one PRIMARY KEY in a table.

Syntax for PRIMARY KEY:

CREATE TABLE table\_name(

Column\_name datatype CONSTRAINT constraint\_name PRIMARY KEY);

**OUTPUT:**



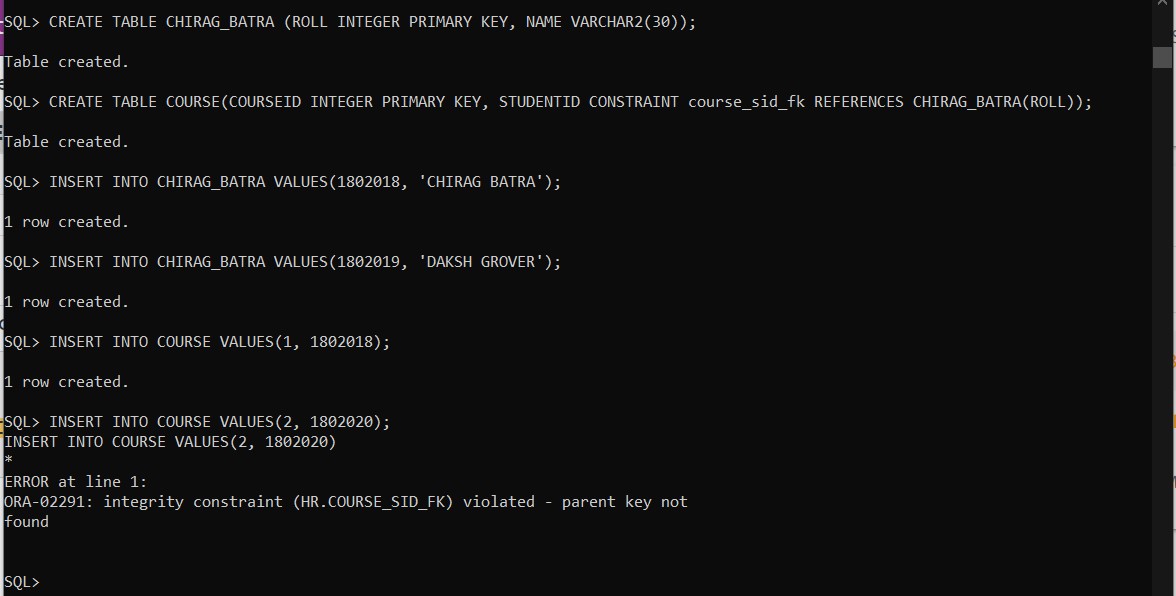
1. **FOREIGN KEY:** FOREIGN KEY is used to link two tables together. It is a field(or collection of fields) in one table that refers to PRIMARY KEY in another table. The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

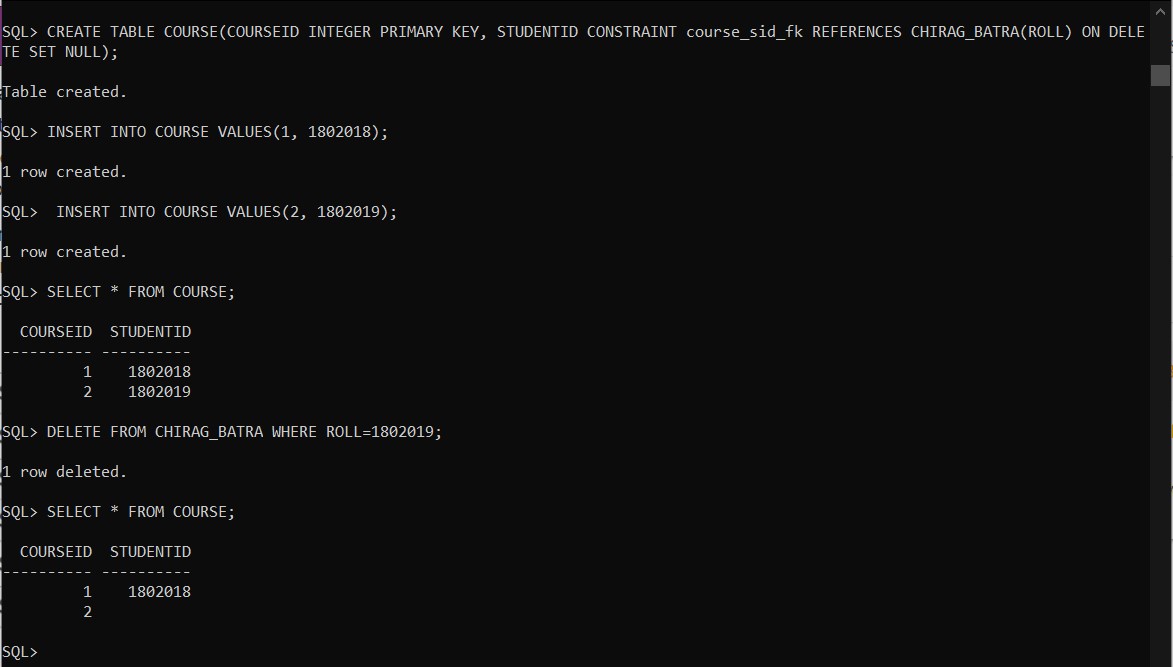
Syntax for FOREIGN KEY:

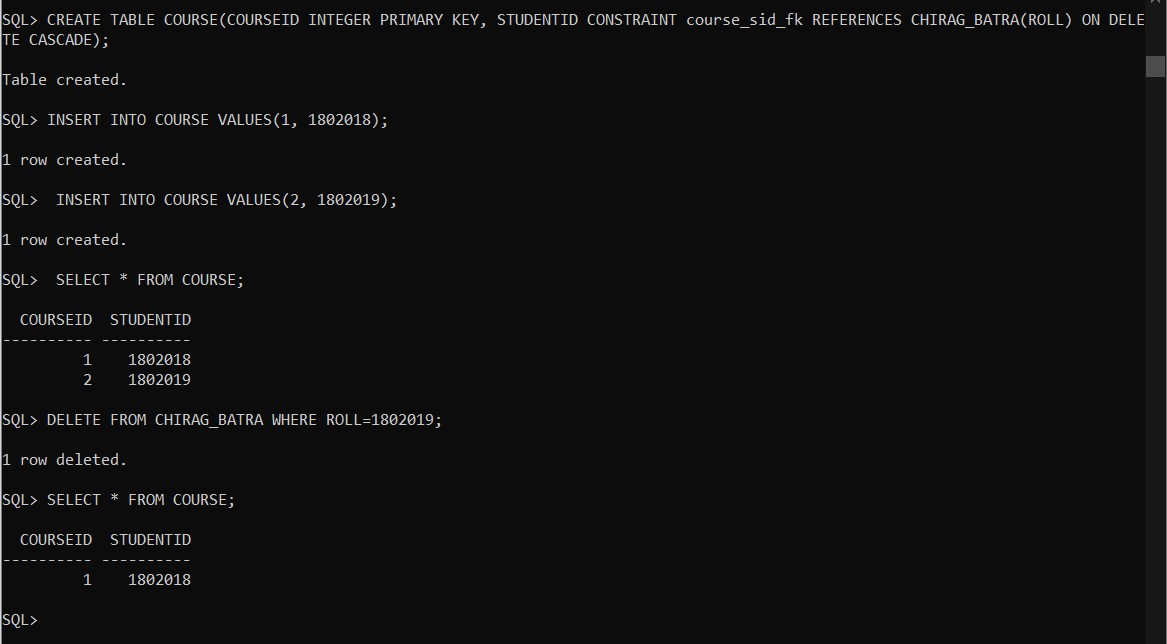
CREATE TABLE table\_name(

column\_name datatype CONSTRAINT constraint\_name REFERENCES table\_name1(column\_name1));

**OUTPUT:**







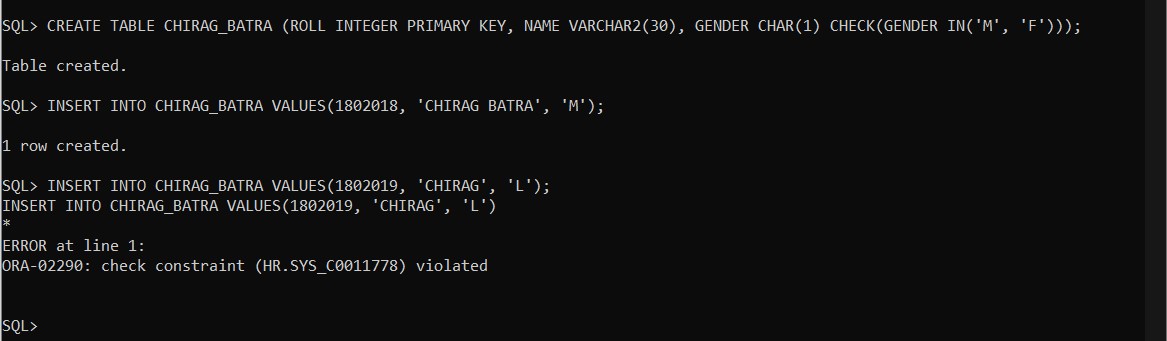
1. **CHECK:** CHECK constraint is used to limit the values that can be specified for a column.

Syntax for CHECK:

CREATE TABLE table\_name(

Column\_name datatype CONSTRAINT constraint\_name CHECK( column\_name IN (values)));

**OUTPUT:**



# PRACTICAL NO.11

**OBJECTIVE:**

To understand the use and working of joins.

**THEORY:**

SQL joins are used to relate information in different tables. Joins are used to combine columns from different tables. The connection between tables is established through the where clause. There are five types of Joins:

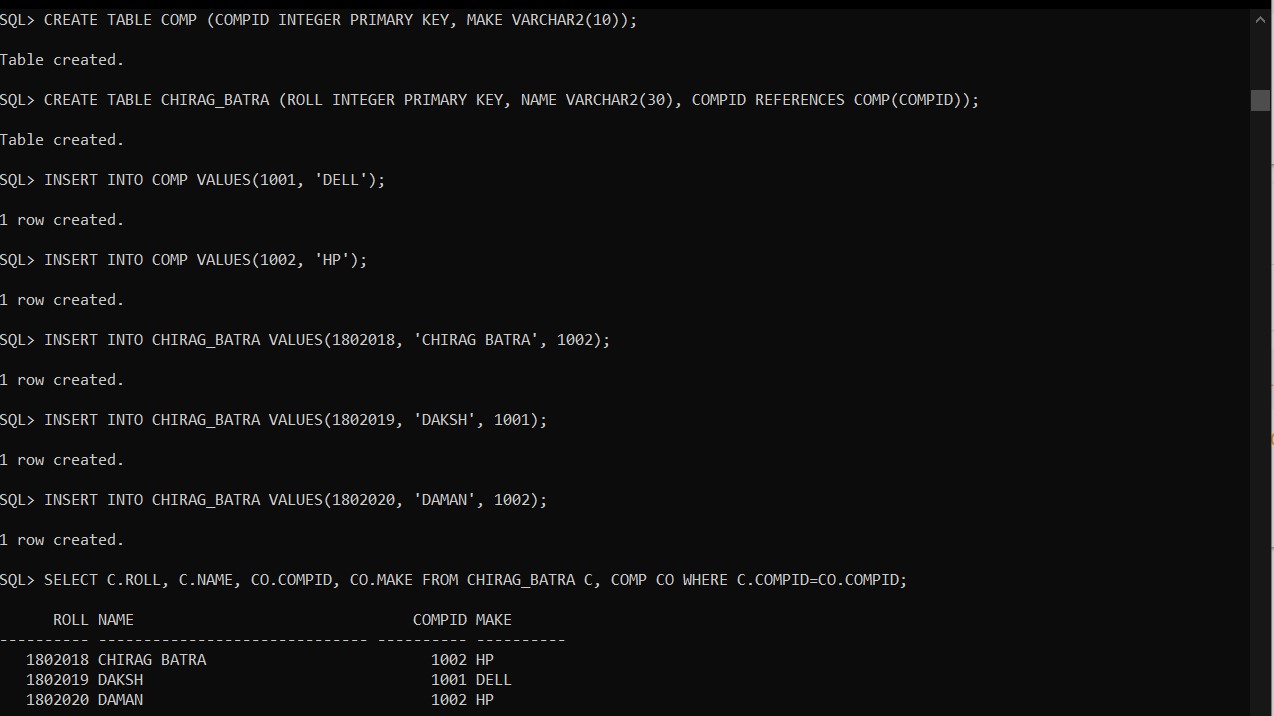
* Equijoins
* Cartesian Joins
* Outer Joins
* Non-Equi Joins
* Self Joins

1. **Equijoins:** When two tables are joined together using equality of values in one or more columns, they make an Equijoin. Table prefixes are utilized to prevent ambiguity and the where clause specifies the columns being joined. Equijoins are also called Simple joins or Inner joins.

Syntax of Equijoin:

SELECT column\_name(s) FROM table t1, table t2 WHERE condition;

**OUTPUT:**

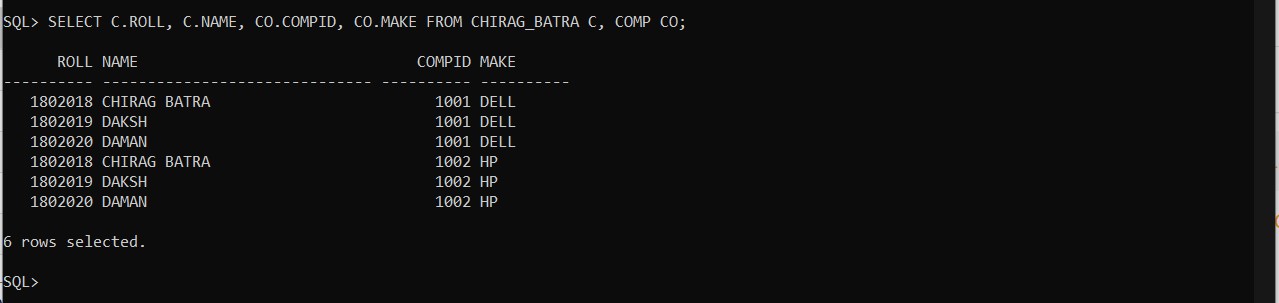


1. **Cartesian Join:** The cartesian product returns the number of rows equal to the product of all rows in all the tables joined. Cartesian product is useful in finding out all the possible combination of columns from different tables.

Syntax for Cartesian Join:

SELECT column\_name(s) FROM table t1, table t2;

**OUTPUT:**

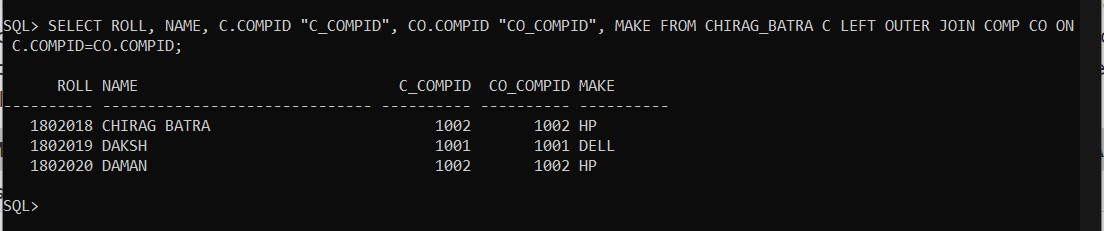


1. **Outer Join:** The SQL join condition returns all rows from both tables which satisfy the join condition along with the rows which do not satisfy the join condition from one of the tables. The outer join is used in such cases where any values in one table that do not have corresponding value in another table. We have further three types of outer join:
   * Left Outer Join
   * Right Outer Join
   * Full Outer Join

Syntax for Left Outer Join:

SELECT column\_name(s) FROM table1 LEFT OUTER JOIN table2 ON condition;

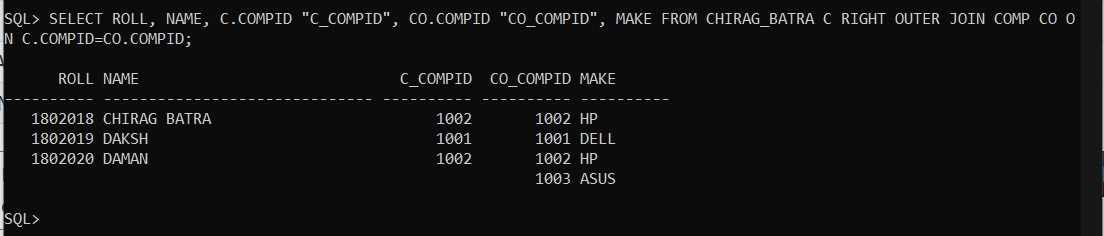
**OUTPUT:**



Syntax for Right Outer Join:

SELECT column\_name(s) FROM table1 RIGHT OUTER JOIN table2 ON condition;

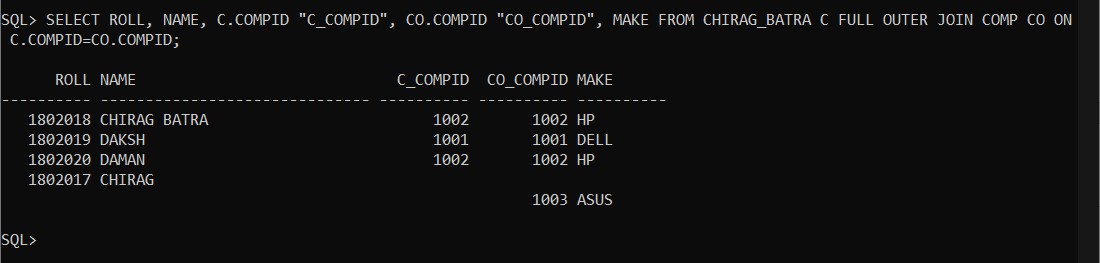
**OUTPUT:**



Syntax for Full Outer Join:

SELECT column\_name(s) FROM table1 FULL OUTER JOIN table2 ON condition;

**OUTPUT:**

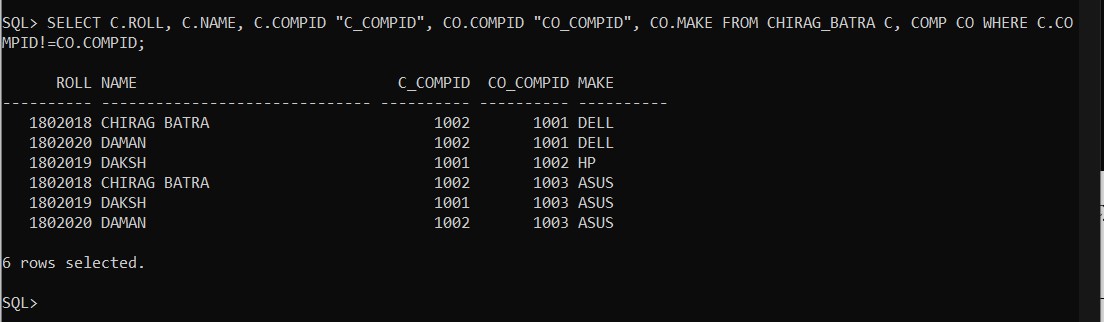


1. **Non-Equi Join:** When two or more tables are joined together using non equality of values in one or more columns, they make a Non-Equi Join.

Syntax for Non-Equi Join:

SELECT column\_name(s) FROM table t1, table t2 WHERE condition;

**OUTPUT:**

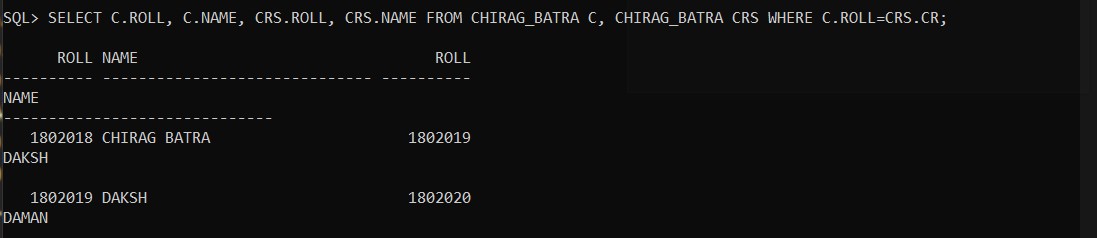


1. **Self Join:** Self Join is to join a table to itself means that each row of the table is combined with itself and with every other row of the table. This type of join is used when a table has a foreign key that references its own primary key. The self join can be viewed as join of two copies of the same table.

Syntax for Self Join

SELECT column\_name(s) FROM table 1 WHERE condition;

**OUTPUT:**



# PRACTICAL NO.12

**OBJECTIVE:**

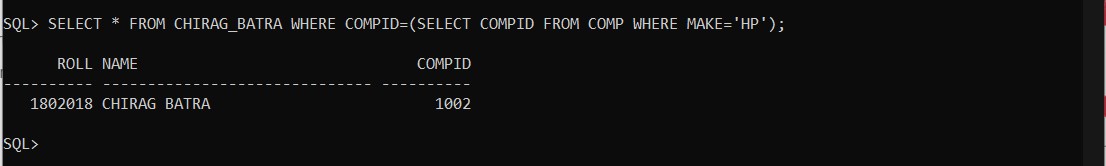
To understand the use and working of Sub-Queries.

**THEORY:**

A Subquery is a query within another SQL query and embedded within the WHERE clause. Syntax for Sub Query:

SELECT column\_name(s) FROM table\_name where column\_name operator (SELECT column\_name FROM table\_name);

**OUTPUT:**



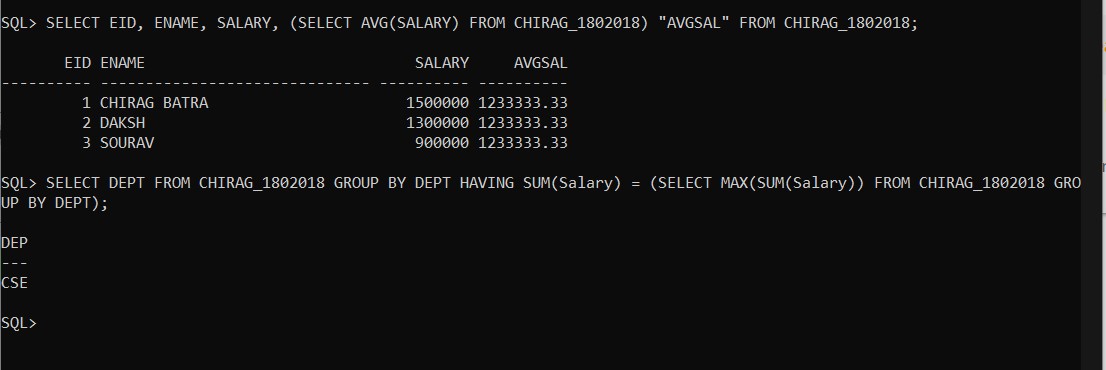
There are two types of subqueries:

* Single row sub query
* Multiple row sub query

1. **Single Row Sub Query:** A single row subquery returns zero or one row to the outer SQL statement. You can place a subquery in a WHERE clause, a HAVING clause, or a FROM clause of a SELECT statement. It use single row comparison operators:

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| = | Equal to |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |
| != or <> | Not equal to |

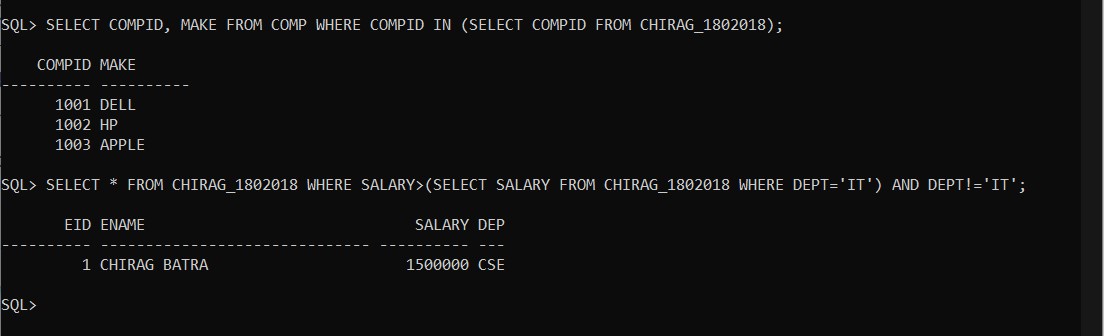
**OUTPUT:**



1. **Multiple row Sub Query:** Multiple-row subqueries are nested queries that can return more than one row of results to the parent query. Multiple-row subqueries are used most commonly in WHERE and HAVING clauses. It uses multiple row comparison operators:

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| IN | Equal to any member in the list. |
| ANY | Compare value to each value returned by the subquery. |
| ALL | Compare value to every value returned by the subquery. |

**OUTPUT:**



# PRACTICAL NO.13

**OBJECTIVE:**

To perform data security and privileges command: Grant and Revoke, Rollback and Commit on a table.

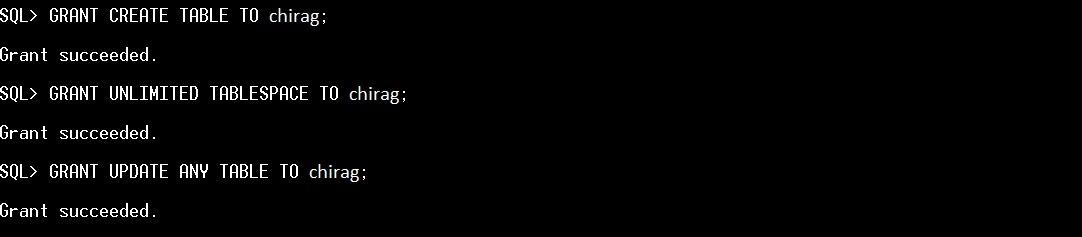
**THEORY:**

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

Syntax for GRANT:

GRANT SELECT privilege\_list ON table\_name TO SYSTEM;

**OUTPUT:**

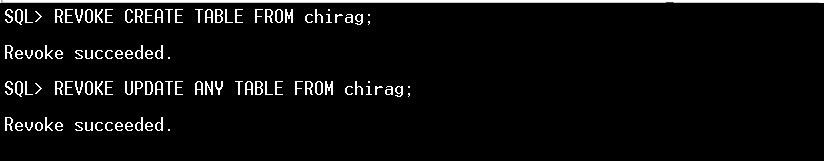


1. **REVOKE:** The REVOKE command removes user access rights or privileges to the database objects.

Syntax for ROLLBACK:

REVOKE SELECT privilege\_list ON table\_name FROM SYSTEM;

**OUTPUT:**

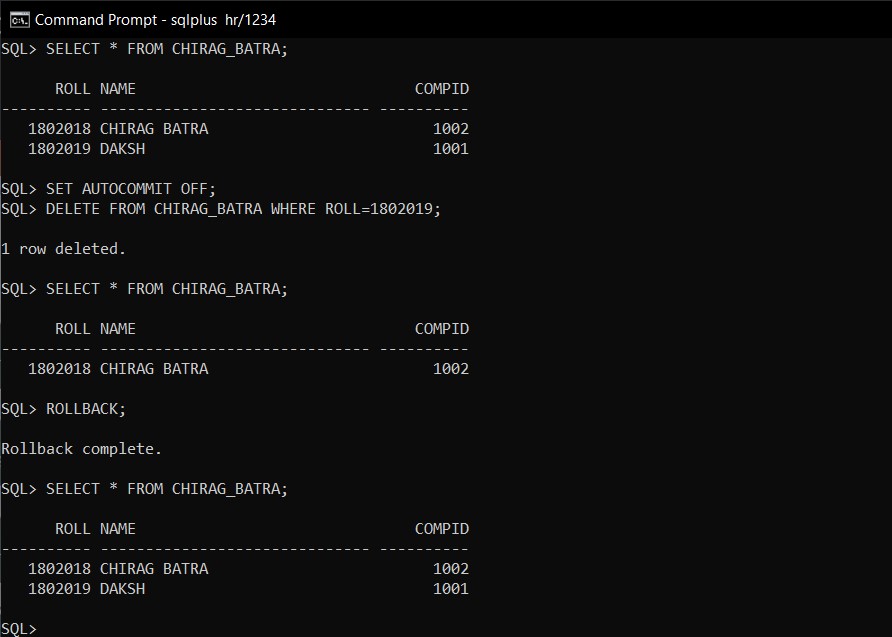


1. **ROLLBACK:** The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued**.**

Syntax for ROLLBACK:

Rollback;

**OUTPUT:**

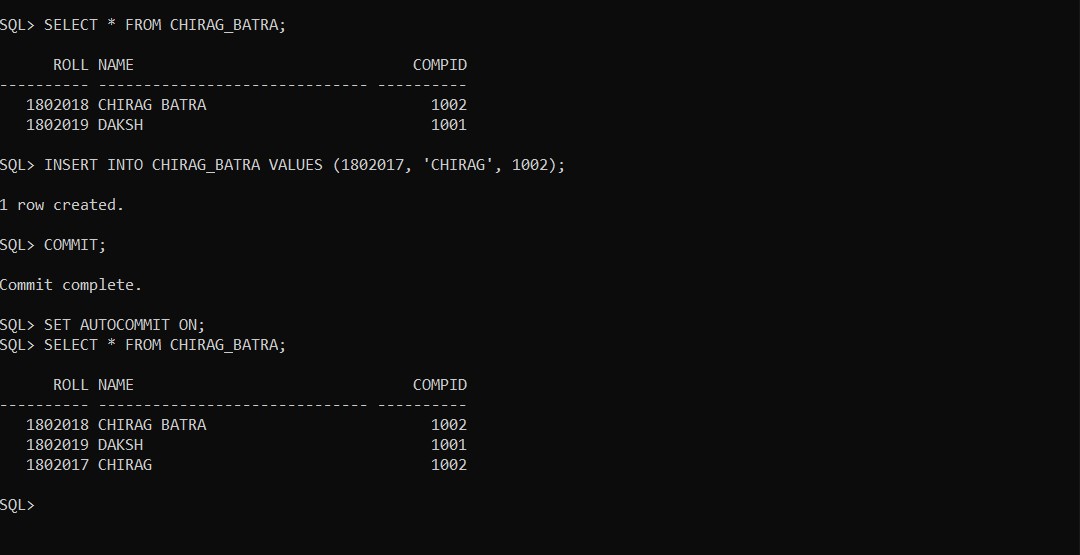


1. **COMMIT:** The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

Syntax for COMMIT:

Commit;

**OUTPUT:**



# PRACTICAL NO.14

**OBJECTIVE:**

To make Views of a table and to perform Insert, Update, Select and Delete operations on View.

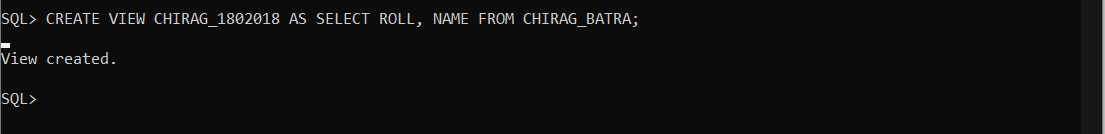
**THEORY:**

**VIEWS:** To reduce redundant data to the minimum possible, Oracle allows the creation of an object called a view. A View is mapped, to a Select sentence. This technique offers a simple, effective way of hiding columns of a table.

Syntax for Views:

CREATE VIEW view\_name AS SELECT column\_list FROM table\_name;

**OUTPUT:**

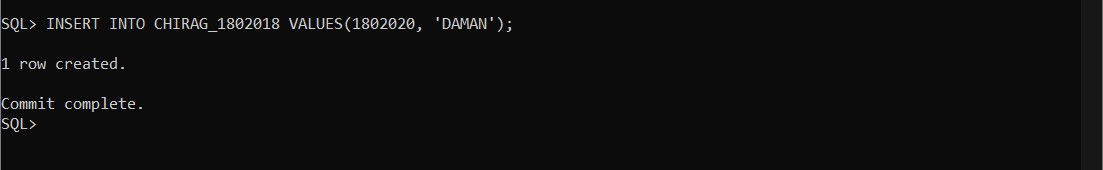


**Insertion in Views:** It is used to Insert rows in views. We must specify a list of values for a particular row.

Syntax for insertion:

INSERT INTO view\_name VALUES list\_of\_values;

**OUTPUT:**

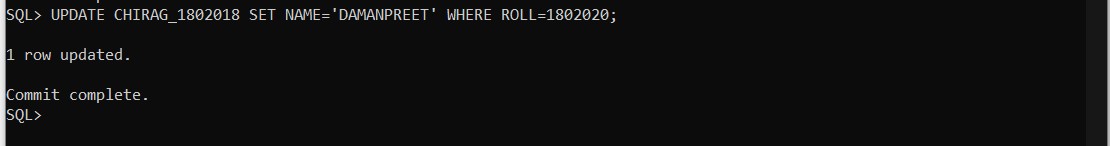


**Updation in Views:** Columns in a view can be updated. It is used to modify attribute values of one or more selected rows. Values of a single column or group of columns can be updated.

Syntax for Updation:

UPDATE view\_name SET column\_name=value WHERE condition;

**OUTPUT:**



**Selection in Views:** SELECT is used to retrieve information from the view.

Syntax for Selection:

SELECT column\_name(s) FROM view\_name;

**OUTPUT:**



**Deletion in Views:** It is used to remove existing rows from a view. The entire row is deleted from the view. A set of rows can also be deleted from the table by specifying the condition.

Syntax for Deletion:

DELETE FROM view\_name WHERE condition;

**OUTPUT:**



# PRACTICAL NO.15

**OBJECTIVE:**

To make indexes and sequences for a table.

**THEORY:**

**Indexes:** An Index is a database object that is used by the server to find a row in a table quickly. Indexing a table is an access strategy, that is , a way to sort and search records in the table. Indexes are essential to improve the speed with which the records can be located and retrieved from a table. Indexes are of two types:

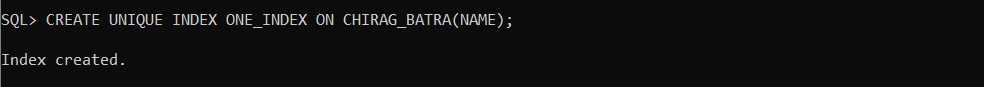
* Simple Index
* Composite Index

1. **Simple Index:** An index created on a single column of a table is called a simple index.

Syntax for Simple Index:

CREATE UNIQUE INDEX index\_name ON table\_name(column\_name);

**OUTPUT:**

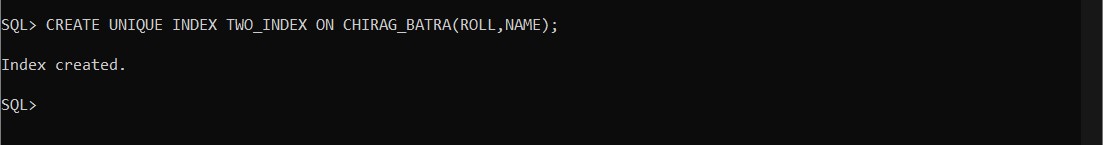


1. Composite Index: An Index created on more than one columns is called a composite index.

Syntax for Composite Index:

CREATE UNIQUE INDEX index\_name ON table\_name(column\_name(s));

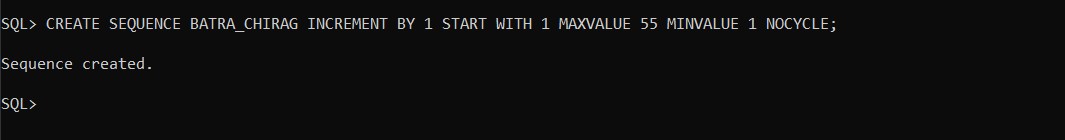
**OUTPUT:**



**Sequences:** Sequence is a database object from which multiple users may generate unique integers. We can use sequences to automatically generate primary key values. Syntax for sequence:

CREATE SEQUENCE sequence\_name INCREMENTED BY 1 condition(s);

**OUTPUT:**



# PRACTICAL NO.16

**OBJECTIVE:**

Introduction to PL/SQL Structure.

**THEORY:**

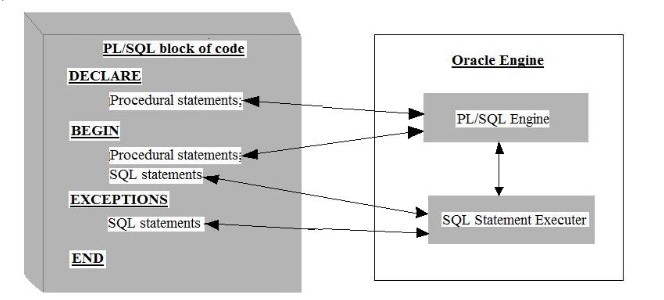
**PL/SQL** stands for Procedural Language Extension of SQL.

PL/SQL is a combination of SQL along with the procedural features of programming languages.

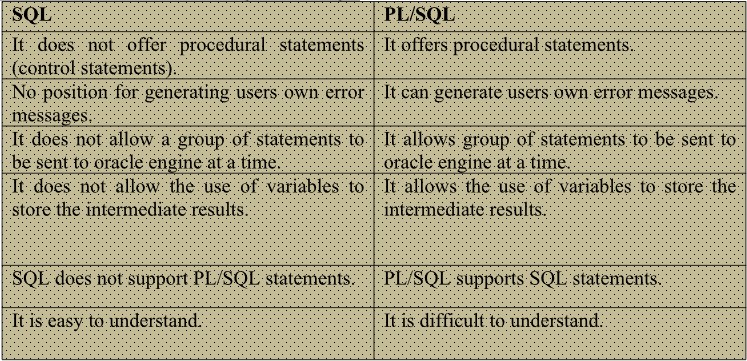
It was developed by oracle Corporation in the early 90’s to enhance the capabilities of SQL.

**Execution of PL/SQL Block:**

The PL/SQL statements written on the client side are passed to the PL/SQL engine at server side and all the SQL statements are send to the SQL executer. After the execution of whole block the result is send back to the client side. The execution of the whole block is done in one go.



**Difference between SQL & PL/SQL:**



# PRACTICAL NO.17

**OBJECTIVE:**

Compute Addition, Subtraction, Product and Division of given two numbers.

**THEORY:**

SET SERVEROUTPUT ON;

declare a number; b number; r1 number; r2 number; r3 number; r4 number; begin a := &amp; b := &amp1; r1:=a+b; r2:=a-b; r3:=a\*b; r4:=a/b; dbms\_output.put\_line('Sum is: ' ||r1); dbms\_output.put\_line('Difference is: ' ||r2); dbms\_output.put\_line('Product is: ' ||r3); dbms\_output.put\_line('Division is: ' ||r4); end;

**OUTPUT:**



# PRACTICAL NO.18

**OBJECTIVE:**

To Print Multiplication Table of a number.

**THEORY:**

SQL>

Declare table\_of number := &amp; cnt number := 1;

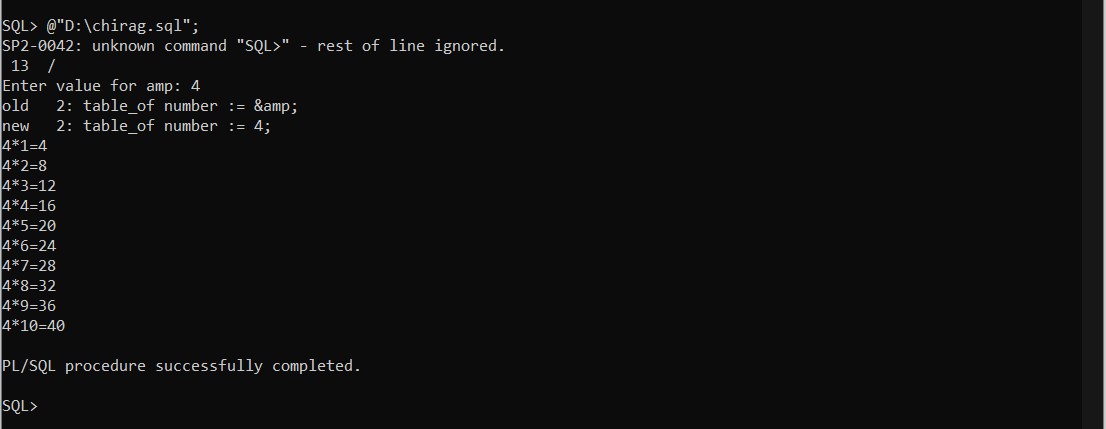
Result number; Begin

while (cnt != 11) loop

Result :=table\_of \* cnt;

DBMS\_output.put\_line(table\_of|| '\*' || cnt || '=' || Result); cnt := cnt + 1; end loop; end;

**OUTPUT;**



# PRACTICAL NO.19

**OBJECTIVE:**

For Loop – To Find Reverse of a given number.

**THEORY:**

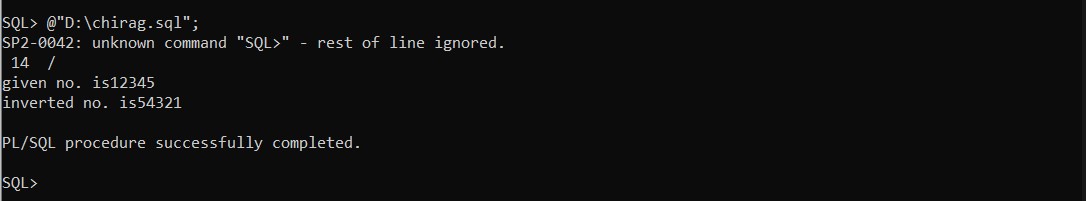
SQL>

Declare gvn\_no varchar(5) := '12345'; str\_length number(2); inverted\_no varchar(5); Begin str\_length := length(gvn\_no); for cntr IN Reverse 1..str\_length loop inverted\_no := inverted\_no || substr(gvn\_no,cntr,1);

End loop;

DBMS\_output.put\_line('given no. is' || gvn\_no); DBMS\_output.put\_line('inverted no. is' || inverted\_no); end;

**OUTPUT:**



# PRACTICAL NO.20

**OBJECTIVE:**

Implementation of Procedure

**THEORY:**

**Procedures:** Procedures are named PL/SQL blocks that can take parameters, perform an action and can be invoked. A procedure is generally used to perform an action and to pass values. Procedures are made up of:

1.A declarative part

2.An executable part

3.An optional exception handling part

Declarative Part: The declarative part may contain declaration of cursors, constants, variables, exceptions and subprograms. These objects are local to the procedure. The objects become invalid once you exit from it.

Executable Part: This part contains PL/SQL block consisting of statements that assign values, control execution and manipulate ORACLE data.

Exception Handling Part: This part contains code that performs action to deal with exceptions raised during the execution of the program.

Syntax:

CREATE OR REPLACE

PROCEDURE [.schema] procedure\_name

(argument {IN, OUT, IN OUT} datatype,){IS, AS}

Variable declarations;

Constant declarations;

BEGIN

PL/SQL subprogram body;

EXCEPTION

Exception PL/SQL block; END;

**Types of Procedures:**

1.Local Procedure

2.Stored Procedure

**LOCAL PROCEDURE:** These procedures are declared within the PL/SQL block and called from the begin section of the PL/SQL block.

The following is a simple example of a procedure:

**PROGRAM:**

declare a number; b number; c number; d number; e number; f number;

procedure process ( a in number, b in number, c out number, d out number, e out number, f out number) is begin c:=a+b; d:=a-b; e:=a\*b; f:=a/b; end; begin a:=&firstnumber; b:=&secondnumber;

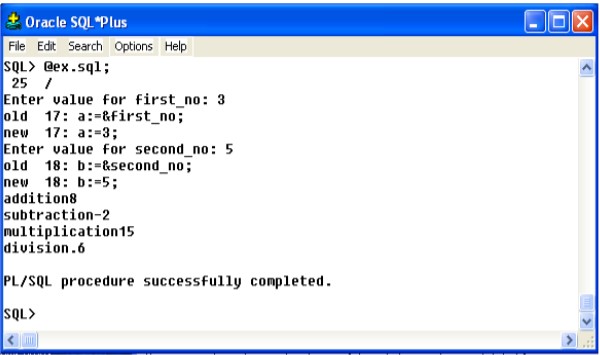
process(a, b, c, d, e, f);

DBMS\_output.put\_line(‘addition is’ || c);

DBMS\_output.put\_line(‘subtraction is’ || d);

DBMS\_output.put\_line(‘multiplication is’ || e); DBMS\_output.put\_line(‘division is’ || f); end;

**OUTPUT:**



# PRACTICAL NO.21

**OBJECTIVE:**

Implementation of Triggers.

**THEORY:**

**Triggers:** A trigger is a special kind of stored procedure that is invoked whenever an attempt is made tomodify the data in the table it protects. Triggers are automatically executed or firedwhensome events occur. Modifications to the table are made using INSERT, UPDATE,ORDELETE statements. Triggers are used to enforce data integrity and business rules such asautomatically updating summary data. It allows performing cascading delete or updatingoperations. If constraints exist on the trigger table, they are checked prior to the triggerexecution. If constraints are violated statement will not be executed and trigger will not run.Triggers are associated with tables and they are automatic. Triggers are automaticallyinvoked by SQL SERVER. Triggers prevent incorrect, unauthorized, or inconsistent changesto data.

**Uses of Triggers:**

1. A trigger can permit DML statements against a table only if they are issued,during regular business hours or on predetermined weekdays.
2. A trigger can also be used to keep an audit trail of a table.
3. It can be used to prevent invalid transactions.
4. Enforce complex security authorizations.
5. Exception handling.
6. Generation of primary key and foreign key.

**How to apply Triggers:**

A trigger has three basic parts:

1. Triggering Event or Statement:It is a SQL statement that causes a trigger to be fired. It can be an INSERT, UPDATEor DELETE statement for a specific table.
2. Trigger Restriction:A trigger restriction specifies a Boolean expression that must be TRUE for thetrigger to fire. It is an option available for triggers that are fired for each row. Atrigger restriction is specified using a WHEN clause.
3. Trigger Action:A trigger action is the PL/SQL code to be executed when a triggering statement isencountered and any trigger restriction evaluates to TRUE.

**Types of Triggers:**

* + **Row Triggers:** A row trigger is fired each time the table is affected by the triggering statement, for example, if an UPDATE statement update multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.
  + **Statement Triggers:** A row trigger is fired once on behalf of the triggering statement, independent of the number of rows the triggering statement affects.
  + **Before Triggers:** Before triggers execute the trigger action before the triggering statement. These types of triggers are commonly used in the following situations:
    1. BEFORE triggers are used when the trigger action should determine whether or not the triggering statement should be allowed to complete. By using BEFORE trigger, user can eliminate unnecessary processing of the triggering statement.
    2. BEFORE triggers are used to derive specific column values before completing a triggering INSERT or UPDATE statement.
  + **After Triggers:** After triggers execute the trigger after the triggering statement is executed. These types of triggers are commonly used in the following situations:
    1. AFTER triggers are used when the triggering statement should complete before executing the trigger action.
    2. If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

**Combinations Triggers:** Using the above triggers, four types of triggers could be created. There are twelve combinations of triggers.

**Before Statement Trigger:** Before executing the triggering statement, the trigger action is executed.

**Before Row Trigger:** Before modifying each row affected by the triggering statement and BEFORE applying appropriate integrity constraints, the trigger is executed.

**After Statement Trigger:** After executing the triggering statement and applying and deferred integrity constraints, the trigger action is executed.

**After Row Trigger:** After modifying each row affected by the triggering statement and applying appropriate integrity constraints, the trigger action is executed for the current row. Unlike BEFORE row triggers, AFTER row triggers have rows locked.

**Creation of Triggers:** Triggers are created with the CREATE TRIGGER statement. This statement specifies that the on which table trigger is defined and on which events trigger will be invoked.

Syntax:

|  |
| --- |
| CREATE OR REPLACE TRIGGER [Schema.] &lt;TriggerName&gt;  {BEFORE, AFTER, Instead of}  {DELETE,INSERT,UPDATE [OF Column,…]}  ON[Schema.] &lt;Tablename&gt;  [REFERENCING {OLD AS old, NEW as new}]  [FOR EACH ROW [WHEN condition]]  DECLARE  &lt;Variable declarations&gt;;  &lt;Constant declarations&gt;;  BEGIN  &lt;PL/SQL subprogram body&gt;;  EXCEPTION  &lt;Exception PL/SQL block&gt;;  END; |

**Deleting a Trigger:** To drop Trigger one can use DROP TRIGGER statement.

Syntax:

DROP TRIGGER &lt;TriggerName&gt;

Example

To start with, using the CUSTOMERS table

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

The following program creates a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table.

This 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 &gt; 0) DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.salary - :OLD.salary; dbms\_output.put\_line(&#39;Old salary: &#39; || :OLD.salary); dbms\_output.put\_line(&#39;New salary: &#39; || :NEW.salary); dbms\_output.put\_line(&#39;Salary difference: &#39; || sal\_diff);

END;

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

Trigger created.

The following points need to be considered here −

* OLD and NEW references are not available for table-level triggers, rather use themfor record-level triggers.
* To query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
* The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but a trigger can be written on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

**Triggering a Trigger:**

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, &#39;Kriti&#39;, 22, &#39;HP&#39;, 7500.00 );

When a record is created in the CUSTOMERS table, the above create

trigger, display\_salary\_changes will be fired and it will display the following result −

Old salary:

New salary: 7500

Salary difference:

Because this is a new record, old salary is not available and the above result comes as null.

The UPDATE statement will update an existing record in the table −

UPDATE customers

SET salary = salary + 500

WHERE id = 2;

When a record is updated in the CUSTOMERS table, the above create

trigger, display\_salary\_changes will be fired and it will display the following result −

Old salary: 1500

New salary: 2000

Salary difference: 500