Program-1

Aim:

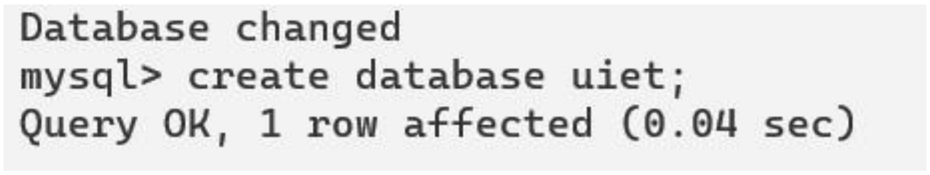
To create a database and writing SQL queries to retrieve information from the database.

Description:

SQL is a database query language used for storing and managing data in
Relational DBMS. SQL was the first commercial language introduced for E.F Codd's Relational model of database. SQL is used to perform all types of data operations in RDBMS.

Creation of database:

“Create Database” statement is used to create a database



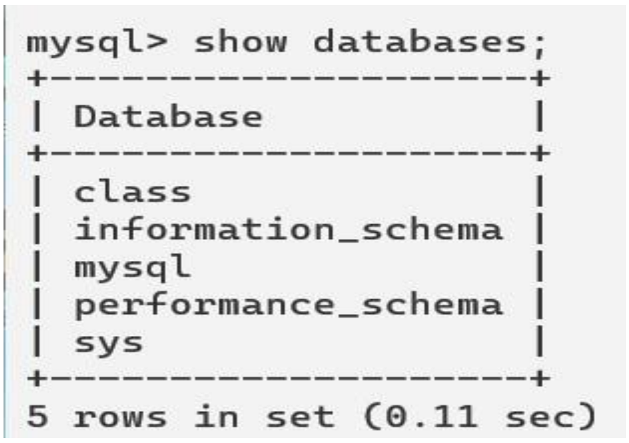
```
Database changed
mysql> create database uiet;
Query OK, 1 row affected (0.04 sec)
```

Syntax:

Create Database {Database Name};
“Show Databases” statement is used to show all the databases in SQL Server.

Syntax:

Show Databases;

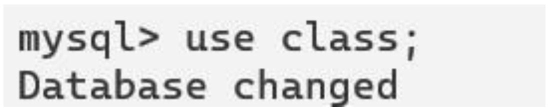


```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| class              |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.11 sec)
```

“Use” statement is used to select a database

Syntax:

Use {Database Name};



```
mysql> use class;
Database changed
```

Creation of Table:

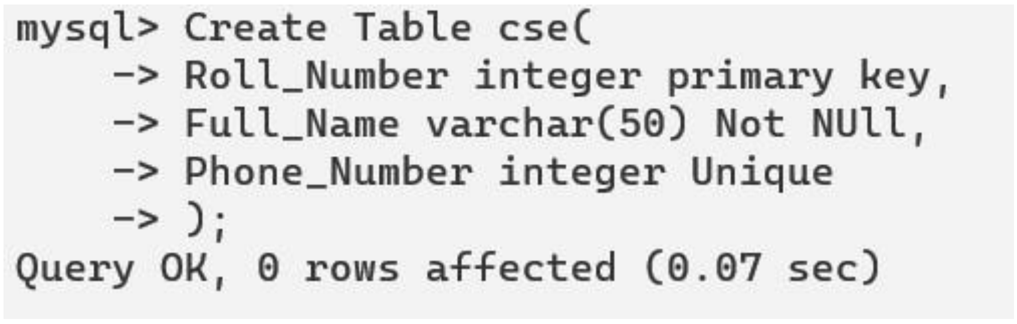
Constraints can be specified when the table is created with the "Create Table" statement, or after the table is created with the "Alter Table" statement.
The following constraints are commonly used in SQL:
Not Null: Ensures that a column cannot have a null value.
Unique: Ensures that all values in a column are different.
Primary Key: Uniquely identifies each row in a table.
Foreign Key: Uniquely identifies a row/record in another table.
Check: Ensures that all values in a column satisfies a specific condition.
Default: Sets a default value for a column when no value is specified.
Index: Used to create and retrieve data from the database very quickly.

Syntax:

```
Create Table {Table Name}(
        Column_1 datatype constraint,
        Column_2 datatype constraint,
        Column_3 datatype constraint,
        .....);
```

Example:

```
Create Table cse(
Roll_Number integer primary key,
Full_Name varchar(50) Not NUll,
Phone_Number integer Unique
);
```

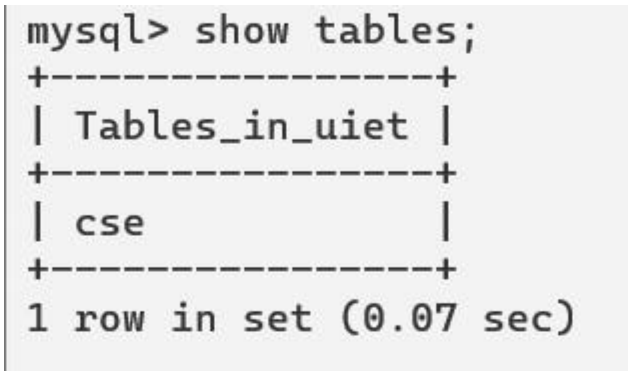


```
mysql> Create Table cse(
    -> Roll_Number integer primary key,
    -> Full_Name varchar(50) Not NUll,
    -> Phone_Number integer Unique
    -> );
Query OK, 0 rows affected (0.07 sec)
```

"Show Table" Statement is used to get a list of all the tables in the database selected.

Syntax:

Show Tables;



```
mysql> show tables;
+----------------+
| Tables_in_uiet |
+----------------+
| cse            |
+----------------+
1 row in set (0.07 sec)
```

Program 2

Aim:

To Insert, View, Delete, Alter, Modify and Update records based on conditions.

Description:

Inserting Values into table:

“Insert into” Statement is used to insert values in a table. The values which are inserted in table are specified with the help of “Values” statement.

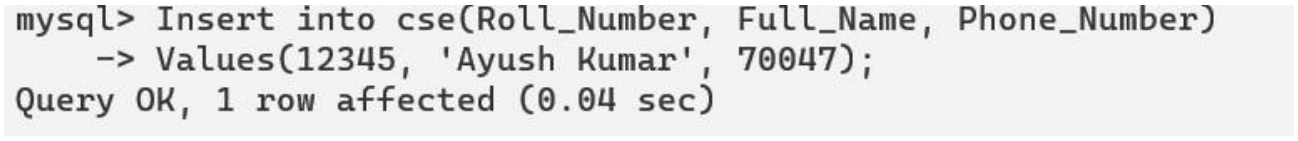
Syntax:

Insert into{table_name}(column_1, column_2, column_3...)

Values(value_1, Value_2, Value_3 ...);

Example:

Insert into cse(Roll_Number, Full_Name, Phone_Number)
Values(12345, ‘Ayush Kumar’, 70047);



```
mysql> Insert into cse(Roll_Number, Full_Name, Phone_Number)
    -> Values(12345, 'Ayush Kumar', 70047);
Query OK, 1 row affected (0.04 sec)
```

View records from a table:

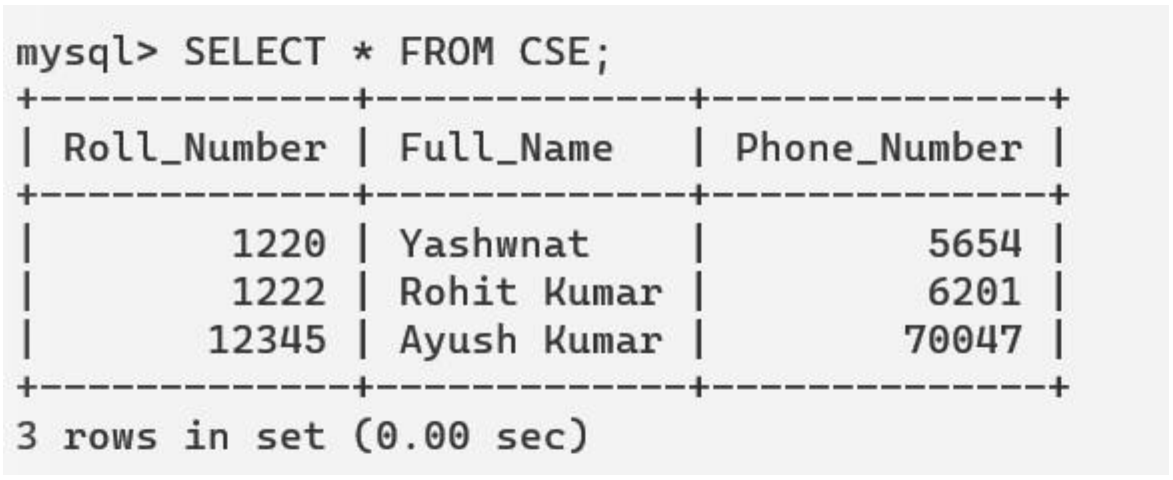
“Select” and “from” is used to view records from a table. “where” is used to specify conditions if we want to see only a particular information satisfying that condition. “*” is used in case we want to see all the column instead of only specifying all the columns individually.

Syntax:

Select {Column_Names} From {Table_Name} where {condition};

Example:

Select * from cse;



```
mysql> SELECT * FROM CSE;
+-------------+-------------+--------------+
| Roll_Number | Full_Name   | Phone_Number |
+-------------+-------------+--------------+
|        1220 | Yashwnat    |         5654 |
|        1222 | Rohit Kumar |         6201 |
|       12345 | Ayush Kumar |        70047 |
+-------------+-------------+--------------+
3 rows in set (0.00 sec)
```

Deleting a record from table:

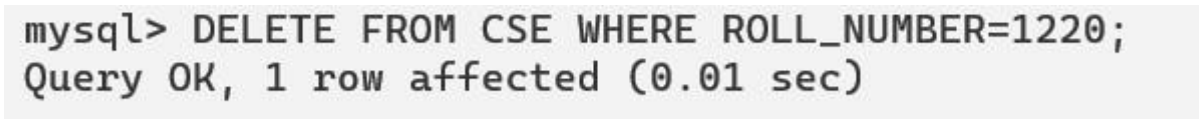
"Delete From" is for delete a record. It is used with "where" which is used to specify the conditon where we want to delete the record.

Syntax:

Delete from {table name} where {condition};

Example:

Delete from cse where Roll_Number = 12348;

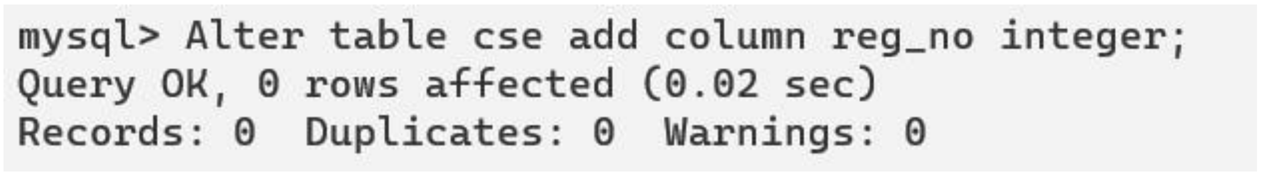


```
mysql> DELETE FROM CSE WHERE ROLL_NUMBER=1220;
Query OK, 1 row affected (0.01 sec)
```

Altering a Table:

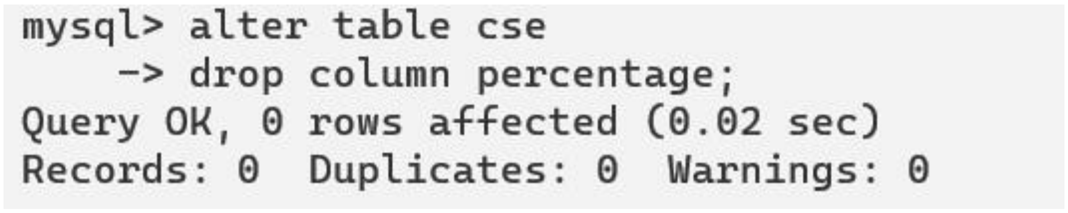
Unlike other command up until now this command do not affect records but it affects the table itself.
"Alter" is used to change the structure of the table.

- **Adding a new column:**
  Used to add a new column in an existing table.
  Syntax:
  Alter Table {table name}
  Add column {column_name datatype}; **Example:**
  Alter table cse add column percentage integer;



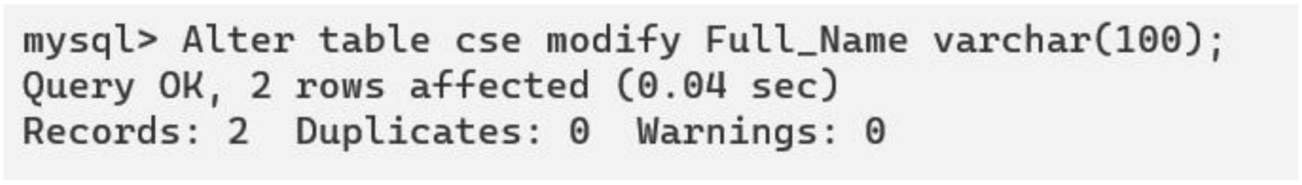
```
mysql> Alter table cse add column reg_no integer;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

- **Deleting an existing column:**
  Used to delete a column in an existing table.
  Syntax:
  Alter table {Table name} Drop column
  {column_name}; **Example:**
  Alter table cse drop column percentage;



```
mysql> alter table cse
    -> drop column percentage;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

- **Modify a column:**
  used to modify a column **Syntax:**
  Alter table {Table_name} Modify column datatype; **Example:**
  Alter table cse modify Full_Name varchar(100);



```
mysql> Alter table cse modify Full_Name varchar(100);
Query OK, 2 rows affected (0.04 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

Updating a record:

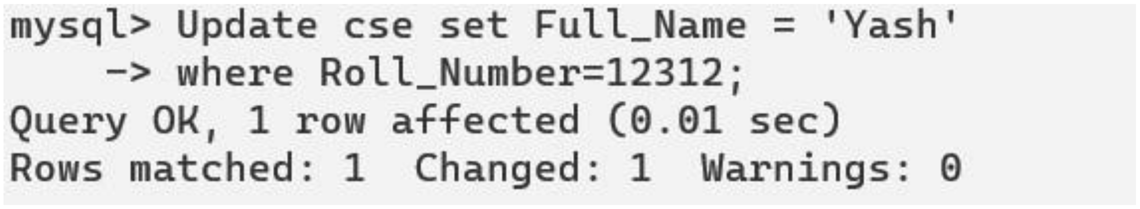
"Update" and "set" statement are used to update records along with the "where" statement.

Syntax:

Update {table_name} set {column} = {new value} where {condition};

Example:

Update cse set Full_Name = 'Yash' where Roll_number = 12312;



```
mysql> Update cse set Full_Name = 'Yash'
    -> where Roll_Number=12312;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Program 3

Aim:

To create Views, Synonyms, Sequence, Indexes, Save point.

Description:

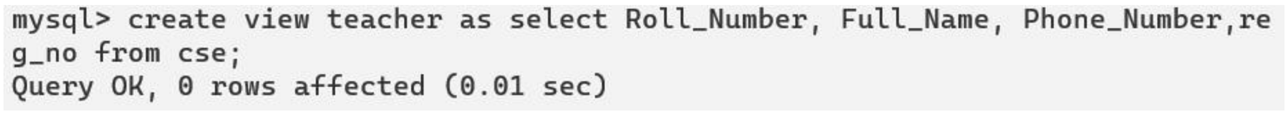
View:

View is an object which gives the user a logical view of data from an underlying table. We can restrict what users can view by allowing them to see only a few columns from a table When a view is created from more than one table, the user can view data from the view without using join conditions and complex conditions Views also hide the names of the underlying tables View is stored as a “Select” statement in the Data Dictionary View contains no data of its own Any updation of rows in the table will automatically reflect in the views A query fired on a view will run slower than a query fired on a base table.

Creation of a view:

Create view {view_name} as select {column_1}, {column_2}... from {table_name} where {condition};

This query help in creating a view (i.e., a virtual table based on result set of SQL statement it contains rows & columns just like a table) of a table.

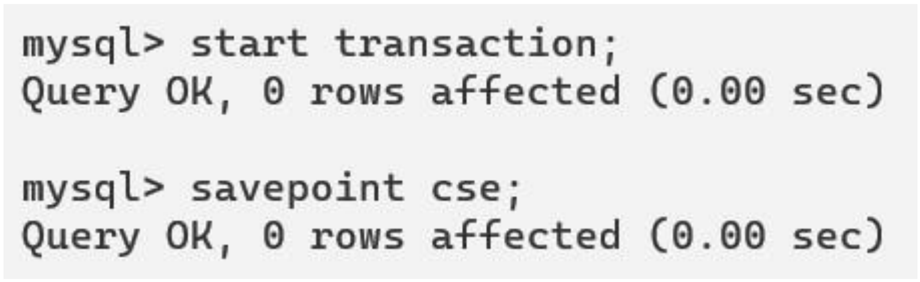


```
mysql> create view teacher as select Roll_Number, Full_Name, Phone_Number,re
g_no from cse;
Query OK, 0 rows affected (0.01 sec)
```

Deletion of a view:

Drop view {view_name};

This query is used to delete a view of a table.

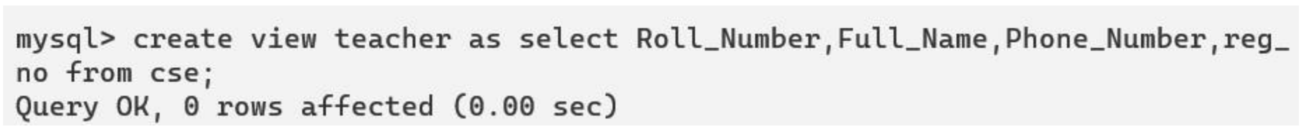


```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> savepoint cse;
Query OK, 0 rows affected (0.00 sec)
```

To add or remove fields in a view:

create or replace view {view_name} as select {column_1}, {column_2}..... from {table_name} where {condition};

This query is used to modify the structure of view i.e., to add or remove fields in a view.



```
mysql> create view teacher as select Roll_Number,Full_Name,Phone_Number,reg_
no from cse;
Query OK, 0 rows affected (0.00 sec)
```

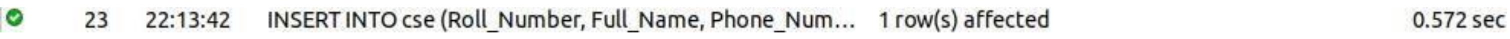
Inserting a row in a view: insert into {view_name} (column_1,
column_2 ...)
values(value_1, value_2 ...);

This query is used to insert a new record in that view and in original table and also in all other views of that table.

Deleteing a row from a view:
Delete from {view_name} where {condition};

This query is used to delete the record that satisfy the given condition & that record is deleted from other views also & also from original table.

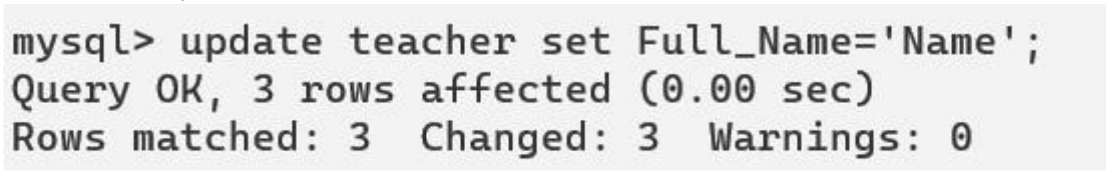
Updating Views:
Update {view_name} set {column_name} = {new vaue} where {condition};



```
23   22:13:42   INSERT INTO cse (Roll_Number, Full_Name, Phone_Num...   1 row(s) affected   0.572 sec
```

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is not met, then we will not be allowed to update the view.

- The select statement which is usd to create the view should not include group by clause or order by clause.
- The select statement should not have the distinct keyword.
- The view should have all not null values.
- The view should not be created using nested quesries or complex queries.
- The view should be created from single table. If the view is created using multiple tables than we will not be allowed to update the view.

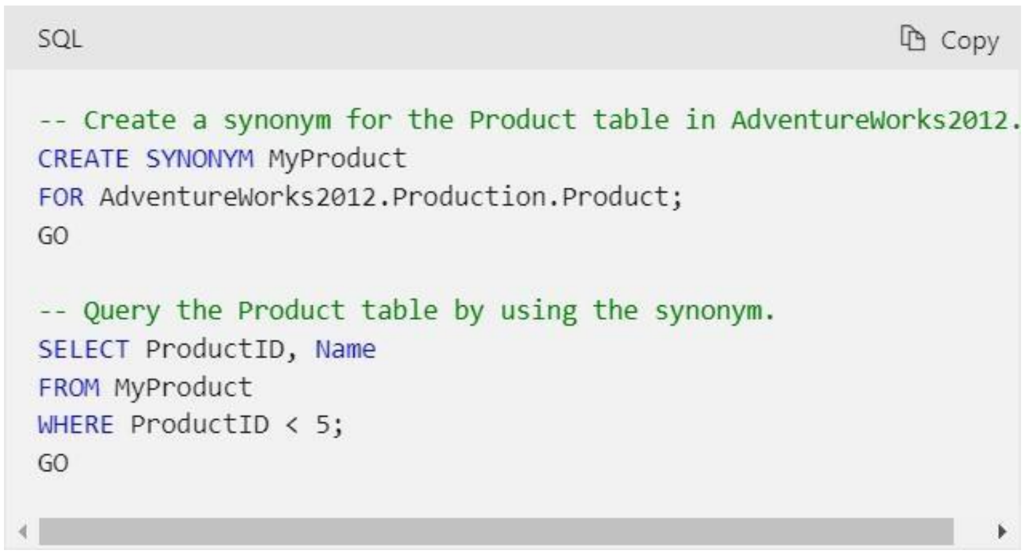


```
mysql> update teacher set Full_Name='Name';
Query OK, 3 rows affected (0.00 sec)
Rows matched: 3  Changed: 3  Warnings: 0
```

Synonyms:
Synonyms are used to create alternate names for tables, views, sequences, etc.

Creating a synonym:
Create synonym {syn_name} for {object_name};

This query creates a synonym for any object i.e., any table, view, etc.



```sql
SQL                                                          Copy

-- Create a synonym for the Product table in AdventureWorks2012.
CREATE SYNONYM MyProduct
FOR AdventureWorks2012.Production.Product;
GO

-- Query the Product table by using the synonym.
SELECT ProductID, Name
FROM MyProduct
WHERE ProductID < 5;
GO
```

View the details of a user synonyms:
Select {syn_name}, {table_name}, {table_owner} from user_synonyms; This query is used to view the deatils of the synonyms.

Droping a synonym:
Drop synonym {syn_name};
This query is used to delete a synonym.

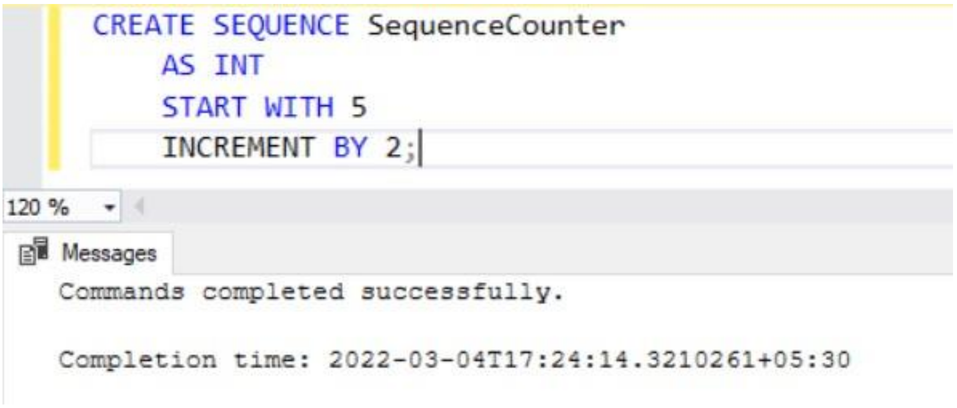
Sequence:
Sequence is used to generate a sequence of numbers. The value generated can have a maximum of 38 digits.
The minimum information required to generate numbers using a sequence are:

- The starting number {s}
- The maximum number {m}
- The increment value {n}

Creating a sequence:
Create sequence {seq_name} increment by {n}

Start with {s} maxvalue {m} {cache/ nocache};

This query creates a sequence which increment by value n and starts with s with maximum value of m.



```sql
CREATE SEQUENCE SequenceCounter
    AS INT
    START WITH 5
    INCREMENT BY 2;
```

```
120 %
Messages
Commands completed successfully.

Completion time: 2022-03-04T17:24:14.3210261+05:30
```

Currval : Gives the current value in sequence. **Nextval:** Gives the next value in sequence.

Select sqn_name.currval from dual; Select
sqn_name.nextval form dual;

Modifying a Sequence:

Modification of a sequence does not allow us to change the “start with” option. Similarly, the maximum value cannot be set to a number less than the current number.
Alter sequence {seq_name} increment by {n}
Start with {s} maxvalue {m} {cache/ nocache};

Drop a Sequence:

A sequence can dropped:

Drop sequence {sqn_name};

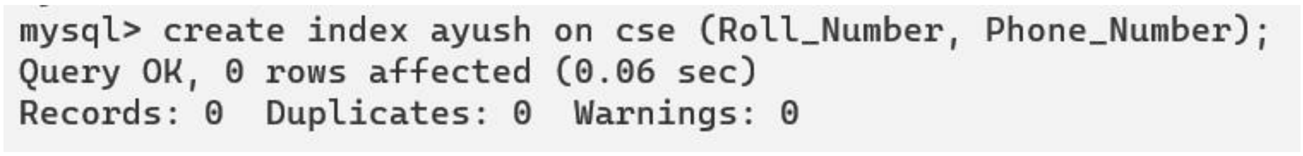
Index:

Index is used for faster retrieval of rows from a table. It can be used implicitly or explicitly.

Creating a Index:

Create index {in_name} on {table_name}(column_1, column_2...);

This query is used to create index on one or more columns.



```
mysql> create index ayush on cse (Roll_Number, Phone_Number);
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Rebuilding an index:

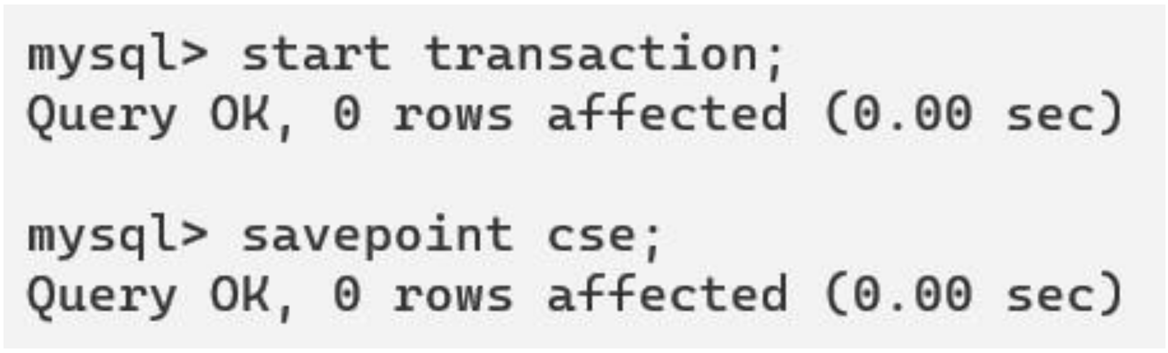
Alter index {in_name} Rebuild;
When a table goes through changes, it is advisable to rebuild indexes based on the table.

Savepoint:

Savepoint is a command in SQL that is used with the rollback command. It is a command in Transaction Control Language that is used to mark the transaction in a table.If you made a transaction in a table, you could mark the transaction as a certain name, and later on, if you want to roll back to that point, you can do it easily by using the transaction's name.

Creating a savepoint:

To create a savepoint we first have to start our transaction with begin/start.
Start Transaction;
Savepoint {sp_name};

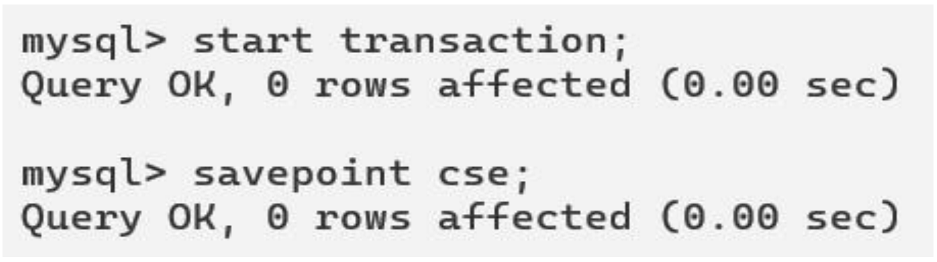


```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> savepoint cse;
Query OK, 0 rows affected (0.00 sec)
```

Rolling back to savepoint:

To roll back to a savepoint we use “Rollback to”.

Rollback to {sp_name};



```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> savepoint cse;
Query OK, 0 rows affected (0.00 sec)
```

Deleting a savepoint:

There is no syntax to delete a savepoint. A savepoint gets automatically deleted when we commit or rollback the trasaction.

Program-4

Aim:

Creating an Employee database to set various constraints.

Description:

Constraints are the rules that we can apply on the type of data in a table. That is, we can specify the limit on the type of data that can be stored in a particular column in a table using constraints.

The available constraints in SQL are:

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.

UNIQUE: This constraint when specified with a column, tells that all the values in the column must be unique. That is, the values in any row of a column must not be repeated.

PRIMARY KEY: A primary key is a field which can uniquely identify each row in a table. And this constraint is used to specify a field in a table as primary key.

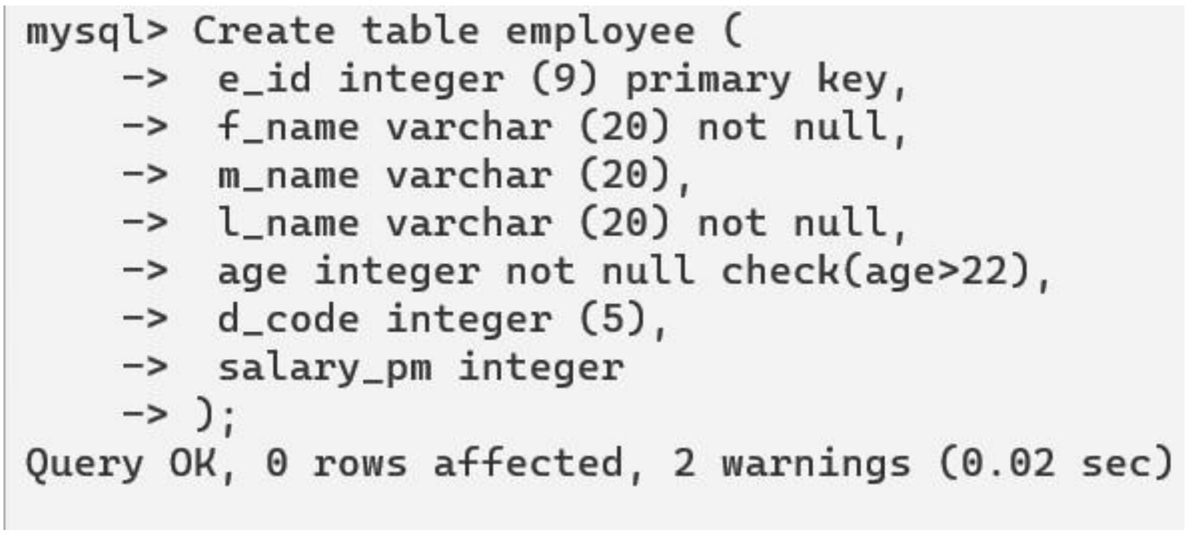
FOREIGN KEY: A Foreign key is a field which can uniquely identify each row in a another table. And this constraint is used to specify a field as Foreign key.

CHECK: This constraint helps to validate the values of a column to meet a particular condition. That is, it helps to ensure that the value stored in a column meets a specific condition.

DEFAULT: This constraint specifies a default value for the column when no value is specified by the user.

Create table employee with various constraints:

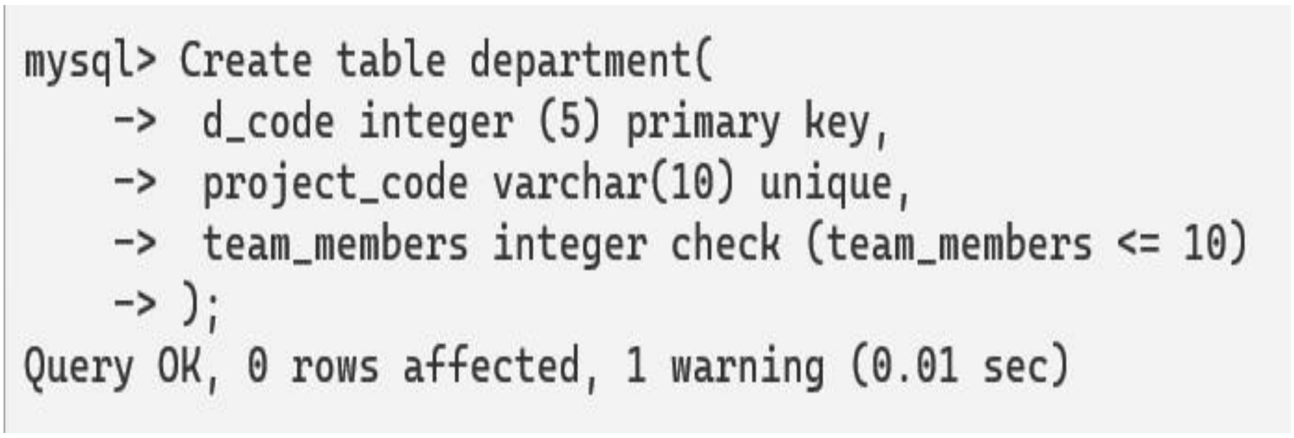
```
Create table employee ( e_id integer (9) primary
        key, f_name varchar (20) not null,
        m_name varchar (20),
        l_name varchar (20) not null, age integer not null
        check(age>22), d_code integer (5), salary_pm
        integer

);
```



```
mysql> Create table employee (
    ->  e_id integer (9) primary key,
    ->  f_name varchar (20) not null,
    ->  m_name varchar (20),
    ->  l_name varchar (20) not null,
    ->  age integer not null check(age>22),
    ->  d_code integer (5),
    ->  salary_pm integer
    -> );
Query OK, 0 rows affected, 2 warnings (0.02 sec)
```

Create table department:

```
Create table department( d_code integer (5) primary
                key, project_code varchar(10) unique,
          team_members integer check (team_members <= 10)
                                                        );
```



```
mysql> Create table department(
    ->  d_code integer (5) primary key,
    ->  project_code varchar(10) unique,
    ->  team_members integer check (team_members <= 10)
    -> );
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

Program - 5

Aim:

Creating relationship between the databases.

Description:

In sql, we can create a relationsip by creating a foreign key constraint. More, specifically we have a parent table and a child table. The parent contains the primary key and the child table contains the foreign key that references to the primary key of the parent table.
When we use SQL to create a relationship, we can create the relationship at the time we create the table, or we can create it later by altering the table.

Create a Relationship When Creating the Table:

Here's an example of creating a relationship within your "Create Table" statement at the time you create the table.

Syntax:

```
Create table Parent_Table(
        P_column_1 datatype primary key,
        P_column_2 datatype
        P_column_3 datatype
);
Create table Child_Table(
        C_Column_1 datatype primary key,
        C_Column_2 datatype
        C_column_3 datatype constraint Parent_Child foreign key (C_Column_3)
        References Parent_Table (P_Column_1)
);
```

Here we created two tables; one called "Parent_Table" and the other called

"Child Table".

Add a Relationship to an Existing Table:

You can also add a relationship to an existing table, simply by using the

"Alter Table" statement. For Example, let say we didn't create a foreign key in the table in previous example of Parent_child relationship and we want to create a relationship now, after we have created the tables.

Syntax:

Alter Table Child_Table

Add Constraint Parent_Child

Foreign Key(C_Column_3)

References Parent_Table(P_Column_1);

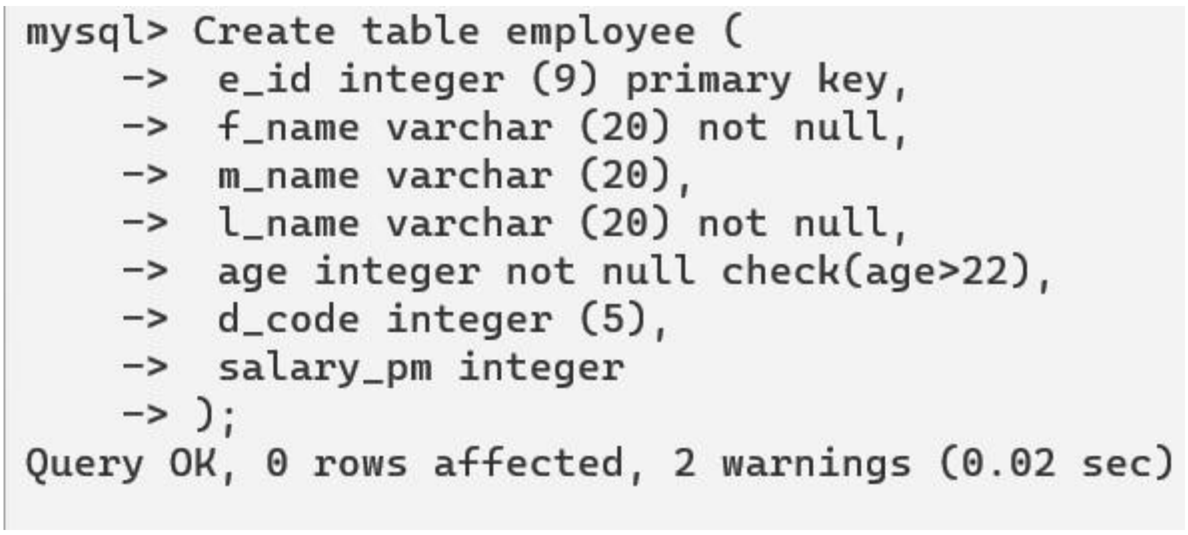
Note: SQLite Doesn't support adding foreign keys with the Alter Table Statement.

Example:

Create table employee (e_id integer (9) primary key,
f_name varchar (20) not null, m_name varchar
(20) default "n/a", l_name varchar (20) not null,
age integer not null check(age>22), d_code integer
(5),

foreign key (d_code) references department(d_code), salary_pm integer

);

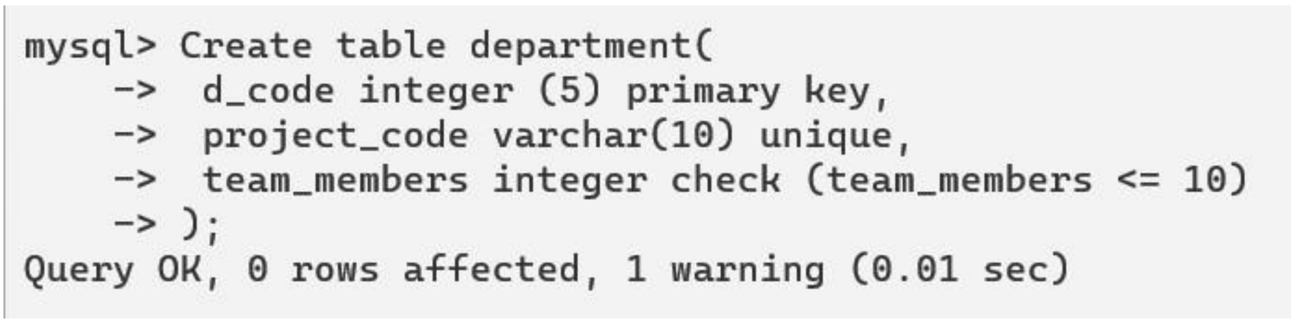


```
mysql> Create table employee (
    ->  e_id integer (9) primary key,
    ->  f_name varchar (20) not null,
    ->  m_name varchar (20),
    ->  l_name varchar (20) not null,
    ->  age integer not null check(age>22),
    ->  d_code integer (5),
    ->  salary_pm integer
    -> );
Query OK, 0 rows affected, 2 warnings (0.02 sec)
```

Create table department:

Create table department(d_code integer (5) primary
key, project_code varchar(10) unique,
team_members integer check (team_members
<= 10)

);



```
mysql> Create table department(
    ->  d_code integer (5) primary key,
    ->  project_code varchar(10) unique,
    ->  team_members integer check (team_members <= 10)
    -> );
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

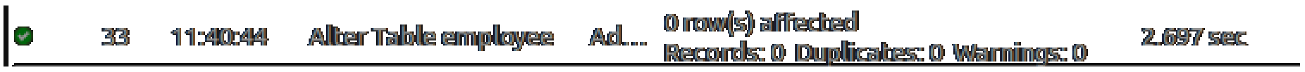
Add a Foreign key to an existing table in SQLite:

By default, SQL Server relationships are created using “on delete no action” and “on update no action”. Therefore, the previous examples were created using this setting.

However, different DBMSs may use other default settings.

Either way, you can explicitly specify this in your code. So we can modify the previous example to look like this: **Syntax:**

```
Alter Table employee

        Add Constraint Parent_Child

        Foreign Key (C_Column_3)

        References Parent_Table (P_Column_1)

        On delete no action

        On update no action;
```



```
33    11:40:44    Alter Table employee    Ad...    0 row(s) affected
                                                   Records: 0  Duplicates: 0  Warnings: 0    2.697 sec
```

What this actually means is that, if someone was to try to delete or update a record in the “Primary Key”, an error would occur and the change would be rolled back. This is SQL Server’s way of preventing any changes that could break the referential integrity of your system.

Basically, the reason you create a relationship in the first place is to enforce referential integrity.

However, you do have some options with how you want SQL Server to deal with these situations.

Specifically, you can use any of the following values:

- **No Action:** An error is raised and the delete/update action on the row in the parent table is rolled back.
- **Cascade:** Corresponding rows are deleted from/ updated in the referencing table if that row is deleted from/updated in the parent table.
- **Set Null:** All the values that make up the foreign key are set to “Null” if the corresponding row in the parent table is deleted or updated. This requires that the foreign key columns are nullable.
- **Set Default:** All the values that make up the foreign key are set to their default values if the corresponding row in the parent table is deleted or updated. For this constraint to execute, all

foreign key columns must have default definitions. If a column is nullable, and there is no explicit default value set, “Null” becomes the implicit default value of the column.

Experimnt-6

Aim:

Study of PL/SQL block.

Description:

- **PL/SQL:**

PL/SQL stands for Procedural Language extension of SQL.

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

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

A Simple PL/SQL Block:

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

PL/SQL Block consists of three sections:

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

Declaration Section:

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

Execution Section:

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

Exception Section

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors. Every statement in the above three sections must end with a semicolon ; . PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

A Sample PL/SQL Block Looks like:

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

PL/SQL Block Structure:

```
DECLARE
        v_variable VARCHAR2(5);
BEGIN
        SELECT column_name INTO v_variable
        FROM table_name;
EXCEPTION
        WHEN exception_name THEN
        ...
END;
```

Block Types

1. Anonymous

```
        [DECLARE]
BEGIN --statements
        [EXCEPTION] END;
```

2. Procedure

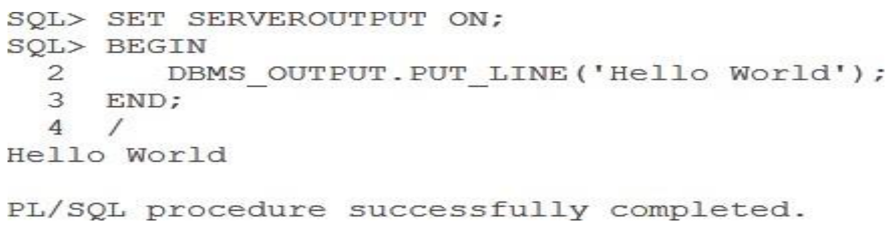
```
        PROCEDURE name
IS BEGIN --statements
        [EXCEPTION] END;
```

3. Function

```
        PROCEDURE name
        FUNCTION name
        RETURN datatype
IS BEGIN --statements
        RETURN value;
```

[EXCEPTION] END;

Result:



```
SQL> SET SERVEROUTPUT ON;
SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE('Hello World');
  3  END;
  4  /
Hello World

PL/SQL procedure successfully completed.
```

Thus the PL/SQL blocks are studied.

Program – 7

Aim:

Write a PL/SQL block to satisfy some conditions by accepting input from the user.

Description:

PL/SQL Control Structure provides conditional tests, loops, flow control and branches that let to produce well-structured programs.

Syntax:

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

PL/ SQL General Syntax:

```
SQL> declare
        <variable declaration>; begin
        <executable statement >; end;
```

PL/ SQL General Syntax for if Condition:

```
SQL> declare
        <variable    declaration>;    begin
        if(condition) then
                <executable statement >;
        end;
```

PL/ SQL General Syntax for If and Else Condition: SQL> declare

```
        <variable declaration>; begin if
        (test condition) then
                <statements>; else
                <statements>; end
        if; end;
```

PL/ SQL General Syntax for Nested if Condition: SQL> declare

```
        <variable declaration>; begin if
        (test condition) then
                <statements>; else if (test
        condition) then
                <statements>; else
                <statements>; end
        if; end;
```

PL/ SQL General syntax for Looping statement: SQL> declare

```
        <variable declaration>; begin

        loop
                <statement>; end loop;
        <executable statement>; end;
```

PL/ SQL General Syntax For Looping Statement: SQL> declare

```
        <variable declaration>; begin
        while <condition> loop
                <statement>; end loop;
        <executable statement>; end;
```

PL/SQL Coding for addition of two numbers:

PROCEDURE:

Step 1: Start

Step 2: Initialize the necessary variables.

Step 3: Develop the set of statements with the essential operational parameters.

Step 4: Specify the Individual operation to be carried out.

Step 5: Execute the statements.

Step 6: Stop.

PROGRAM:

```
SQL>          set serveroutput on
SQL>        declare
a number;
b number;

c number;

begin   a: =&a;        b: =&b;
```

```
                        c: =a+b;
dbms_output.put_line ('sum of' ||a|| 'and '||b||' is '||c);

                      end;
                     /
```

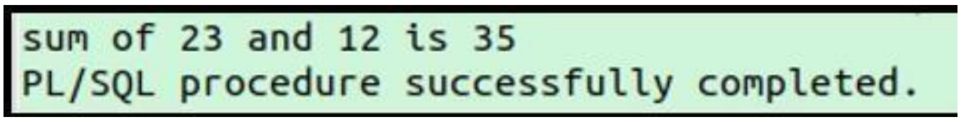
Input:

Enter value for a: 23 old:

a:=&a; new: a:=23;

Enter value for b: 12 old:

b:=&b; new: b:=12; **Output:**



```
sum of 23 and 12 is 35
PL/SQL procedure successfully completed.
```

PL/ SQL Program for if Condition:

(Write a PL/SQL Program to find out the maximum value using if condition) **Procedure:**

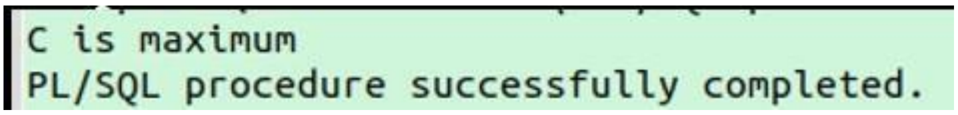
Step 1: Start
Step 2: Initialize the necessary variables.

Step 3: invoke the if condition. **Step 4:** Execute the statements.

Step 5: Stop.
Program:

```
SQL>            set serveroutput on

SQL>          declare

             b number; c
             number;

             BEGIN

             B:=10;  C:=20;  if(C>B)  THEN  dbms_output.put_line('C  is
             maximum');
             end if; end;

             /
```

Output:



```
C is maximum
PL/SQL procedure successfully completed.
```

PL/ SQL Program for If Else Condition:

(Write a PL/SQL Program to check whether the value is less than or greater than 5 using if else condition) **Procedure:**

Step 1: Start

Step 2: Initialize the necessary variables.

Step 3: invoke the if else condition.

Step 4: Execute the statements.

Step 5: Stop.

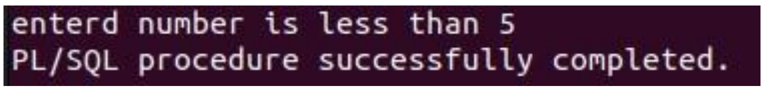
Program:

```
SQL>          set serveroutput on

SQL>        declare

            n number; begin

            dbms_output. put_line('enter a number');
            n:=&number; if n<5
            then
               dbms_output.put_
            line('entered number
            is less than 5');

            else   dbms_output.put_line('entered number is greater than 5');
              end if;
             end;

            /
```

Input:

Enter value for number: 2 old 5:

n:=&number; new 5: n:=2; **Output:**



```
enterd number is less than 5
PL/SQL procedure successfully completed.
```

PL/ SQL Program for If Else If Condition:

(Write a PL/SQL Program to find the greatest of three numbers using if else if)

Procedure:

Step 1: Start

Step 2: Initialize the necessary variables.

Step 3: invoke the if else if condition.

Step 4: Execute the statements.

Step 5: Stop.

Program:

```
SQL>          set server output on

SQL>        declare

            a number; b
            number; c
            number; begin
```

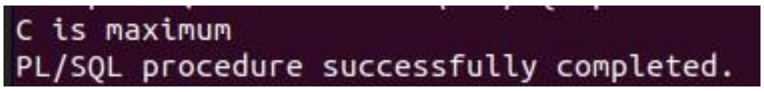
```
a:=&a; b:=&b;
c:=&c;

if(a>b)and(a>c) then dbms_output.put_line('A is maximum');

else if(b>a)and(b>c)then dbms_output.put_line('B is maximum');

else dbms_output.put_line('C is maximum');
  end if;
 end;

/
```

Input:

Enter value for a: 21 old:
a:=&a; new: a:=21;

Enter value for b: 12 old:
b:=&b; new: b:=12;

Enter value for b: 45 old:
c:=&b; new: c:=45; **Output:**



```
C is maximum
PL/SQL procedure successfully completed.
```

PL/ SQL Program for Looping Statement:

(Write a PL/SQL Program to find the summation of odd numbers using for loop)

Procedure:

Step 1: Start

Step 2: Initialize the necessary variables.

Step 3: invoke the for loop condition.

Step 4: Execute the statements.

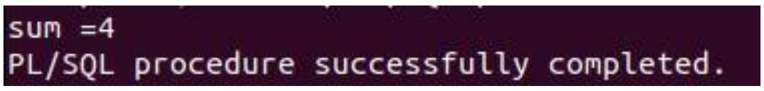
Step 5: Stop.

Program:

```
SQL>            set server output on

SQL>          declare

              n number; sum1 number default
              0; end value number; begin

              end value:=&end value; n:=1;

              for n in 1..endvalue loop if mod(n,2)=1
                      then sum1:=sum1+n;

                      end if; end
                      loop;

              dbms_output.put_line('sum ='||sum1); end;

              /
```

Input:

Enter value for end value: 4 old : end
value:=&end value; new : end value:=4;
Output:



```
sum =4
PL/SQL procedure successfully completed.
```

PL/ SQL Program for looping statement:
(Write a PL/SQL Program to find the factorial of given number using for loop)

Procedure:
Step 1: Start
Step 2: Initialize the necessary variables.

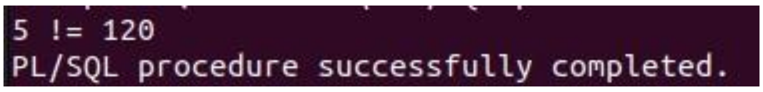
Step 3: invoke the for loop condition.

Step 4: Execute the statements.

Step 5: Stop.
Program:

```
SQL>          set server output on

SQL>        declare

n number;

i number;

p number:=1;

begin

n:=&n;

for i in 1..n loop

p:=p*i;

end loop;

dbms_output.put_line(n ||' ! = '||p);

end;
```

Input:
Enter value for n: 5 old :
n:=&n; new: n:=5; **Output:**



```
5 != 120
PL/SQL procedure successfully completed.
```

Result:
Thus a PL/SQL block to satisfy some conditions by accepting input from the user was created using oracle.

Program-8

Aim:

Write a PL/SQL block that handles all types of exceptions.

Description:

In PL/SQL, the user can catch certain runtime errors. Exceptions can be internally defined by Oracle or the user. Exceptions are used to handle errors that occur in your PL/SQL code. A PL/SQL block contains an EXCEPTION block to handle exception.

There are three types of exceptions:

1. Predefined Oracle errors
2. Undefined Oracle errors
3. User-defined errors

The different parts of the exception.

1. Declare the exception.
2. Raise an exception.
3. Handle the exception.

An exception has four attributes:

1. Name provides a short description of the problem.
2. Type identifies the area of the error.
3. Exception Code gives a numeric representation of the exception.
4. Error message provides additional information about the exception. The predefined divide-by-zero exception has the following values for the attributes:

1. Name = ZERO_DIVIDE
2. Type = ORA (from the Oracle engine)
3. Exception Code = C01476 Error message = divisor is equal to zero **Exception Handling:**

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

PL/SQL Exception message consists of three parts. Type of Exception

An Error Code

A message

Structure Of Exception Handling
General Syntax For Coding The Exception Section

DECLARE

Declaration section

BEGIN

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

Types of Exceptions:

There are 2 types of Exceptions.

a) System Exceptions
b) User-defined Exceptions

a) System Exceptions

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

For example: NO_DATA_FOUND and ZERO_DIVIDE are called Named System exceptions.

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

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

b) User-defined Exceptions

PL/SQL allows us to define our own exceptions according to the need of our program. A user-defined exception must be declared and then raised explicitly, using a RAISE statement. To define an exception we use EXCEPTION keyword as below:

```
EXCEPTION_NAME EXCEPTION;
```

To raise exception that we‟ve defined to use the RAISE statement as follows:

```
RAISE EXCEPTION_NAME
```

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command RAISE. Following is the simple syntax of raising an exception:

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

TYPES OF MORE COMMONLY USED EXCEPTIONS:

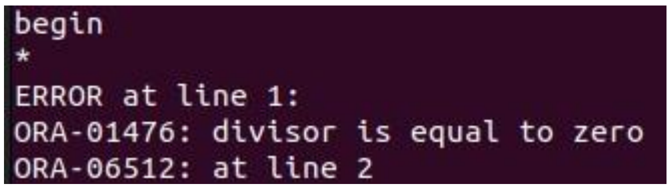
NO_DATA_FOUND	Singleton Select statement returned no data.
TOO_MANY_ROWS	Singleton Select statement returned more than one row.
INVALID_CURSOR	Illegal cursor operation occured.
VALUE_ERROR	Arithmetic, conversion, or truncation error occured.
INVALID_NUMBER	Conversion of a number to a character string failed.
ZERO_DIVIDE	Attempted to divide by zero
DUP_VAL_ON_INDEX	Attempted to insert a duplicate value into a column that has a unique index.
CURSOR_ALREADY_OPEN	Attempted to open a cursorthat was previously opened.
NOT_LOGGED_ON	A database call eas made without being logged in.
TRANSACTION_BACKED_OUT	Usually raised when a remote portion of a transaction is rolled back.
LOGIN_DENIED	Failed to login.
PROGRAM_ERROR	If PL/SQL encounters an internal problem.
STORAGE_ERROR	If PL/SQL runs out of memory or if memory is corrupted.
TIMEOUT_ON_RESOURCE	Timeout occured while PL/SQL was waiting for a resource.
OTHERS	For all of the rest.

Program:

ZERO_DIVIDE EXCEPTION

```
SQL>            BEGIN

            DBMS_OUTPUT.PUT_LINE(1 / 0);

            END;

            /
```

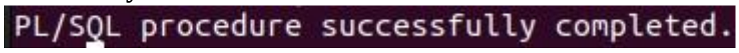
Output:



```
begin
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 2
```

```
BEGIN

      DBMS_OUTPUT.PUT_LINE(1 / 0);

      EXCEPTION

            WHEN ZERO_DIVIDE THEN

            DBMS_OUTPUT.PUT_LINE('Division by zero');

      END;

      /
```

Division by zero:

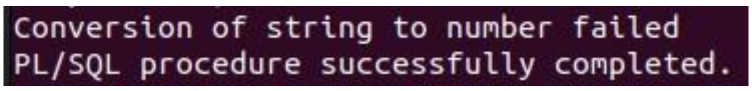


```
PL/SQL procedure successfully completed.
```

INVALID_NUMBER EXCEPTION

```
BEGIN

INSERT INTO employees(DEPARTMENT_ID)VALUES('101x');

EXCEPTION

      WHEN INVALID_NUMBER THEN

      DBMS_OUTPUT.PUT_LINE('Conversion of string to number failed');

end;

/
```

Output:



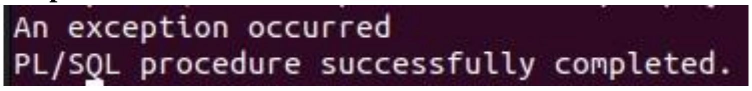
```
Conversion of string to number failed
PL/SQL procedure successfully completed.
```

Other Exceptions

```
BEGIN

DBMS_OUTPUT.PUT_LINE(1 / 0);
```

```
EXCEPTION

WHEN OTHERS THEN

DBMS_OUTPUT.PUT_LINE('An exception occurred');

END;

/
```

Output:

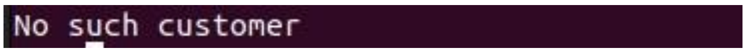


```
An exception occurred
PL/SQL procedure successfully completed.
```

First Create A Table Named Customerss Wth Attribute Id, Name, Address And Then Implement The Following Code:

```
SQL>            declare

                c_id        customerss.id%type;        c_name
                customerss.name%type;                  c_addr
                customerss.address%type;
                begin

                SELECT name,address INTO c_name,c_addr

                FROM customerss WHERE id=c_id;

                     dbms_output.put_line('Name: '|| c_name);

                     dbms_output.put_line('Address: ' || c_addr);

                  EXCEPTION
                           WHEN no_data_found THEN
                             dbms_output.put_line('No such customer!');

                      WHEN others THEN

                          dbms_output.put_line('Error!');
                 END;
                /
```

Output:



```
No such customer
```

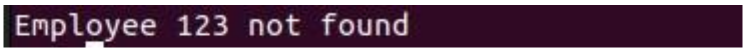
Program:

(The following example illustrates the programmer-defined exceptions. Get

the salary of an employee and check it with the job’s salary range. If the salary is below the range, an exception BELOW_SALARY_RANGE is raised. If the salary is above the range, exception ABOVE_SALARY_RANGE is raised)

```
SET SERVEROUTPUT ON SIZE 100000;

DECLARE

     -- define exceptions
```

```
        BELOW_SALARY_RANGE EXCEPTION;

        ABOVE_SALARY_RANGE EXCEPTION;

        -- salary variables

        n_salary        employees.salary%TYPE;        n_min_salary
        employees.salary%TYPE;                        n_max_salary
        employees.salary%TYPE;
        -- input employee id

        n_emp_id employees.employee_id%TYPE := &emp_id;

BEGIN

        SELECT salary, min_salary max_salary INTO n_salary, n_min_salary, n_max_salary

        FROM employees

        INNER JOIN jobs ON jobs.job_id = employees.job_id

        WHERE employee_id = n_emp_id;

        IF n_salary < n_min_salary THEN

                RAISE BELOW_SALARY_RANGE;

        ELSIF n_salary > n_max_salary THEN RAISE ABOVE_SALARY_RANGE;

        END IF;

        dbms_output.put_line('Employee ' || n_emp_id ||' has salary $' || n_salary );

EXCEPTION WHEN BELOW_SALARY_RANGE THEN

                dbms_output.put_line('Employee ' || n_emp_id || ' has salary below the salary range');

        WHEN ABOVE_SALARY_RANGE THEN

                dbms_output.put_line('Employee ' || n_emp_id || ' has salary above the salary range');

        WHEN NO_DATA_FOUND THEN

                                        DBMS_OUTPUT.PUT_LINE('Employee ' ||n_emp_id || ' not
        found');

END;

/
```



```
Employee 123 not found
```

Result:

Thus a PL/SQL block that handles all type of exceptions was written, executed and verified successfully.

Program-9

Aim:

Creation of Procedures.

Description:

PL/SQL Procedure

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

Header: The header contains the name of the procedure and the parameters or variables passed to the procedure.

Body: The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

How to pass parameters in procedure:

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

IN parameters: The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.

OUT parameters: The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.

INOUT parameters: The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

Creation of Procredure:

```
Create {or Replace} Procedure {procedure_name} (parameter, parameter .... ) IS
        {decalaration_section}
BEGIN
        executable_section
{EXCEPTION
        exception section}
END {procedure_name};
/
```

Example:

Table: create table user(id integer(10) primary key,name varchar2(100)); **Procedure:**

```
create procedure insert_user (id in number, name in varchar2) is
begin
insert into user values(id,name); end;
/
```

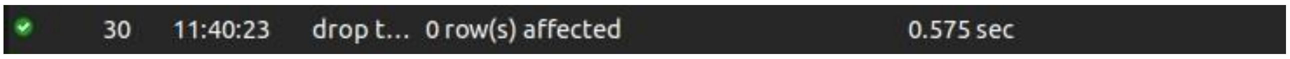
```
6    11:17:07    Create ...    0 row(s) affected    0.195 sec
```

Deleting a procedure:

Drop procedure {procedure_name}; This syntax deletes
the procedure.

Example:

Drop procedure insert_user;



```
30    11:40:23    drop t...  0 row(s) affected                    0.575 sec
```

Program-10

Aim:

Creation of database triggers and functions

Description:

Trigger:

A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated. **Syntax:**

create trigger {trigger_name}

    {before / after}

    {insert / update / delete} on
    {table_name} {for each row}
    {trigger_body}

Explanation of syntax:

1. **create trigger {trigger_name}:** Creates or replaces an existing trigger with the trigger_name.
2. **{before / after}:** This specifies when the trigger will be executed.
3. **{insert / update / delete}:** This specifies the DML operation.
4. **on {table_name}:** This specifies the name of the table associated with the trigger.
5. **{for each row}:** This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. **{trigger_body}:** This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run. AFTER triggers run the trigger action after the triggering statement is run.

Example:

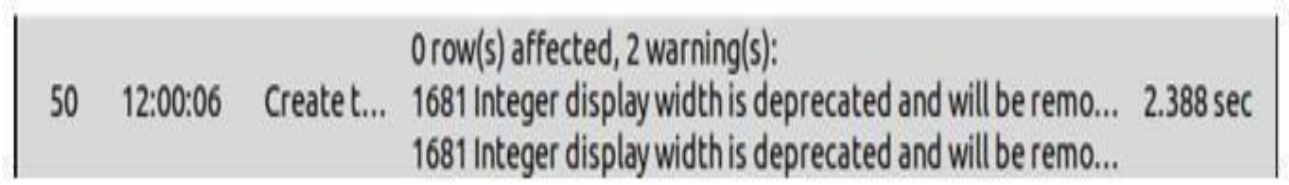
Given Student Report Database, in which student marks assessment is

recorded. In such schema, create a trigger so that the total and percentage of specified marks is automatically inserted whenever a record is insert. Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

create trigger stud_marks

before insert on
Student for each
row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3,

```
Student.per = Student.total * 60 / 100;
```



```
50   12:00:06   Create t...   0 row(s) affected, 2 warning(s):                                   2.388 sec
                              1681 Integer display width is deprecated and will be remo...
                              1681 Integer display width is deprecated and will be remo...
```