FS19CO042

**Practical no. 8**

**Title : Implementation of DCL commands**

* **GRANT**
* **REVOKE**

**Implementation of TCL commands**

* **COMMIT**
* **ROLLBACK**
* **SAVEPOINT**

**Theory :**

1. **DCL :**

**DCL is the abstract of Data Control Language. Data Control Language includes commands such as GRANT, and is concerned with rights, permissions, and other controls of the database system. DCL is used to grant/revoke permissions on databases and their contents. DCL is simple, but MySQL permissions are a bit complex. DCL is about security. DCL is used to control the database transaction. DCL statements allow you to control who has access to a specific object in your database.**

**1. GRANT**

**2. REVOKE**

**GRANT:**

**It provides the user's access privileges to the database. The MySQL database offers both the administrator and user a great extent of the control options. The administration side of the process includes the possibility for the administrators to control certain user privileges over the MySQL server by restricting their access to an entire database or usage limiting permissions for a specific table. It creates an entry in the security system that allows a user in the current database to work with data in the current database or execute specific statements.**

**Syntax :**

**Statement permissions:**

**GRANT { ALL | statement [ ,...n ] }**

**TO security\_account [ ,...n ]**

**Normally, a database administrator first uses CREATE USER to create an account, then GRANT to define its privileges and characteristics.**

**For example:**

**CREATE USER vatsa@'localhost' IDENTIFIED BY 'mypass';**

**GRANT ALL ON MY\_TABLE TO vatsa@'localhost';**

**GRANT SELECT ON Users TO vatsa@'localhost';**

**REVOKE:**

**The REVOKE statement enables system administrators and to revoke (back permission) the privileges from MySQL accounts.**

**Syntax:**

**REVOKE**

**priv\_type [(column\_list)]**

**[, priv\_type [(column\_list)]] ...**

**ON [object\_type] priv\_level**

**FROM user [, user] ...**

**REVOKE ALL PRIVILEGES, GRANT OPTION**

**FROM user [, user] ...**

**For example:**

**REVOKE INSERT ON \*.\* FROM 'vatsa'@'localhost';**

1. **TCL :**

**Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.**

**COMMIT:**

**COMMIT command is used to permanently save any transaction into the database.**

**When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.**

**To avoid that, we use the COMMIT command to mark the changes as permanent.**

**Following is commit command's syntax,**

**Syntax:**

**commit;**

**ROLLBACK:**

**This command restores the database to the last committed state. It is also used with the SAVEPOINT command to jump to a savepoint in an ongoing transaction.**

**If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not committed using the COMMIT command.**

**Following is rollback command's syntax,**

**Syntax:**

**ROLLBACK TO savepoint\_name;**

**SAVEPOINT:**

**SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.**

**Following is savepoint command's syntax,**

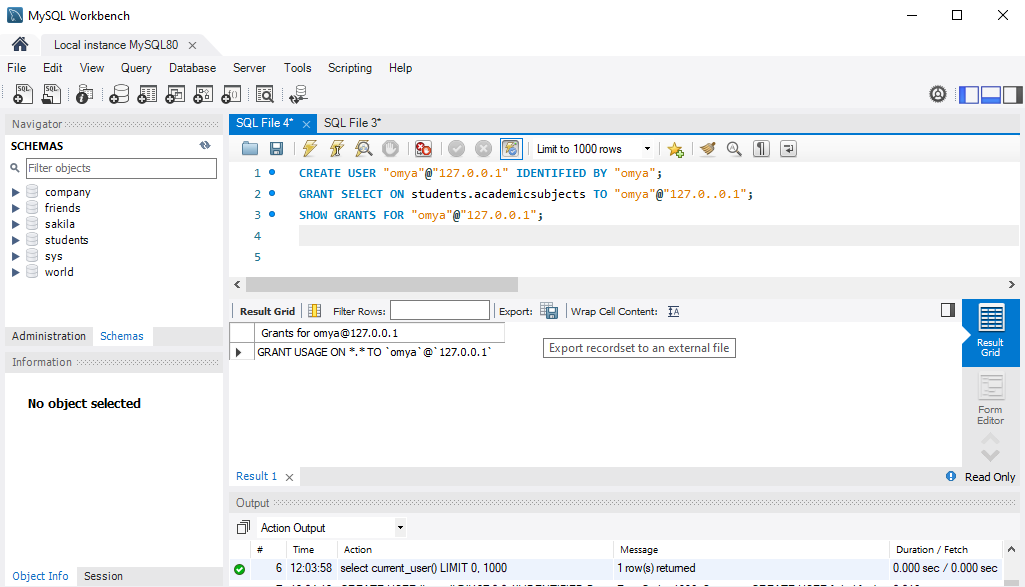
**Syntax:**

**SAVEPOINT savepoint\_name;**

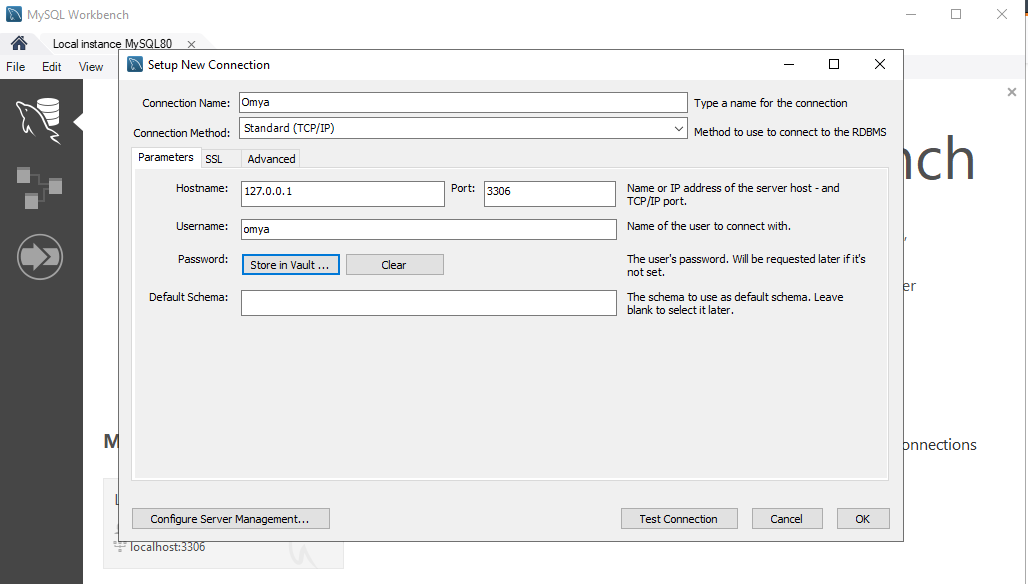
**OUTPUT :**

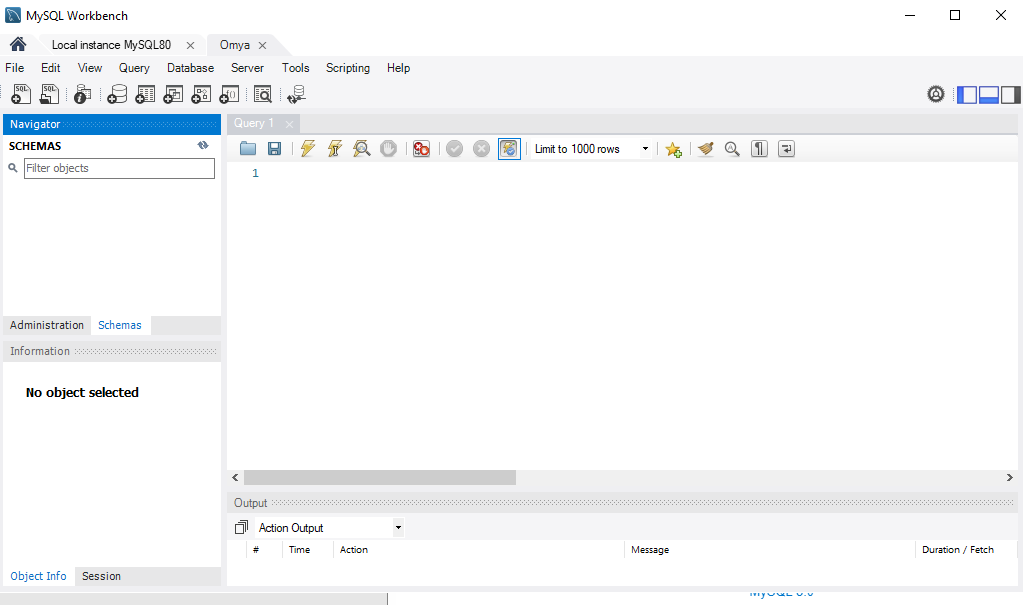
FS19CO042

1. **Practical no. 9 DCL**
2. **GRANT**
3. **Creating user**
4. **Granting permissions**

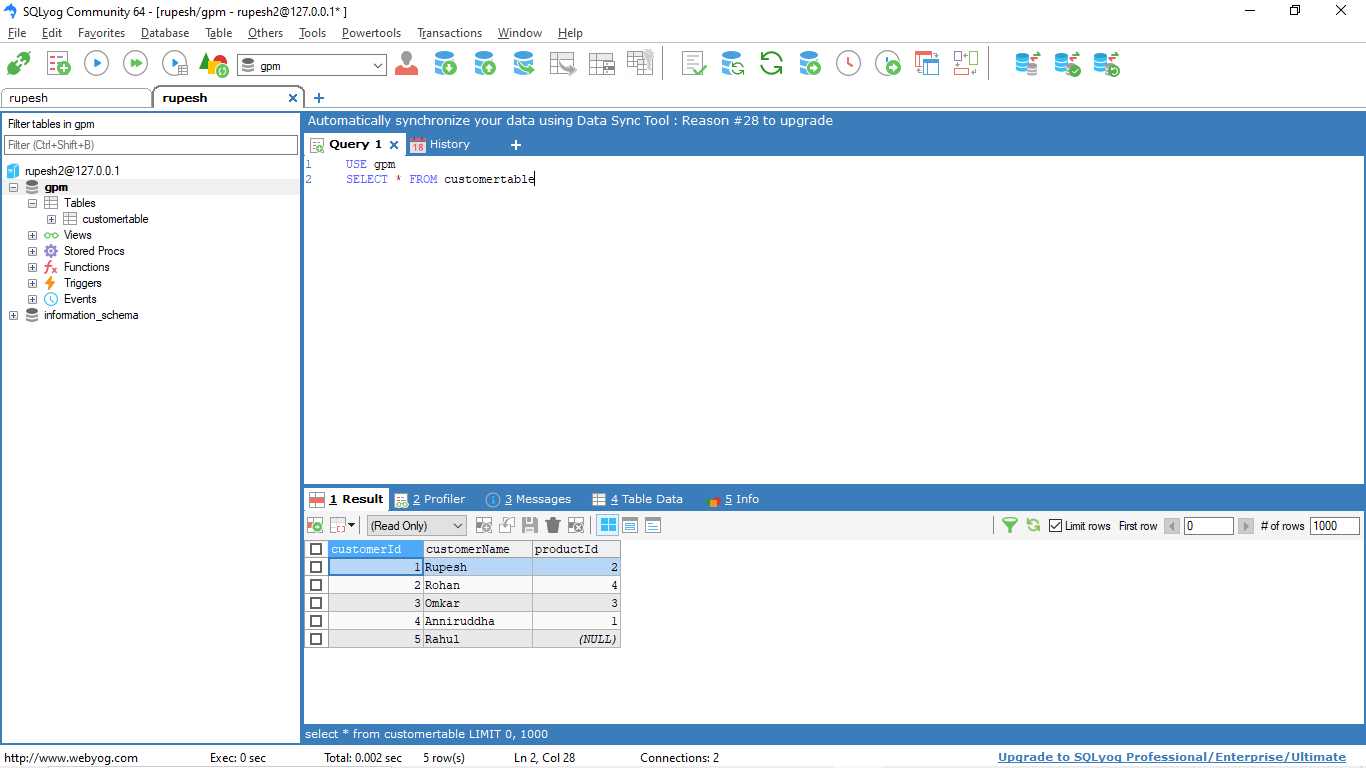
****

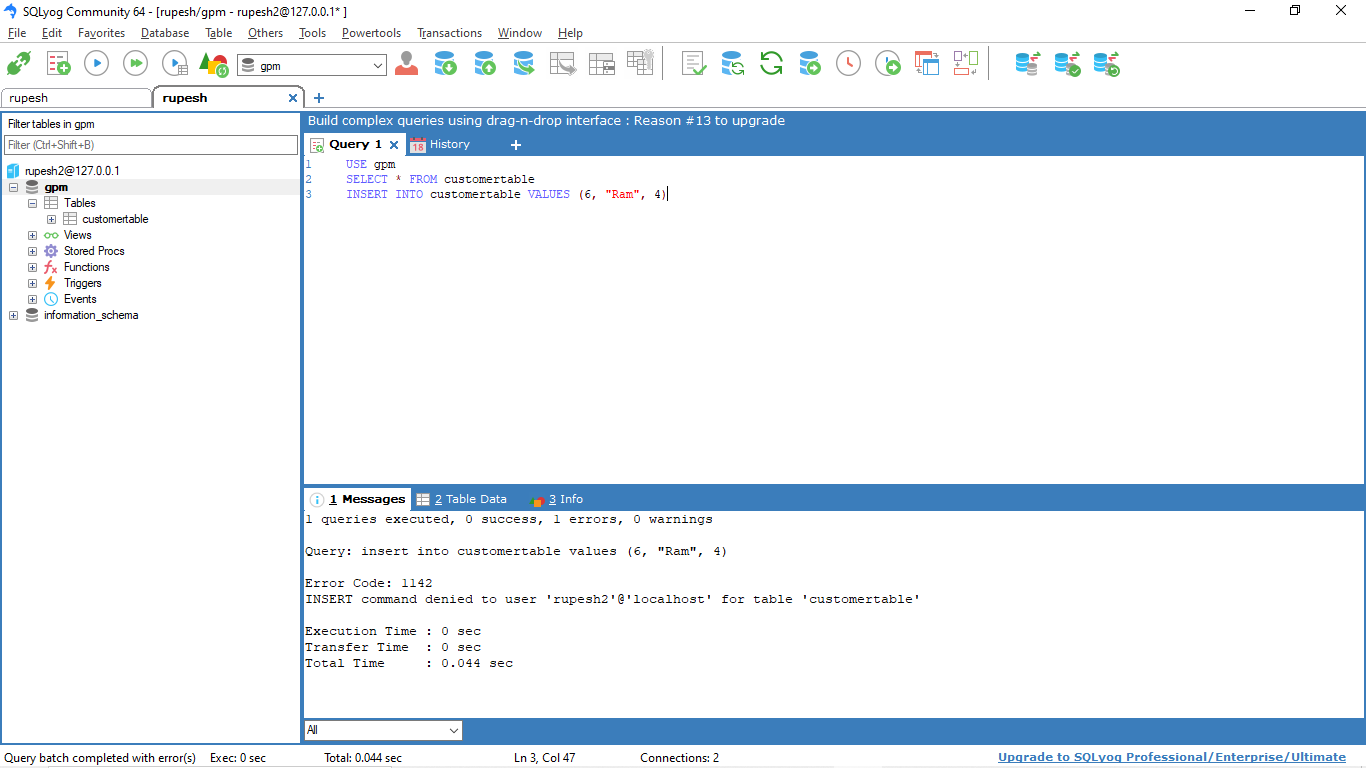
1. **Making new connection**

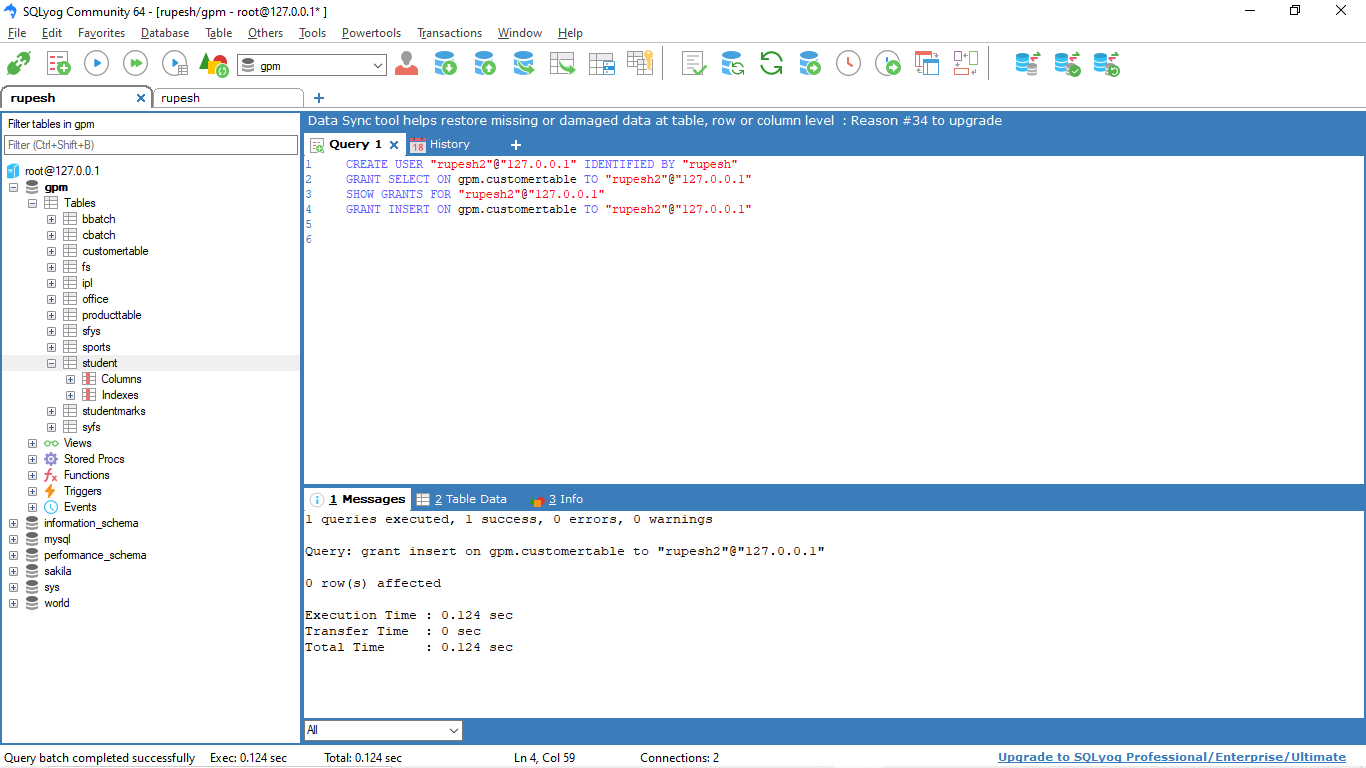
****

****

1. **Use allowed commands**

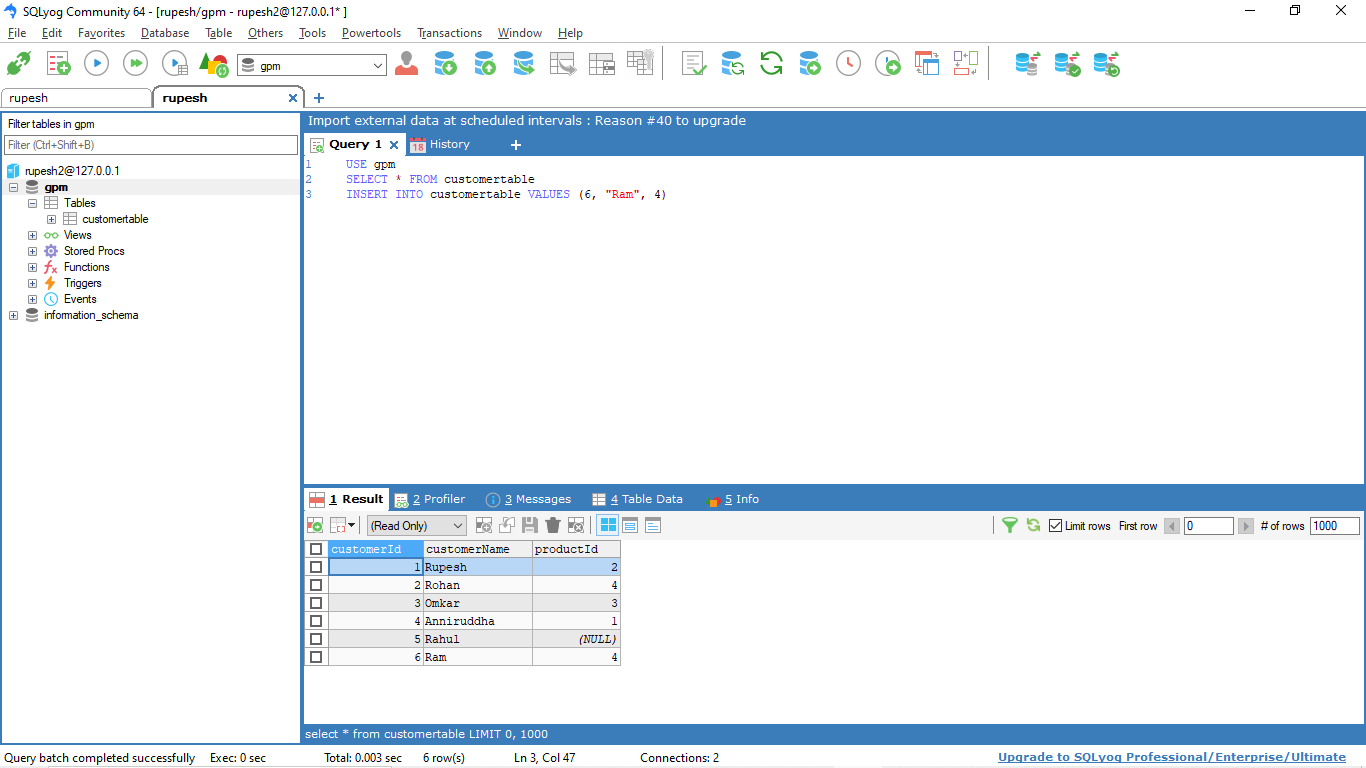
****

****

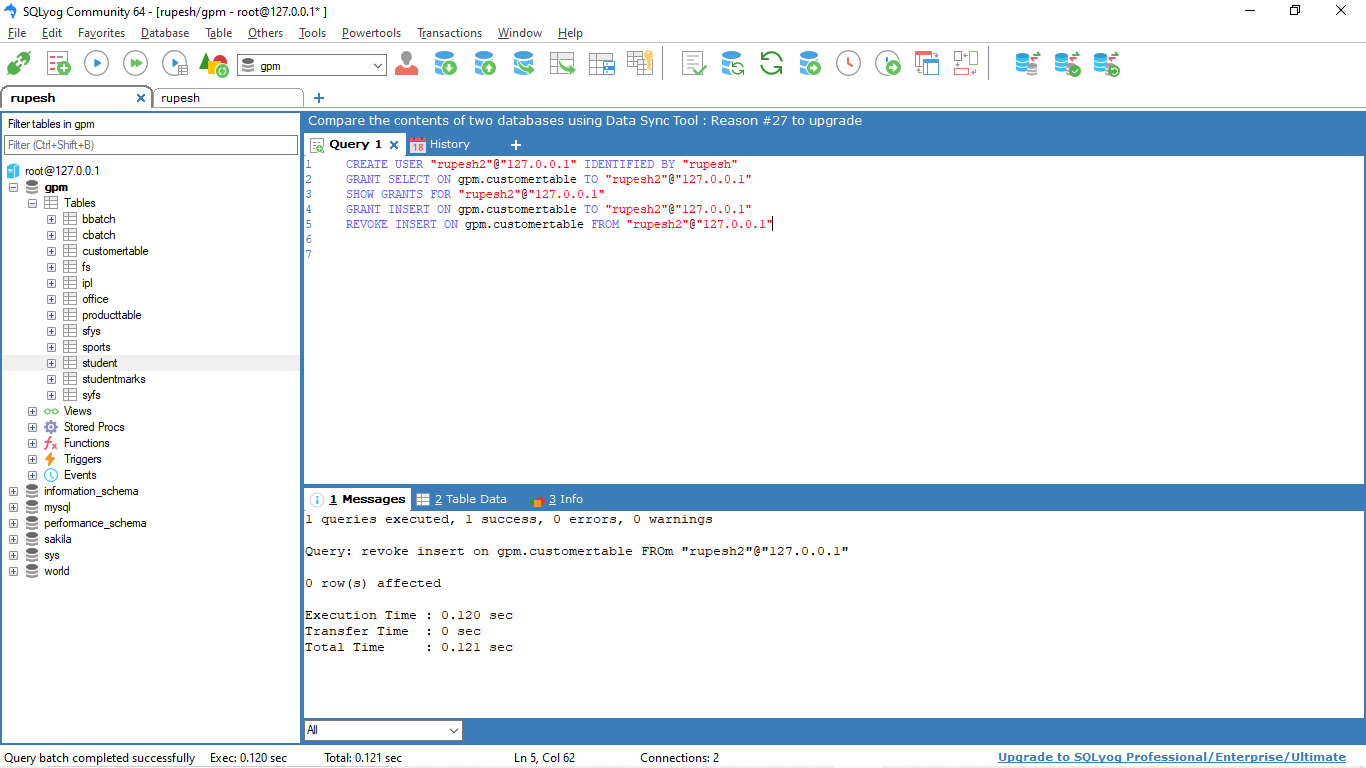
****

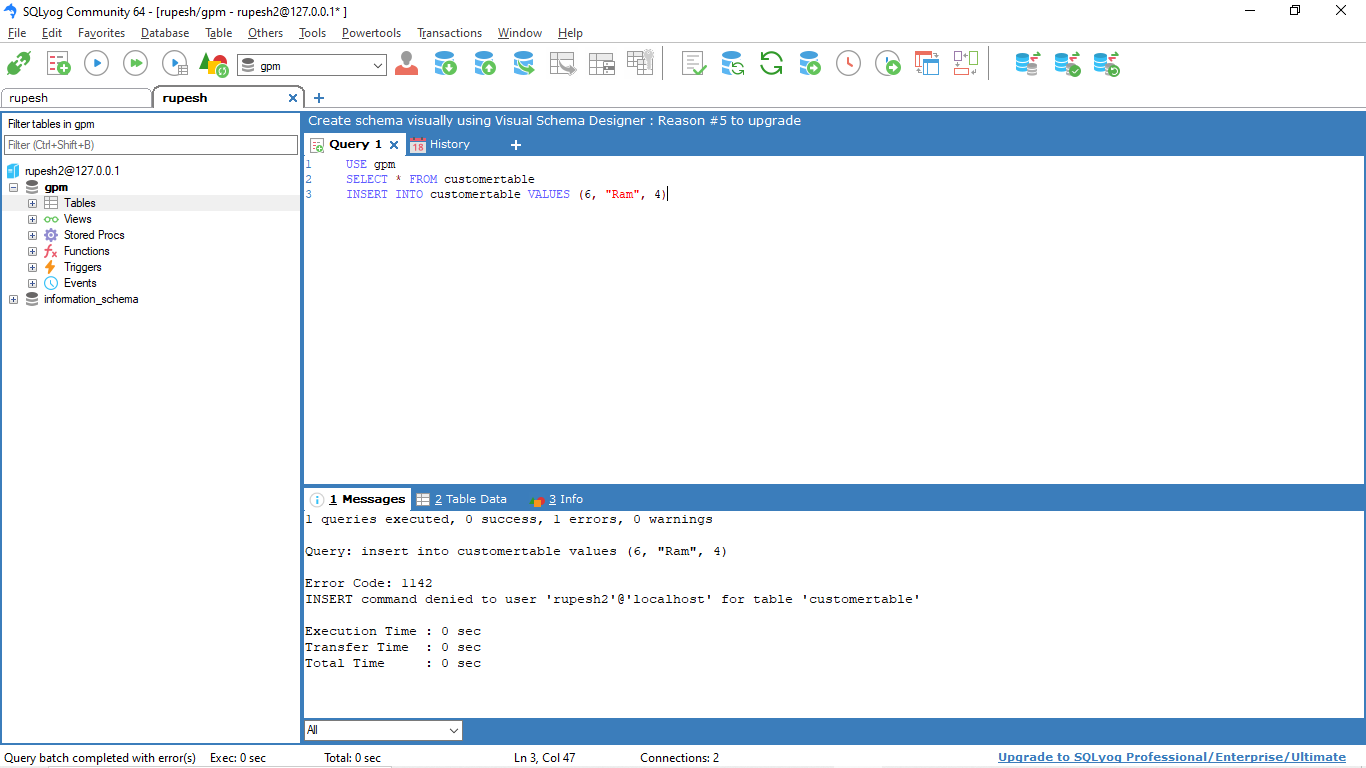
****

****

****

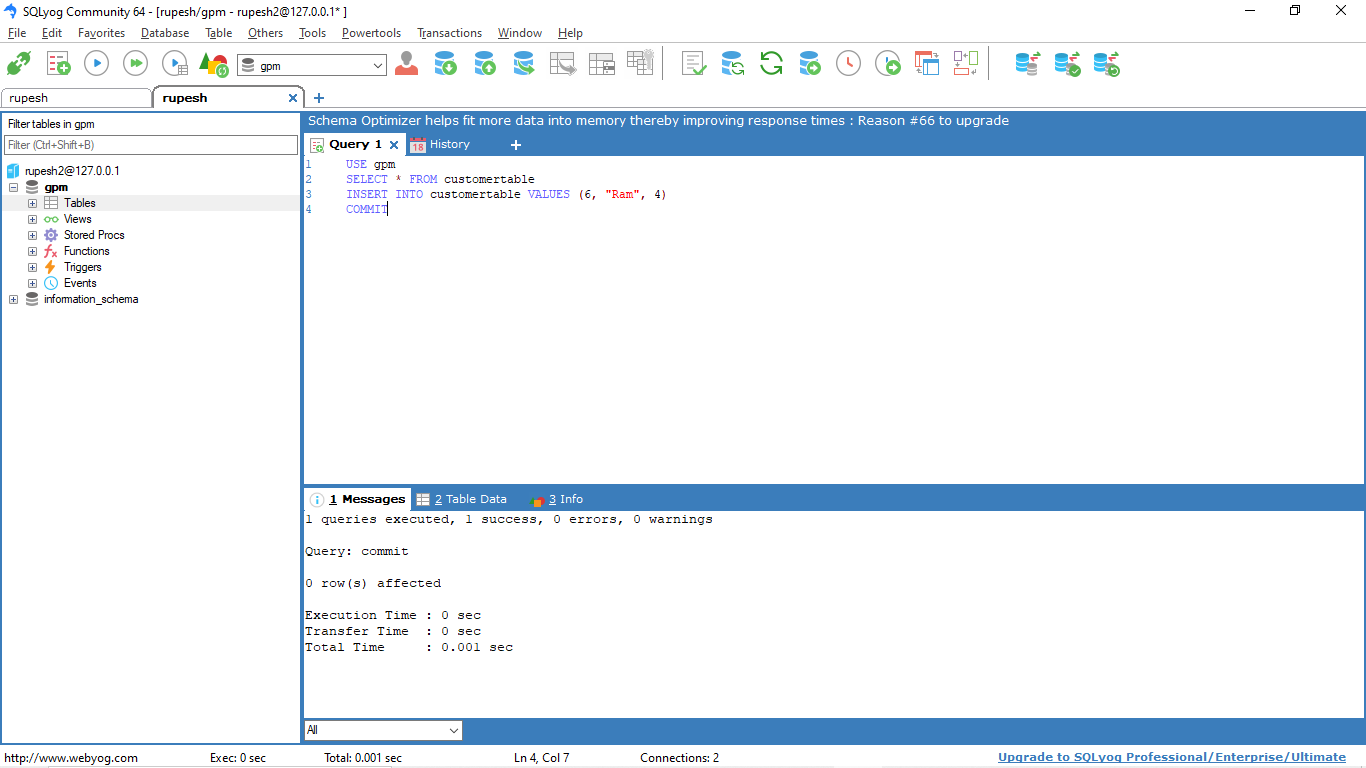
**2. REVOKE**

****

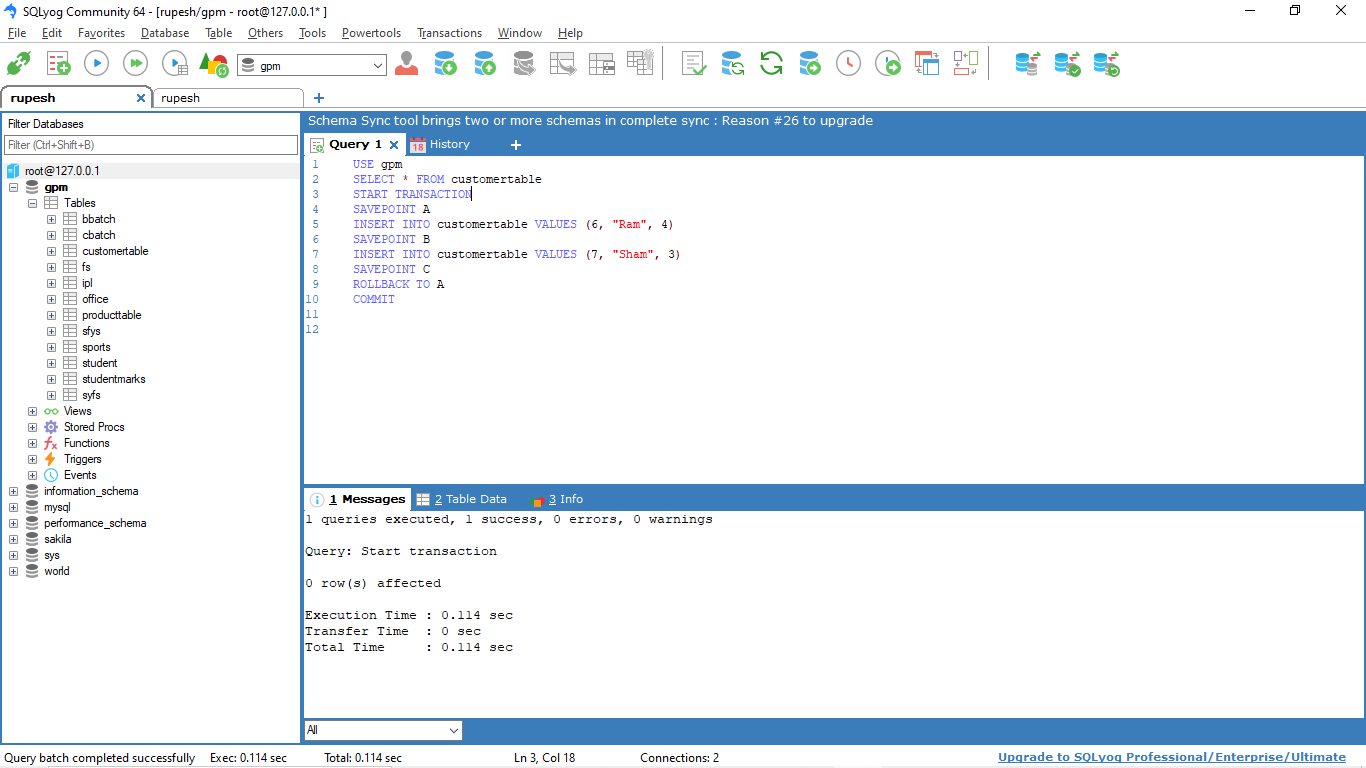
****

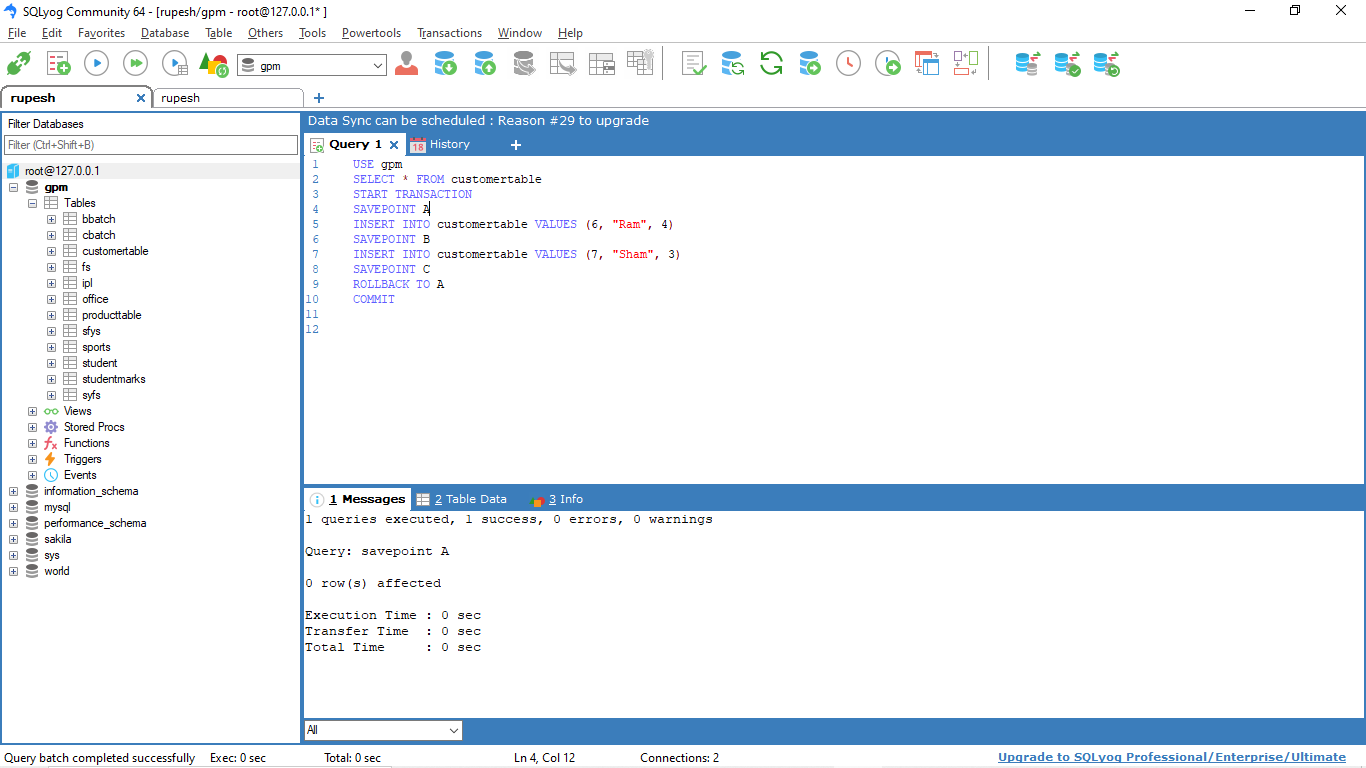
**2. TCL**

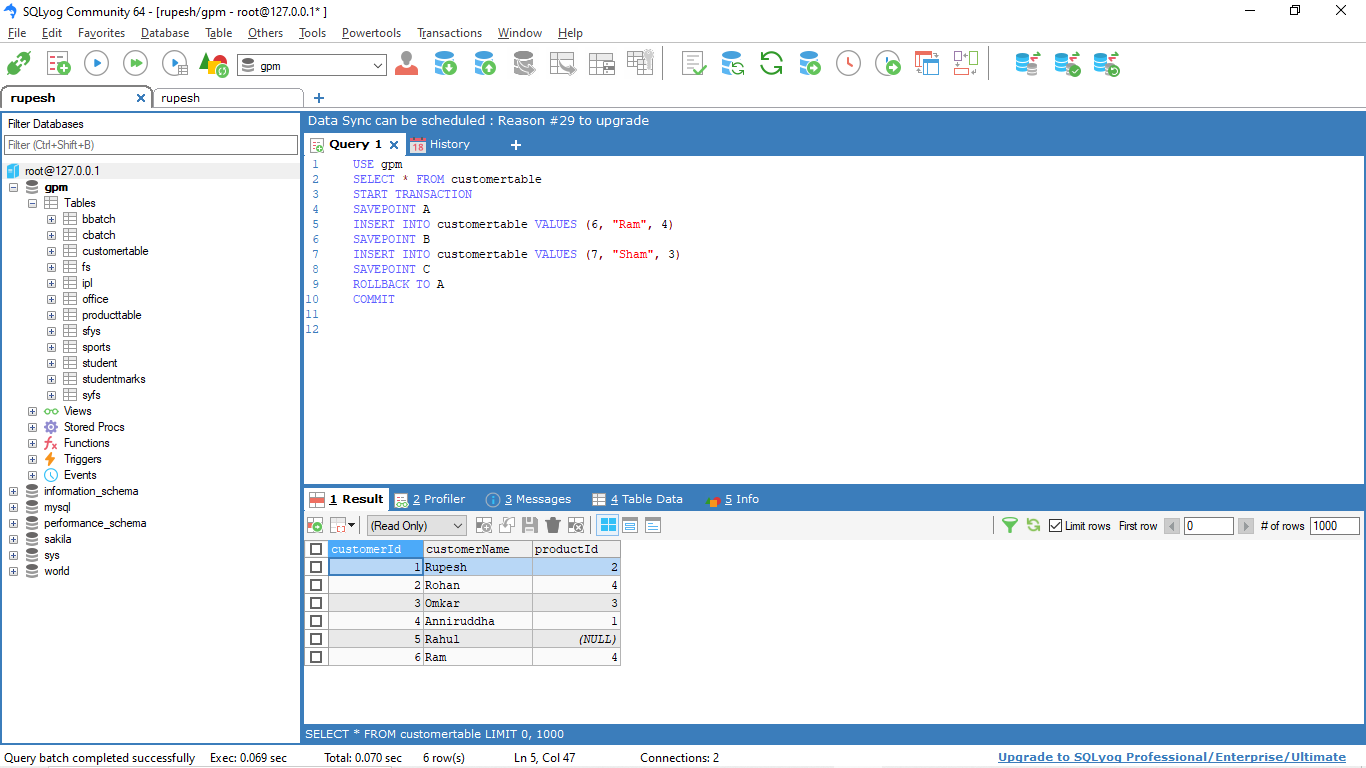
1. **COMMIT**

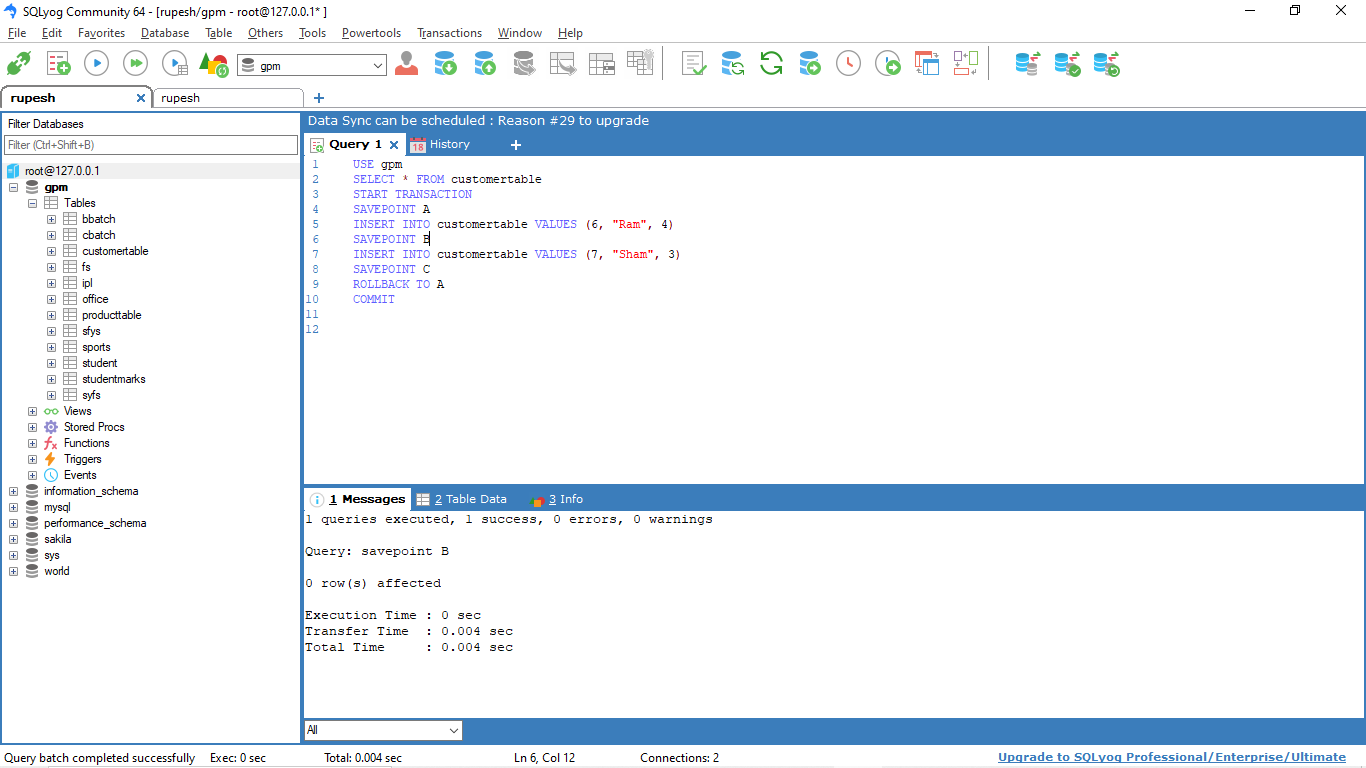
****

1. **SAVEPOINT**

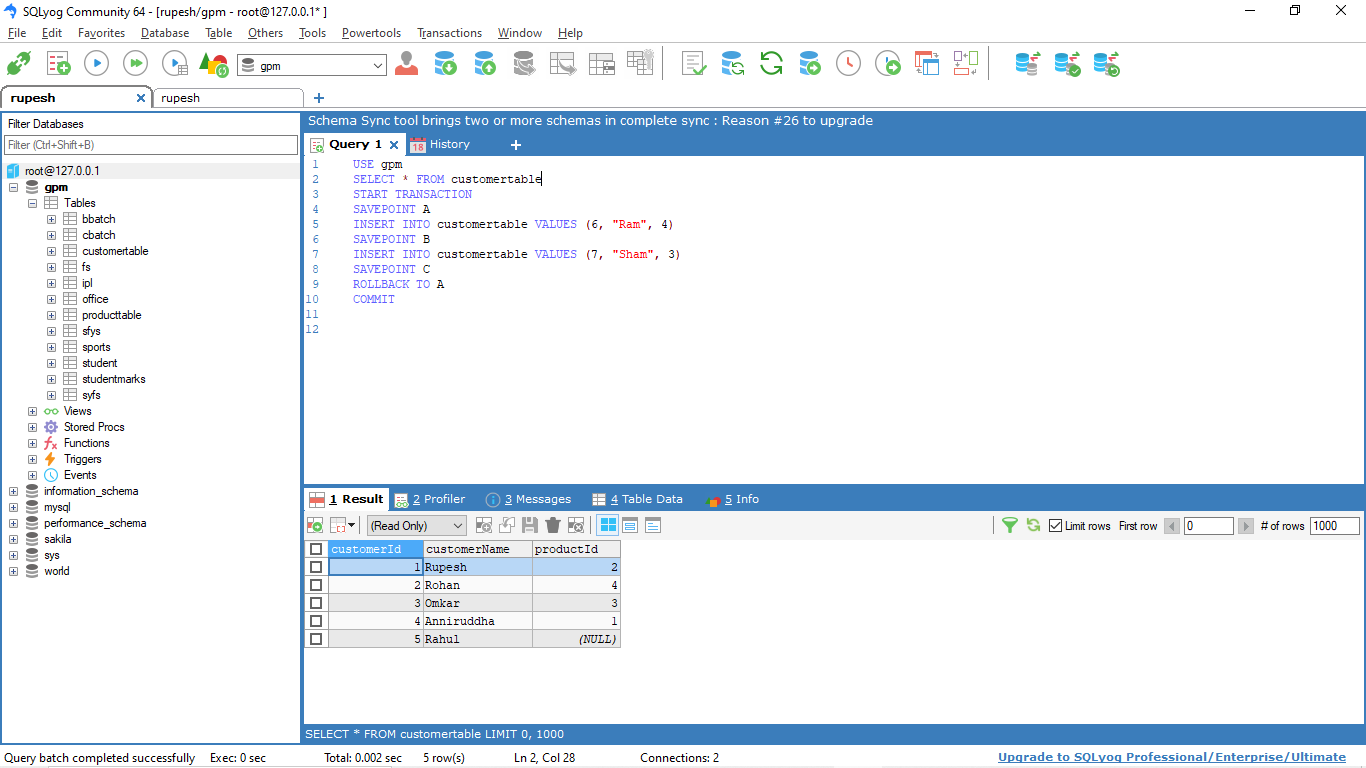
****

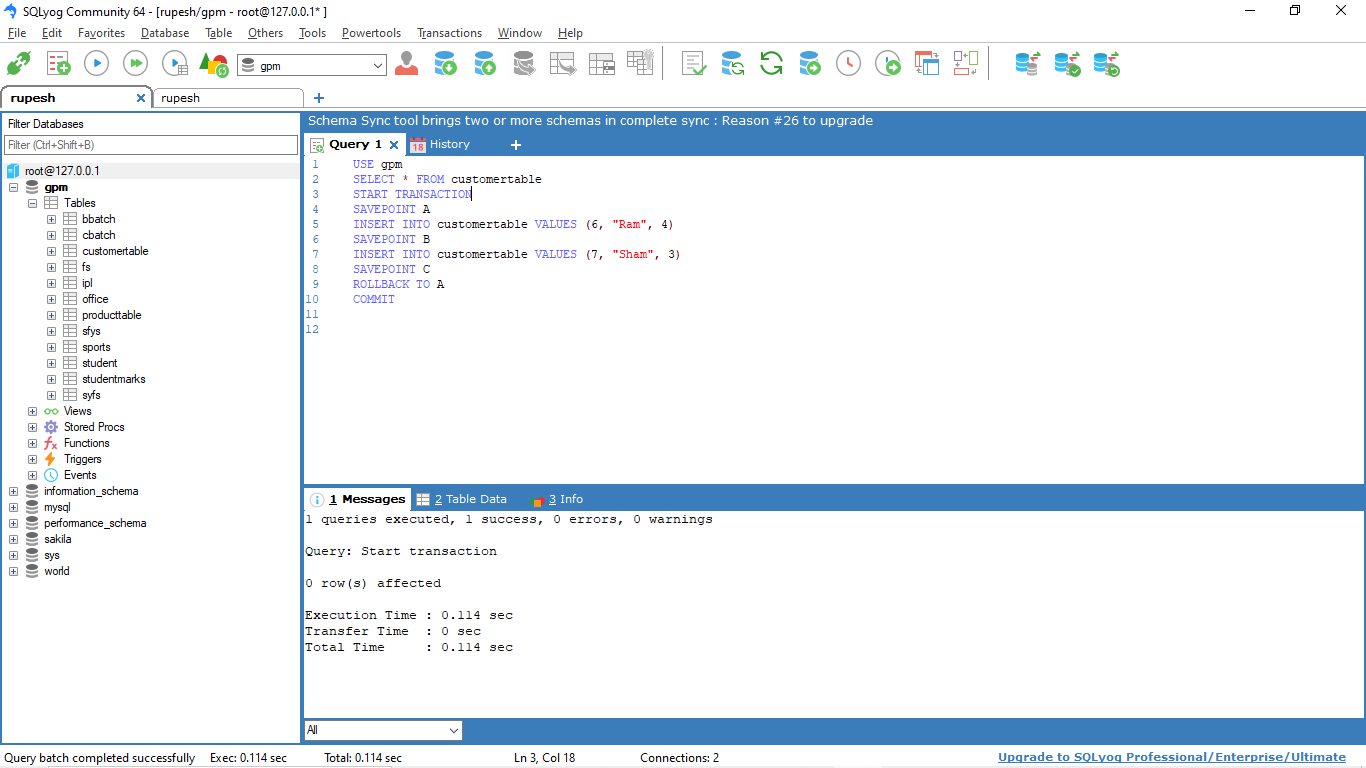
****

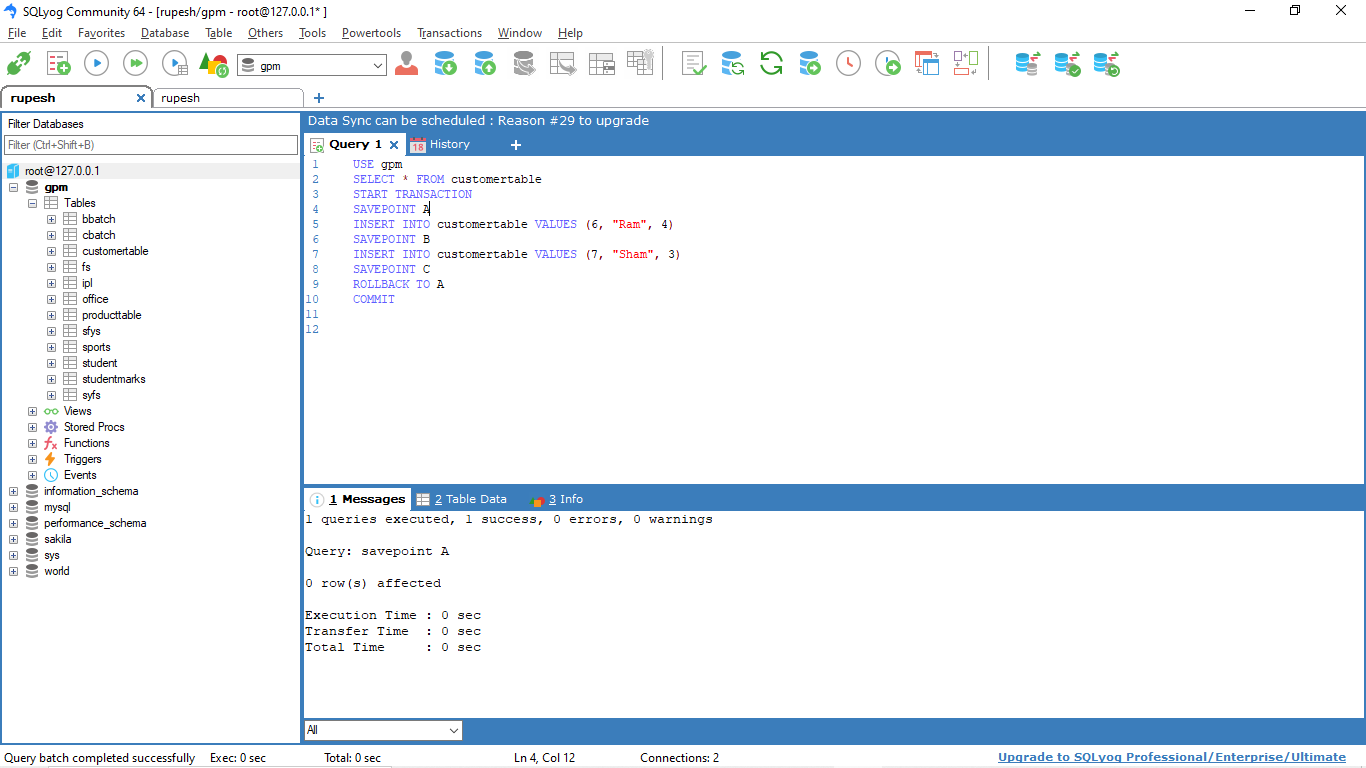
****

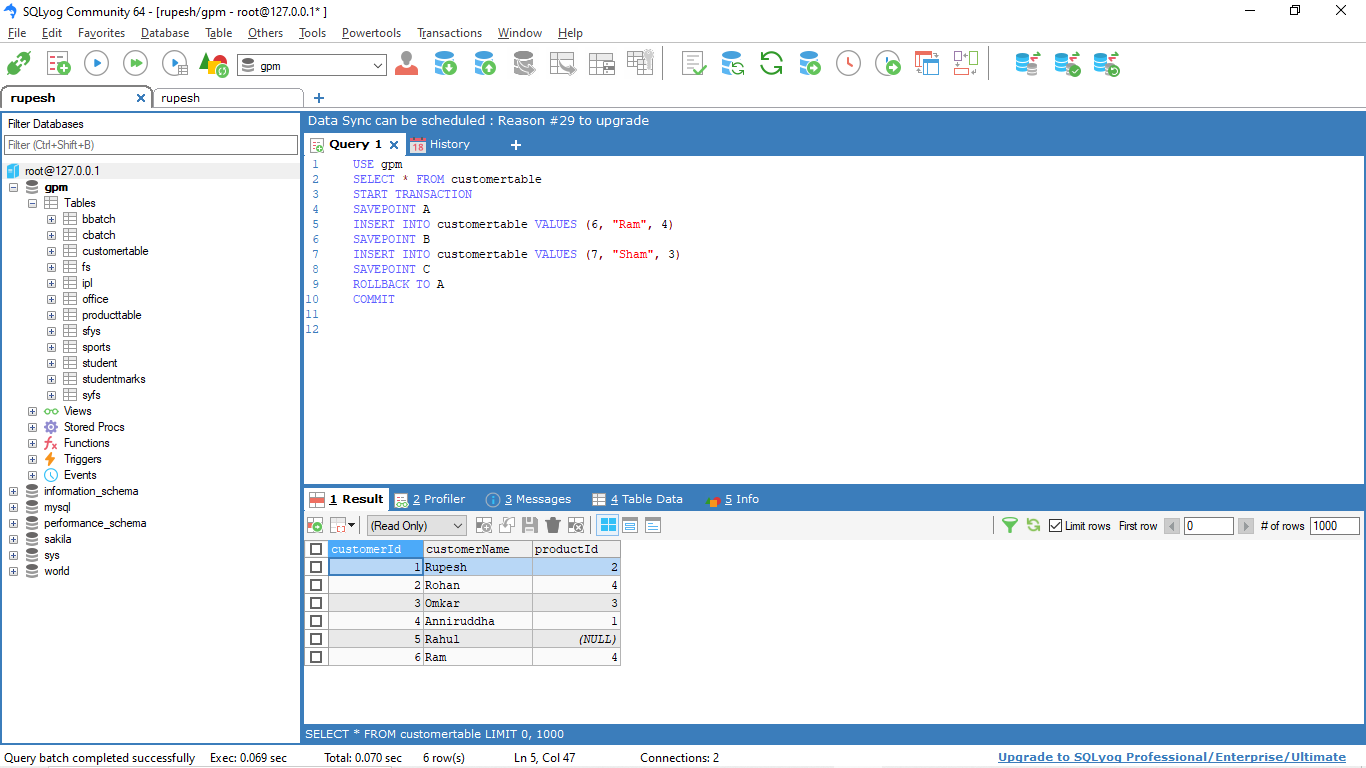
****

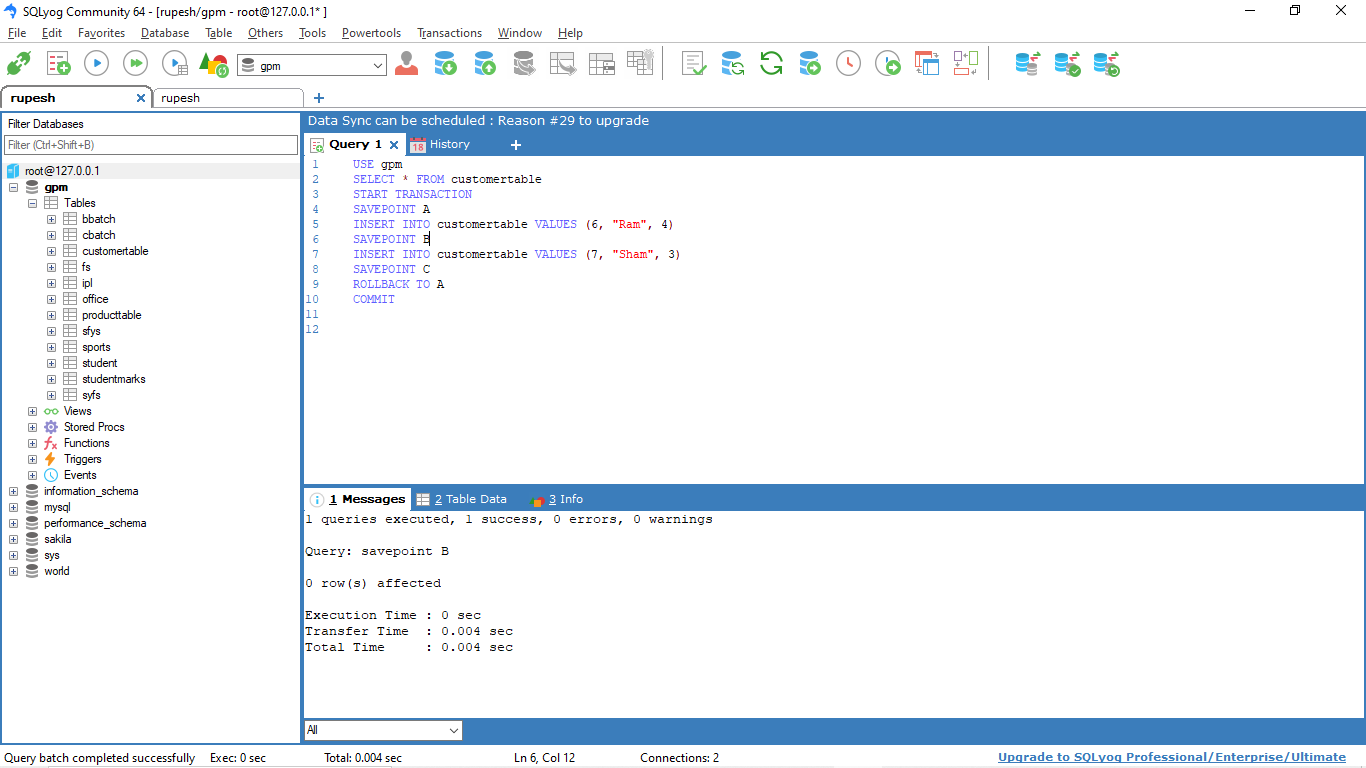
1. **ROLLBACK**

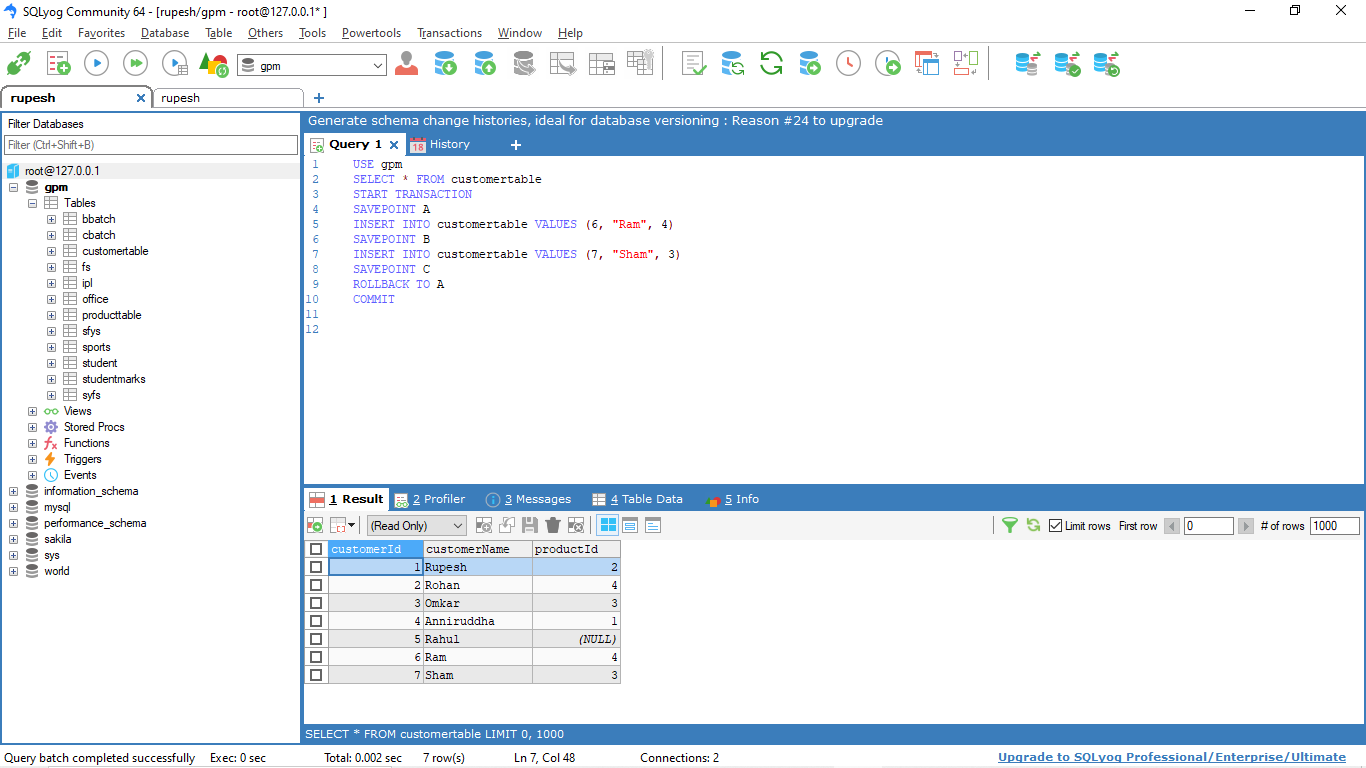
****

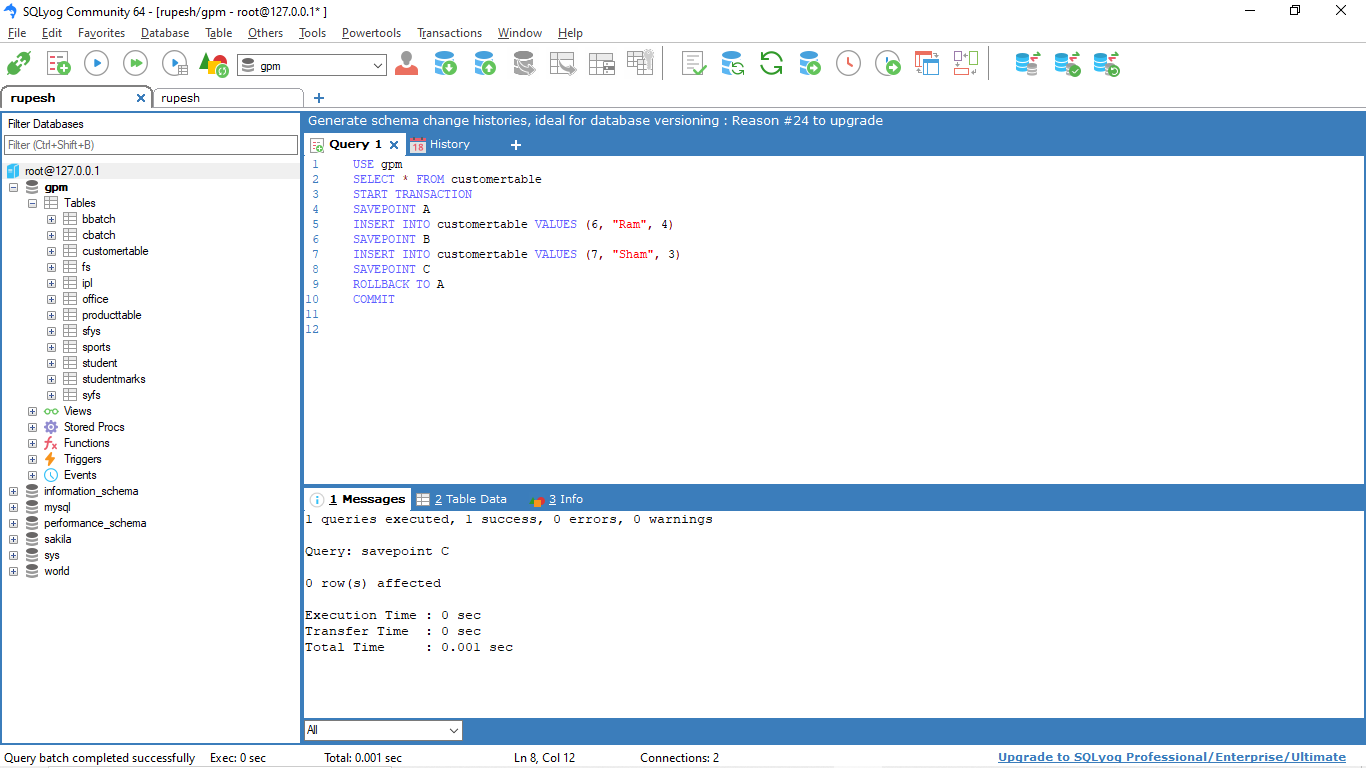
****

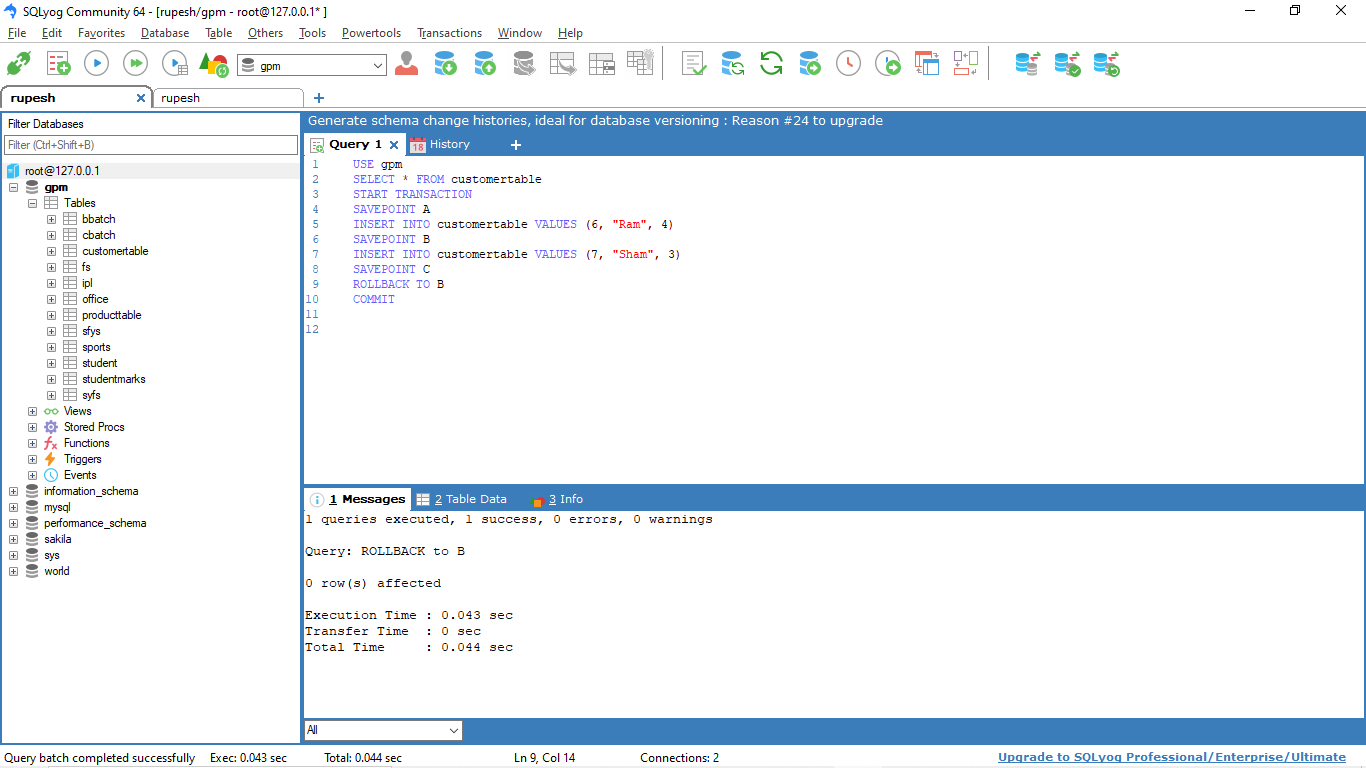
****

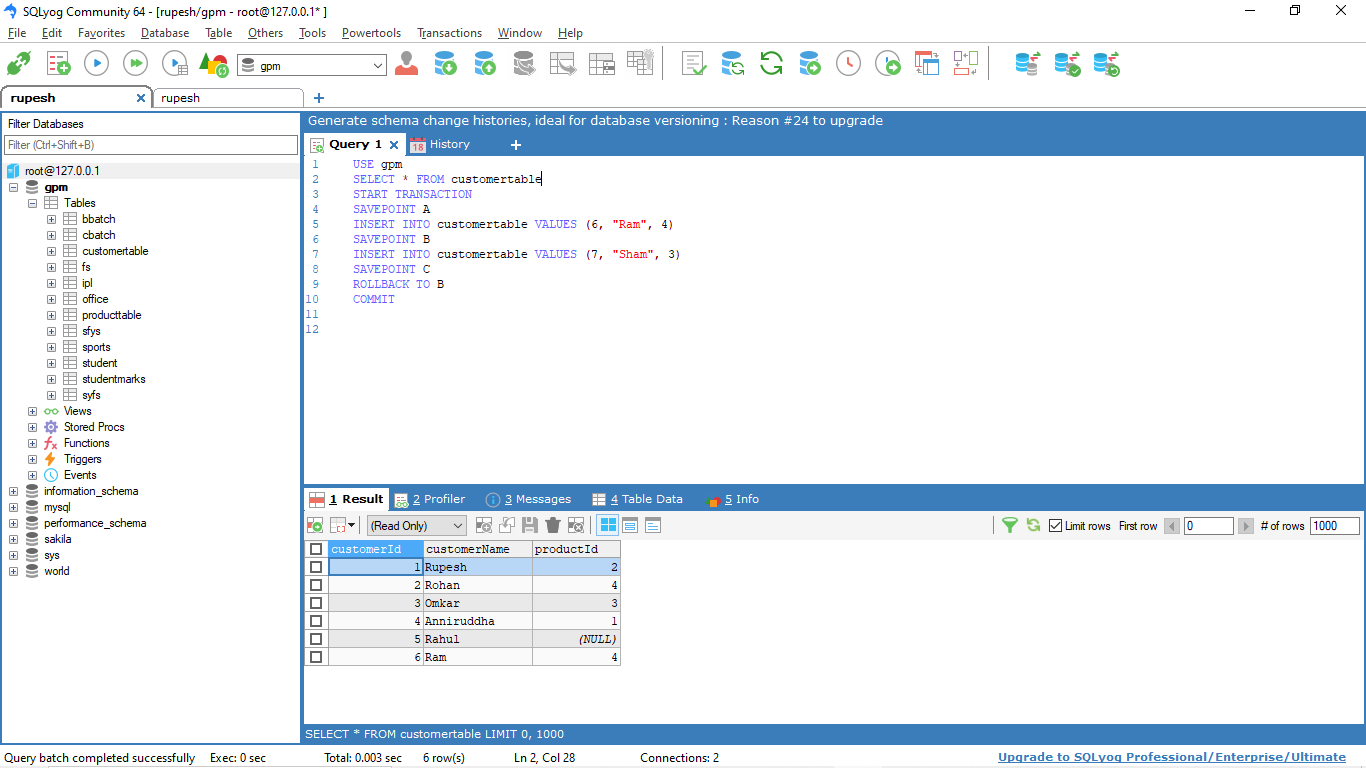
****

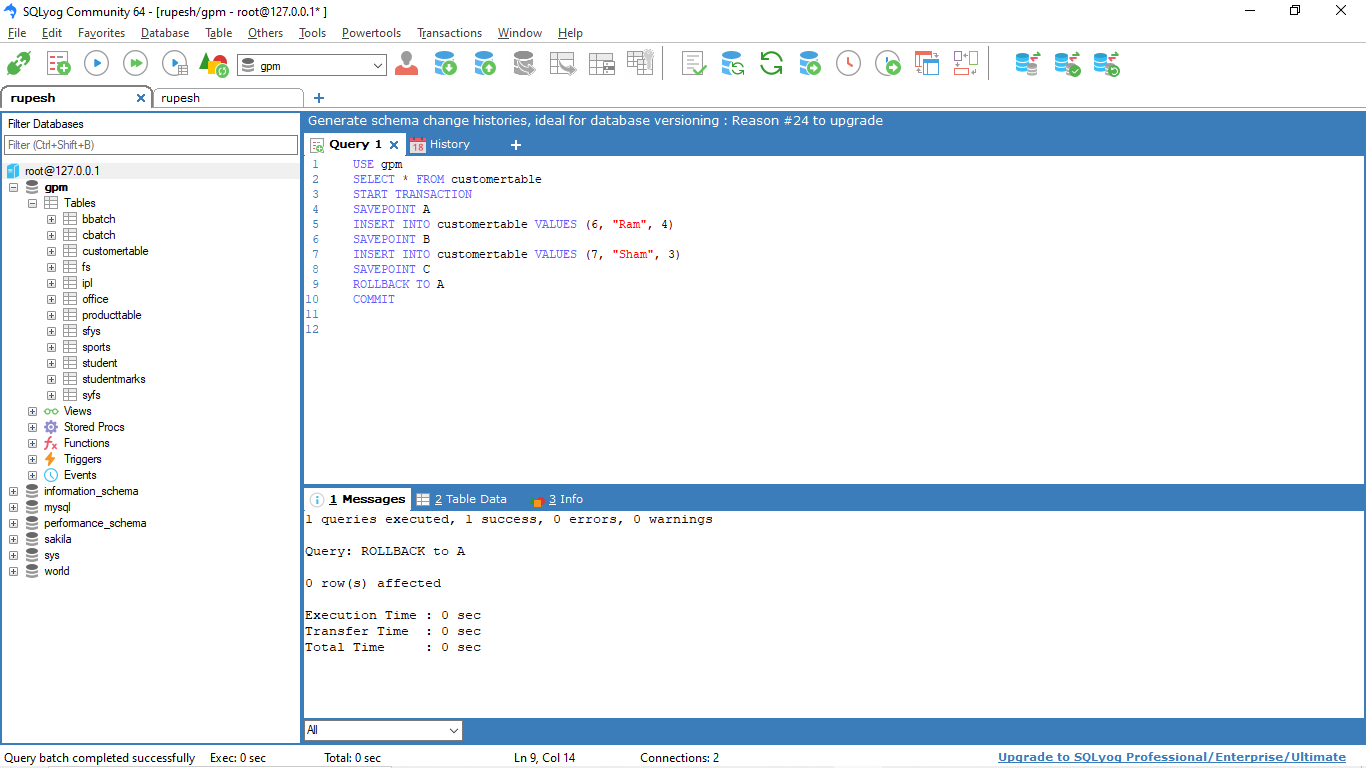
****

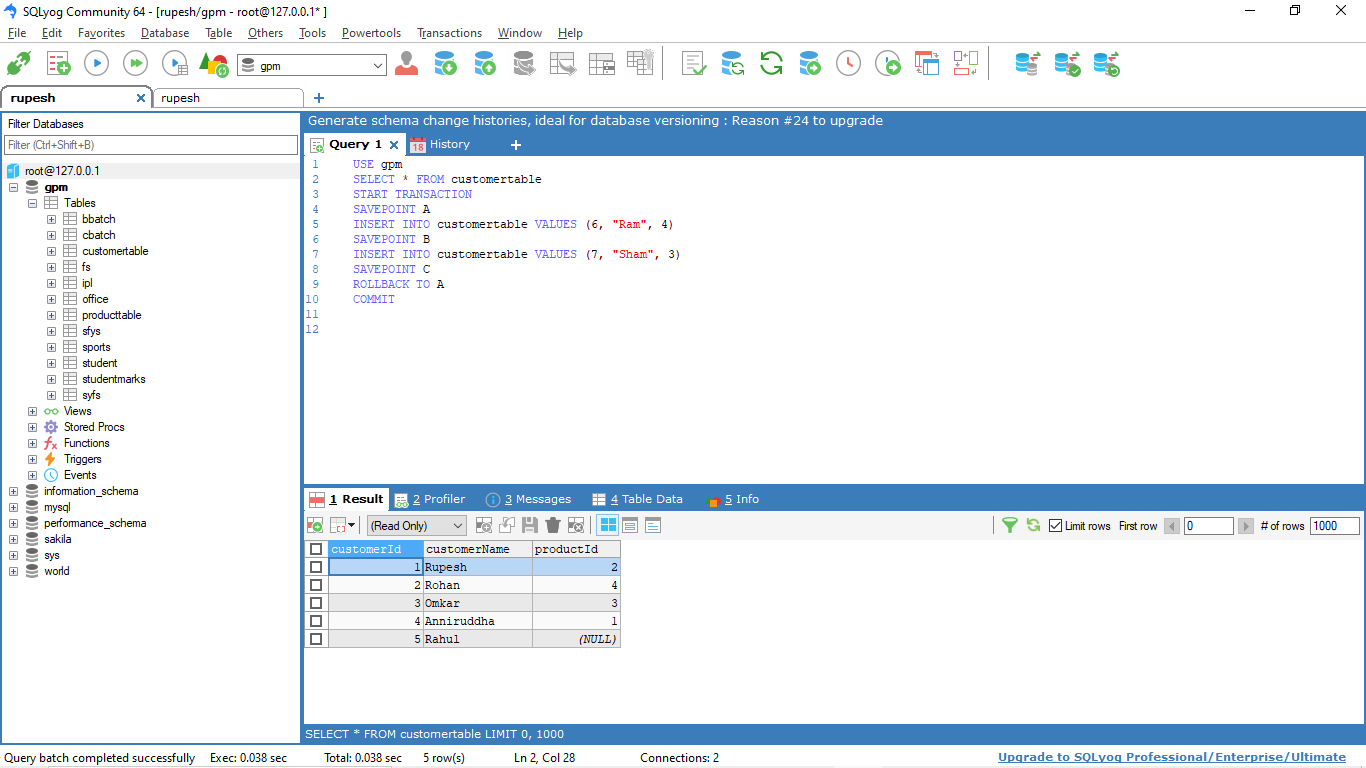
****

****

****

****

****

****

**Conclusion: Thus, we understood and implemented DCL AND TCL commands to manipulate users and permissions.**

FS19CO042

**Practical no. 9**

**Title:** Write a PL/SQL programs using if then else, for, while, nested loop

**Theory:**

1. **IF-THEN Statement**

It is the simplest form of the IF control statement, frequently used in decision-making and changing the control flow of the program execution.

The IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF. If the condition is TRUE, the statements get executed, and if the condition is FALSE or NULL, then the IF statement does nothing.

Syntax :

IF condition THEN

S;

END IF;

1. **IF-THEN-ELSE Statement**

A sequence of IF-THEN statements can be followed by an optional sequence of ELSE statements, which execute when the condition is FALSE.

Syntax :

IF condition THEN

S1;

ELSE

S2;

END IF;

1. **Basic Loop Statement**

Basic loop structure encloses sequence of statements in between the LOOP and END LOOP statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax :

LOOP

Sequence of statements;

END LOOP;

1. **FOR LOOP Statement**

A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax :

FOR counter IN initial\_value .. final\_value LOOP

sequence\_of\_statements;

END LOOP;

1. **WHILE LOOP Statement**

A WHILE LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

WHILE condition LOOP

sequence\_of\_statements

END LOOP;

1. **Nested Loops**

PL/SQL allows using one loop inside another loop. Following section shows a few examples to illustrate the concept.

Syntax :

LOOP

Sequence of statements1

LOOP

Sequence of statements2

END LOOP;

END LOOP;

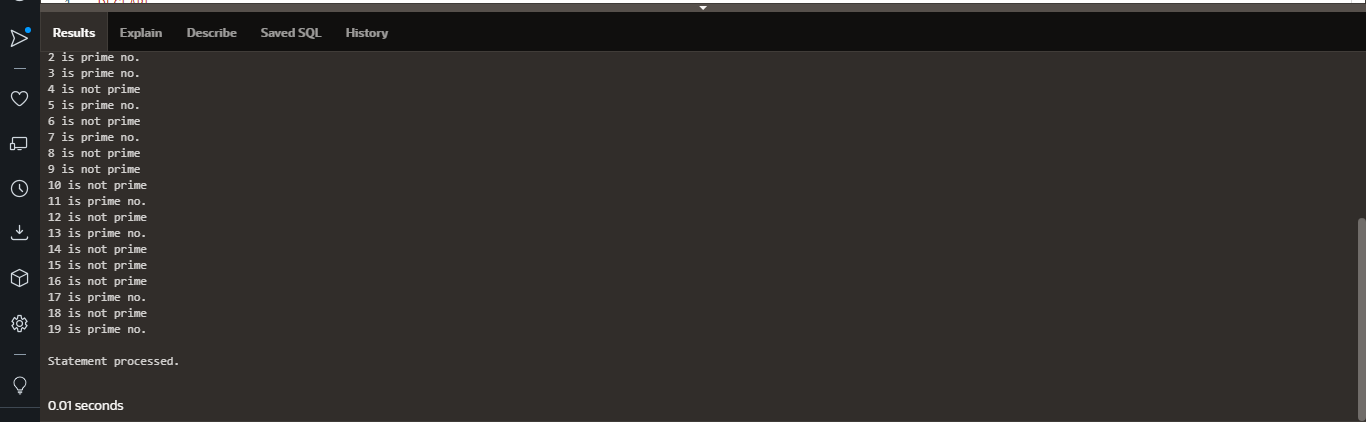
**Query and output:**

DECLARE

    a number(2) := 93;

    c number(2) := 4;

    i number(3) := 2;



    j number(3);

BEGIN

    -- If condition demo

    dbms\_output.put\_line('If condition demo:');

    IF(a >= 30) THEN

        dbms\_output.put\_line('A is greater than 30');

    ELSE

        dbms\_output.put\_line('A is smaller than 30');

    END IF;

    -- For loop demo

    dbms\_output.put\_line('');

    dbms\_output.put\_line('For loop demo: ');

    FOR i in 1..5 LOOP

        dbms\_output.put\_line('i: '||i);

    END LOOP;

    -- While loop demo

    dbms\_output.put\_line('');

    dbms\_output.put\_line('While loop demo: ');

    WHILE c>=0 LOOP

        dbms\_output.put\_line('c: '||c);

        c := c-1;

    END LOOP;

    -- Nested loop demo with example of prime nos

    dbms\_output.put\_line('');

    dbms\_output.put\_line('Nested loop demo, Prime nos from 2 to 20: ');

    LOOP

        j := 2;

        LOOP

            exit WHEN ((mod(i,j)=0) or (j=i));

            j := j+1;

        END LOOP;

    IF (j=i)   THEN

        dbms\_output.put\_line(i || ' is prime no.');

    ELSE

        dbms\_output.put\_line(i || ' is not prime');

    END IF;

    i := i+1;

    exit WHEN i=20;

    END LOOP;

END;

Conclusion: Thus, we implemented PL/SQL programs using if then else, for, while, nested loop

**Practical no. 10**

FS19CO042

**Title:** Write a PL/SQL code to implement implicit and explicit cursors.

**Theory:**

1. **Cursor :**

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

1. **Implicit Cursor :**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK\_ROWCOUNT and %BULK\_EXCEPTIONS, designed for use with the FORALL statement.

1. **Explicit Cursor**

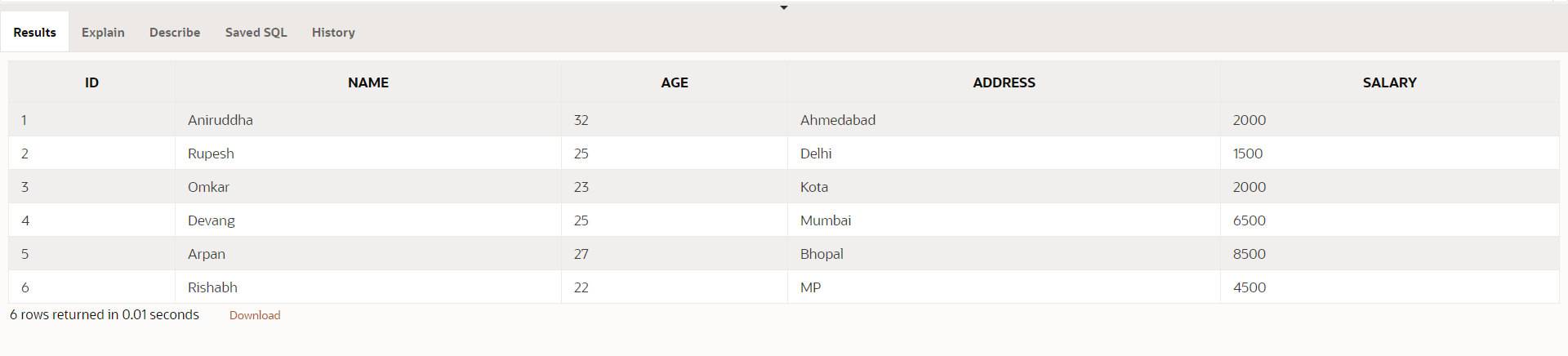
Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

Syntax :

CURSOR cursor\_name IS select\_statement;

**Output :**

1. **Implicit Cursor :**

CUSTOMERS table :

The following program will update the table and increase the salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected

DECLARE

   total\_rows number(2);

BEGIN

   UPDATE customers

   SET salary = salary + 500;

   IF sql%notfound THEN

      dbms\_output.put\_line('no customers selected');

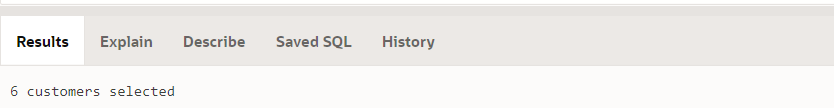
   ELSIF sql%found THEN

      total\_rows := sql%rowcount;

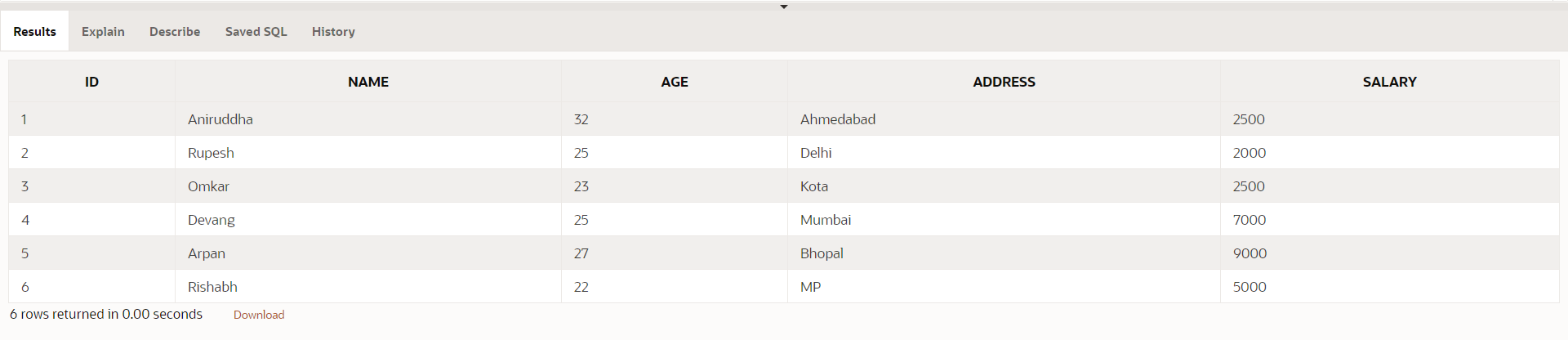
      dbms\_output.put\_line( total\_rows || ' customers selected ');

   END IF;

END;

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

If you check the records in customers table, you will find that the rows have been updated

Customers Table :

1. **Explicit Cursor**

Following is a complete example to illustrate the concepts of explicit cursors

DECLARE

   c\_id customers.id%type;

   c\_name customers.name%type;

   c\_addr customers.address%type;

   CURSOR c\_customers is

      SELECT id, name, address FROM customers;

BEGIN

   OPEN c\_customers;

   LOOP

   FETCH c\_customers into c\_id, c\_name, c\_addr;

      EXIT WHEN c\_customers%notfound;

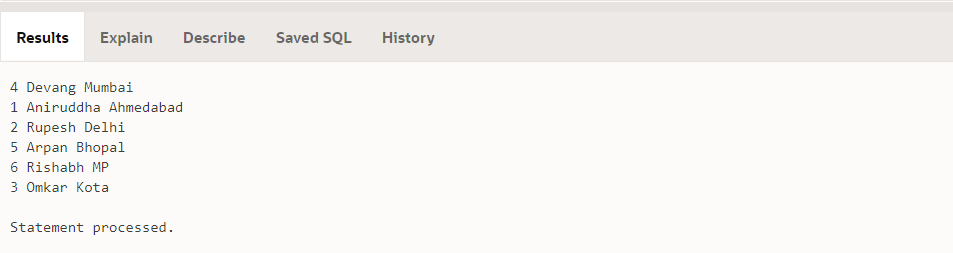
      dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);

   END LOOP;

   CLOSE c\_customers;

END;

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



**Conclusion: Thus we understood and implemented implicit and explicit cursors.**

FS19CO042

**Practical no. 11**

**Title: Write a PL/SQL code to create procedure and function.**

**Theory:**

**A subprogram created inside a package is a packaged subprogram. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.**

**PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −**

1. **Functions :**

**These subprograms return a single value; mainly used to compute and return a value.**

1. **Procedures :**

**These subprograms do not return a value directly; mainly used to perform an action.**

1. **Procedure**

**A procedure is created with the CREATE OR REPLACE PROCEDURE statement. 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.**

**Three ways to pass parameters in procedure:**

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

**The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows :**

**CREATE [OR REPLACE] PROCEDURE procedure\_name**

**[(parameter\_name [IN | OUT | IN OUT] type [, ...])]**

**{IS | AS}**

**BEGIN**

**< procedure\_body >**

**END procedure\_name;**

1. **Function**

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

**The function must contain a return statement.**

**RETURN clause specifies that data type you are going to return from the function.**

**Function\_body contains the executable part.**

**The AS keyword is used instead of the IS keyword for creating a standalone function**

**The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −**

**CREATE [OR REPLACE] FUNCTION function\_name**

**[(parameter\_name [IN | OUT | IN OUT] type [, ...])]**

**RETURN return\_datatype**

**{IS | AS}**

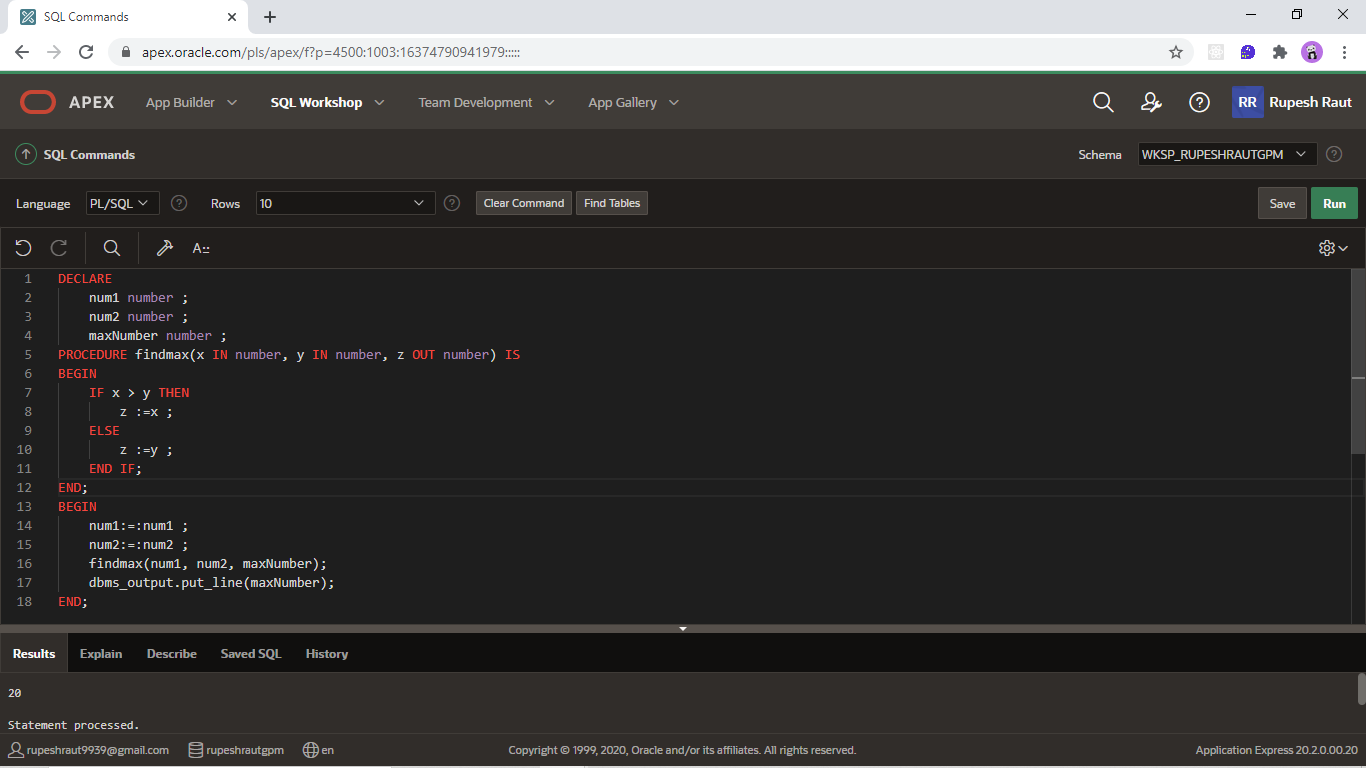
**BEGIN**

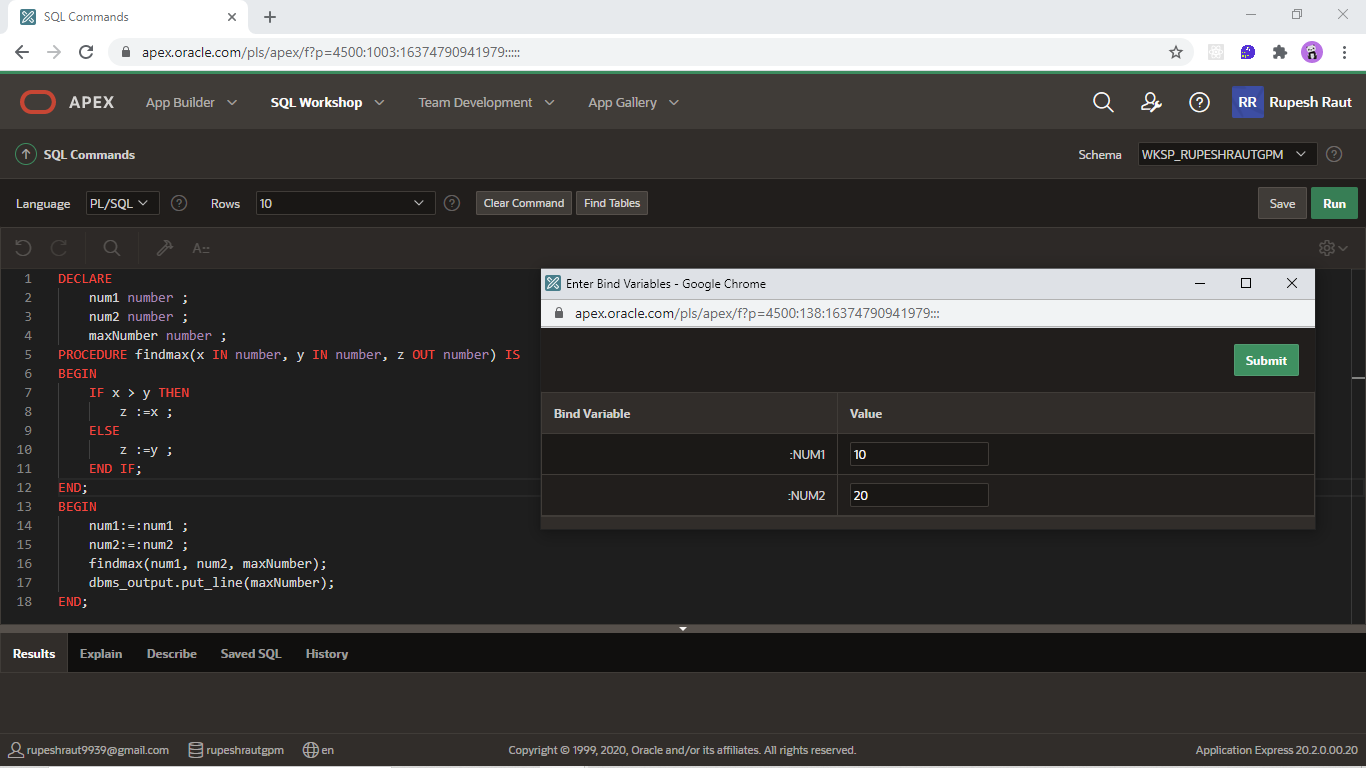
**< function\_body >**

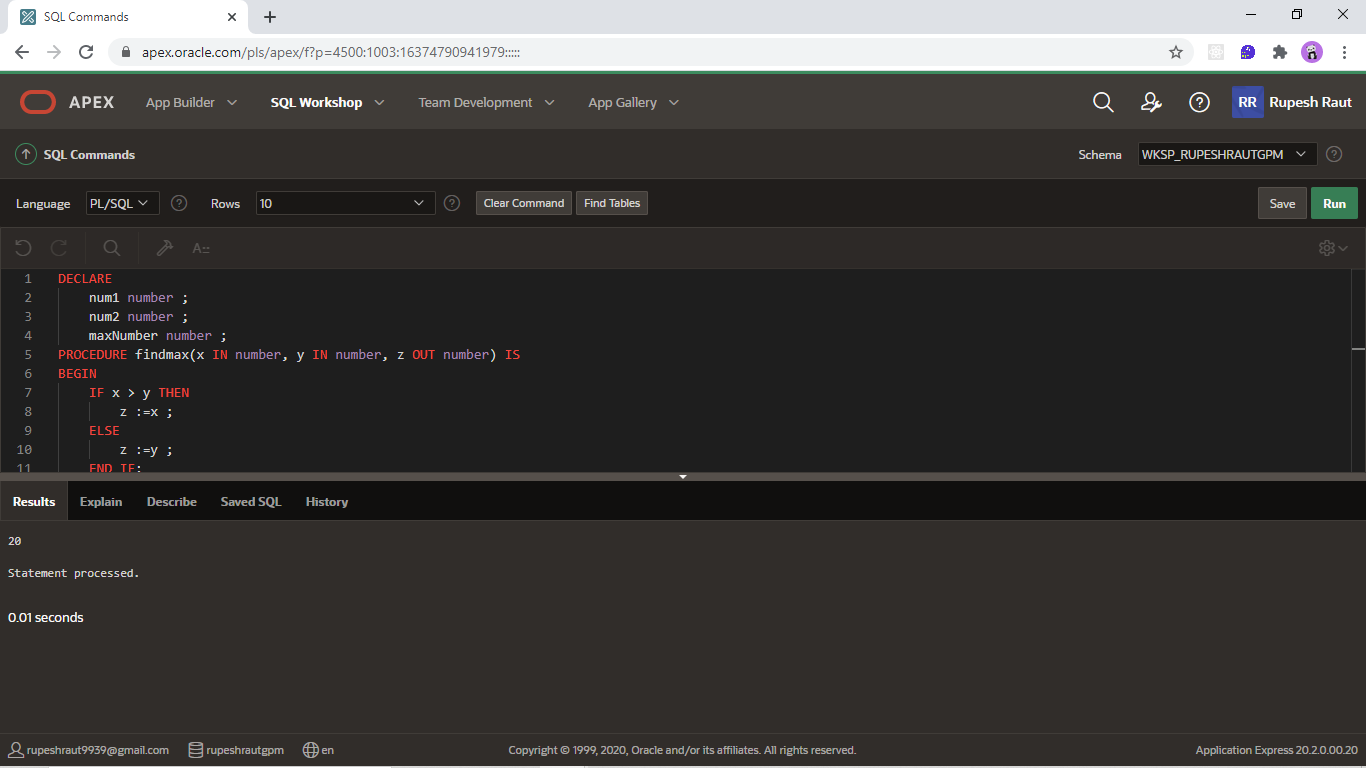
**END [function\_name];**

**Output:**

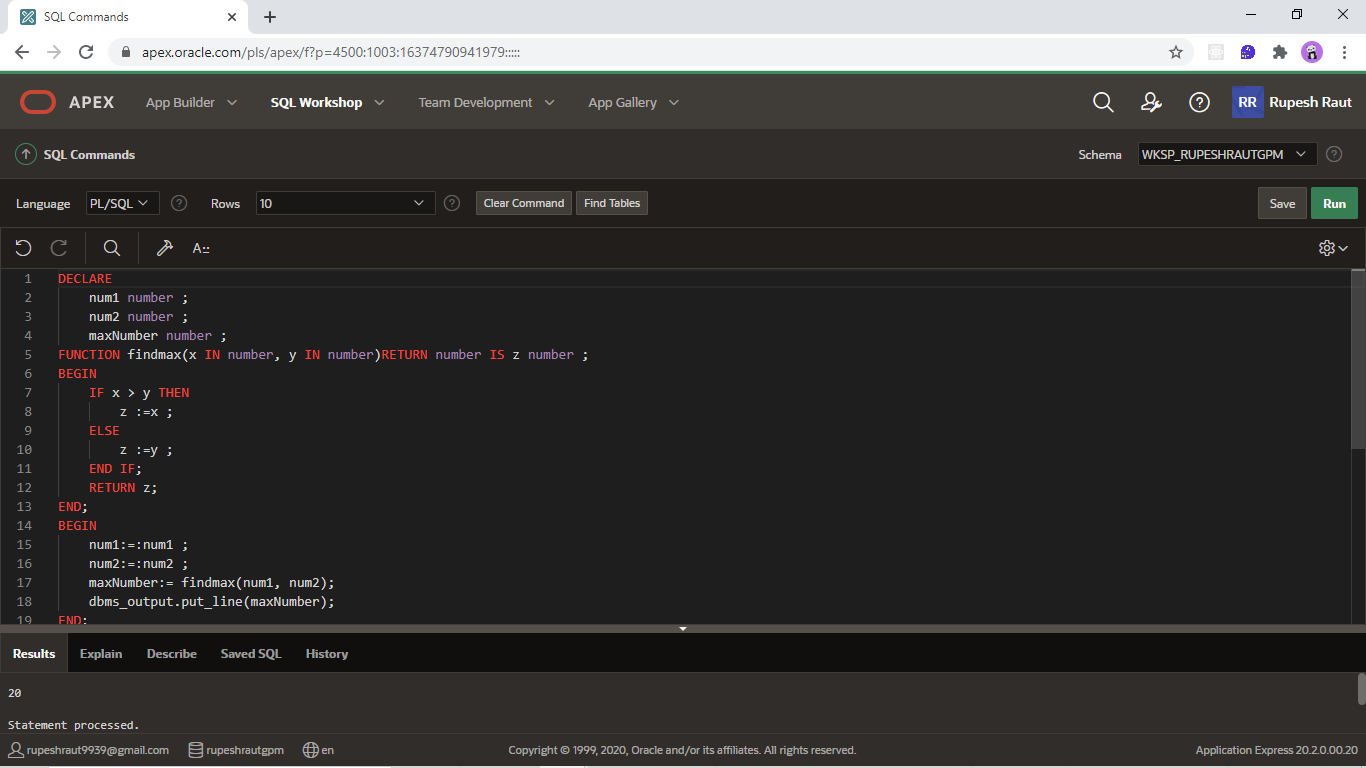
1. **Procedure :**

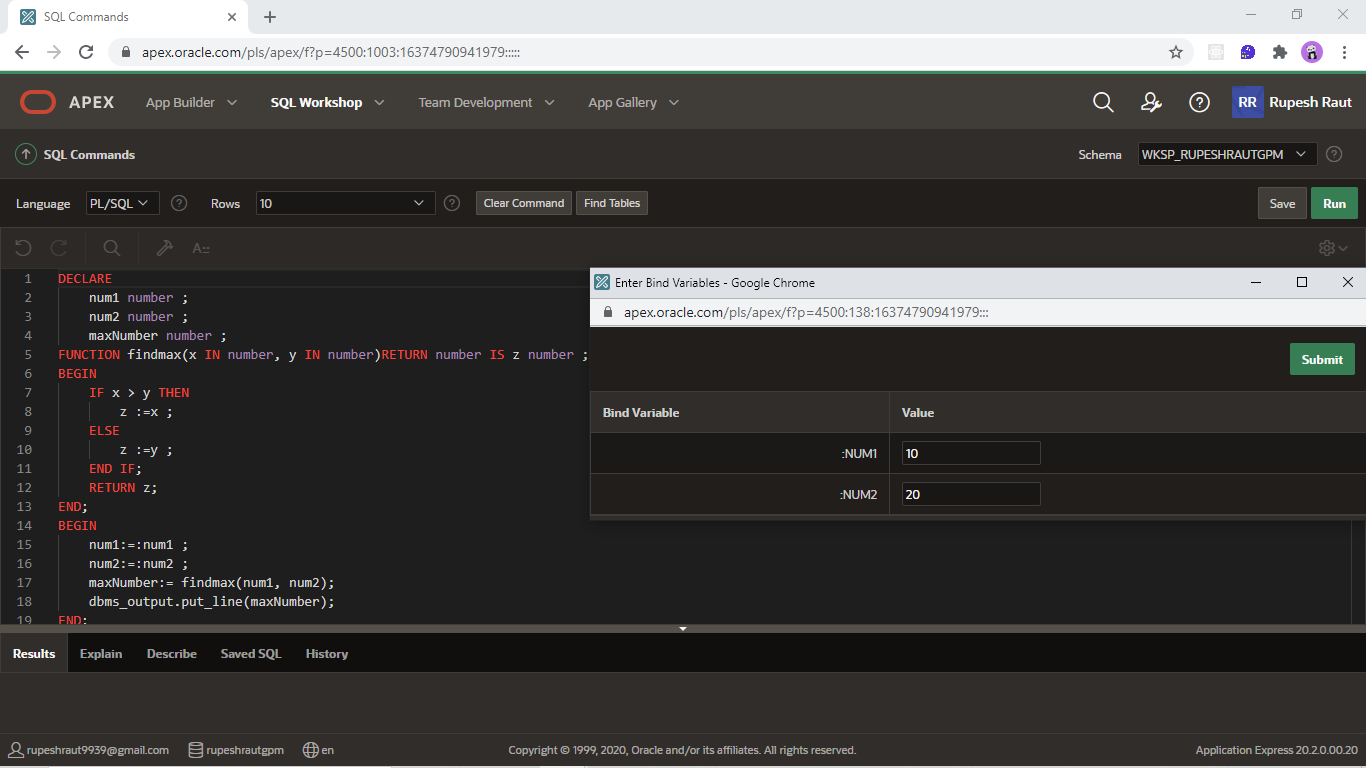
****

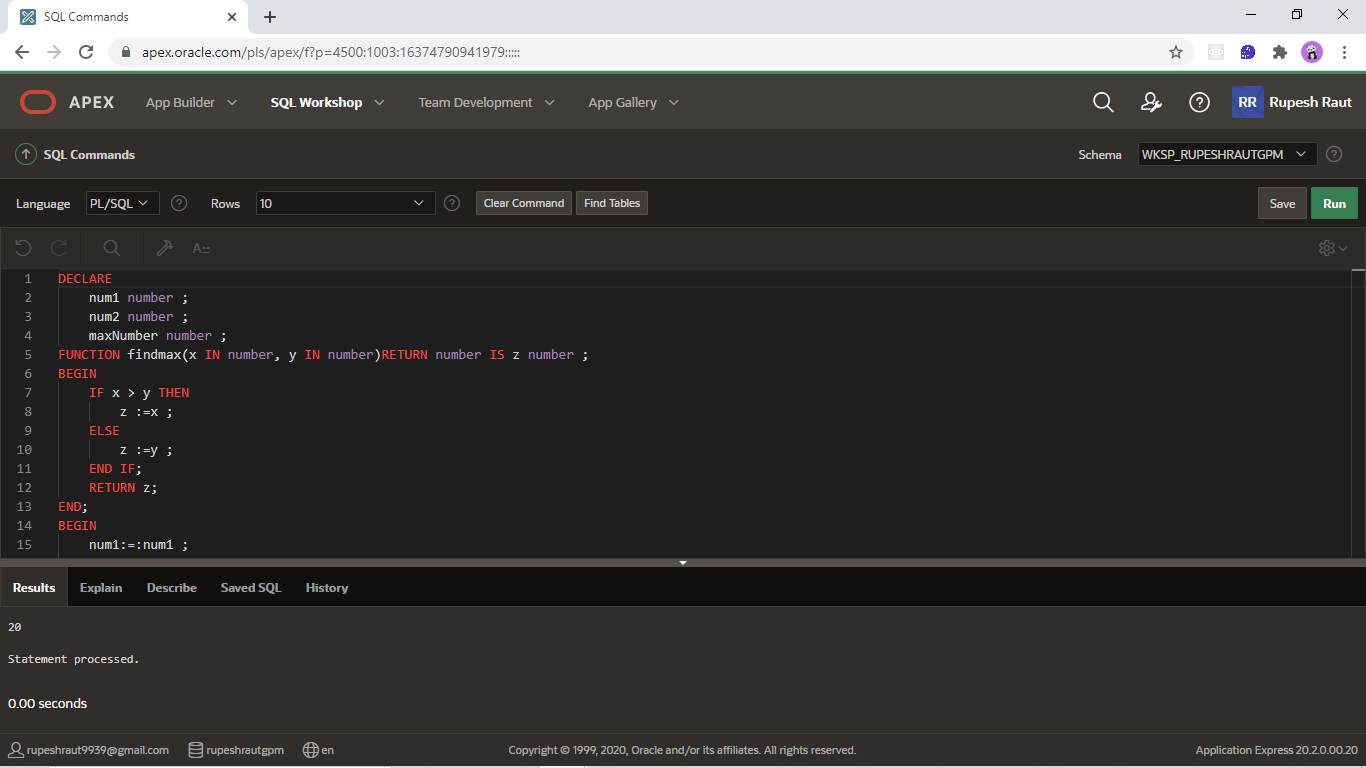
****

****

1. **Function :**

****

****

****

**Conclusion: Thus, we defined and used procedures and functions in PL/SQL.**

**Practical no. 12**

FS19CO042

**Title:** **Write a PL/SQL code to create triggers on given database**

**Theory:**

**Triggers :**

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

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

**Benefits of Triggers**

**Triggers can be written for the following purposes −**

1. **Generating some derived column values automatically**
2. **Enforcing referential integrity**
3. **Event logging and storing information on table access**
4. **Auditing**
5. **Synchronous replication of tables**
6. **Imposing security authorizations**
7. **Preventing invalid transactions**

**Syntax :**

**CREATE [OR REPLACE ] TRIGGER trigger\_name**

**{BEFORE | AFTER | INSTEAD OF }**

**{INSERT [OR] | UPDATE [OR] | DELETE}**

**[OF col\_name]**

**ON table\_name**

**[REFERENCING OLD AS o NEW AS n]**

**[FOR EACH ROW]**

**WHEN (condition)**

**DECLARE**

**Declaration-statements**

**BEGIN**

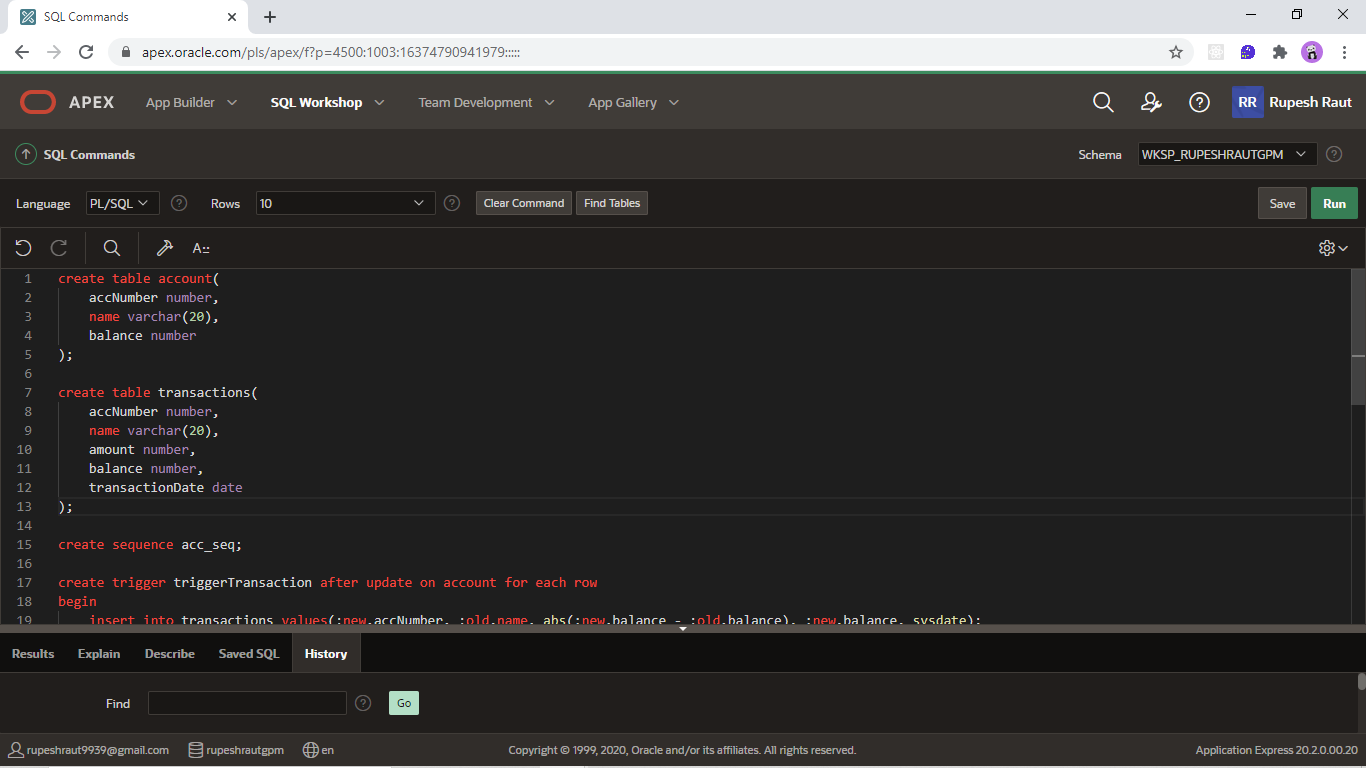
**Executable-statements**

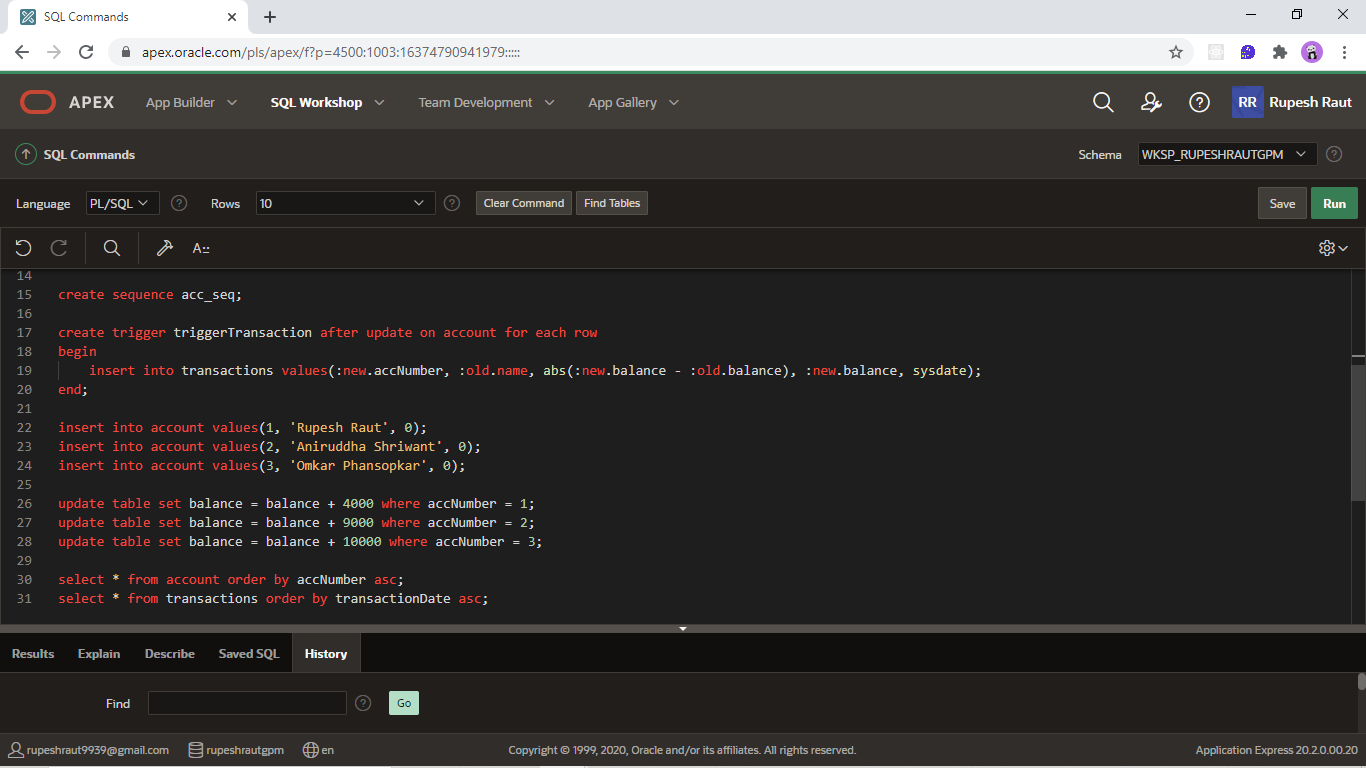
**EXCEPTION**

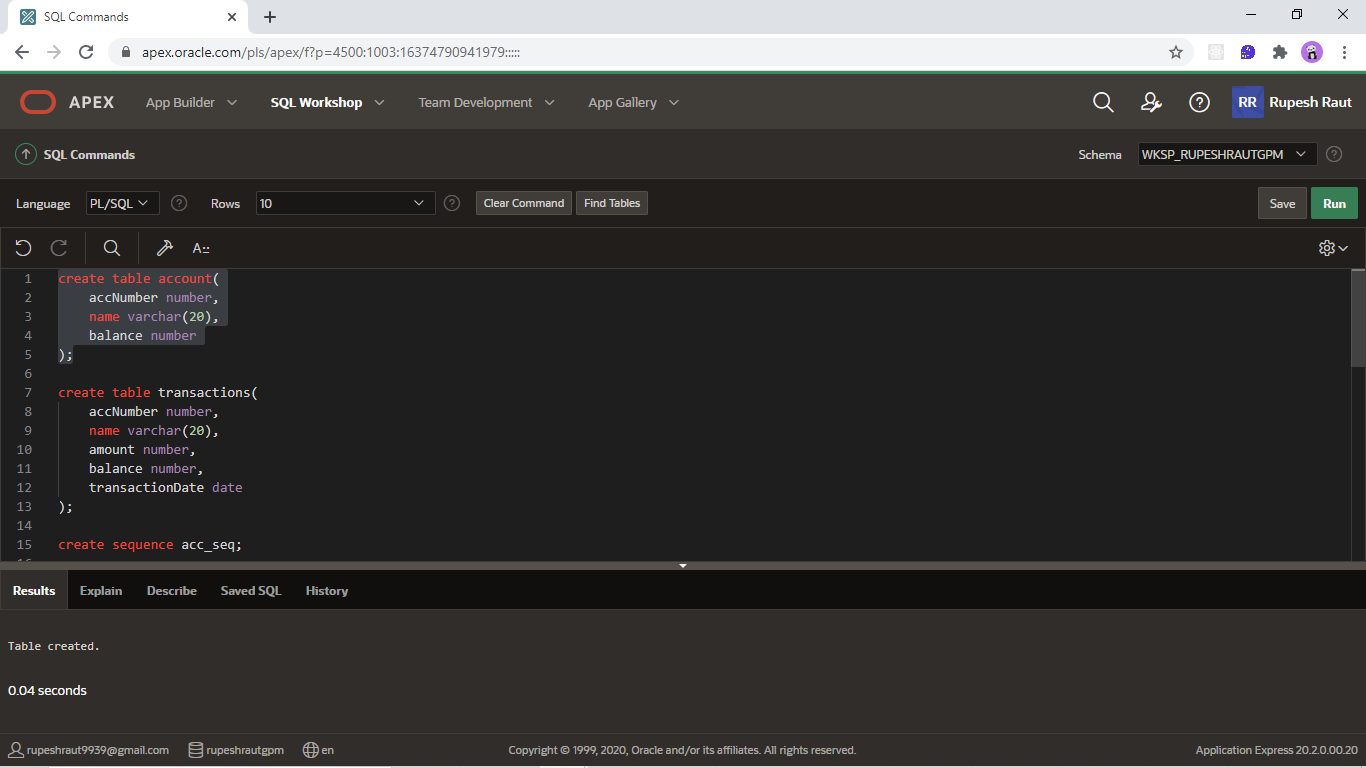
**Exception-handling-statements**

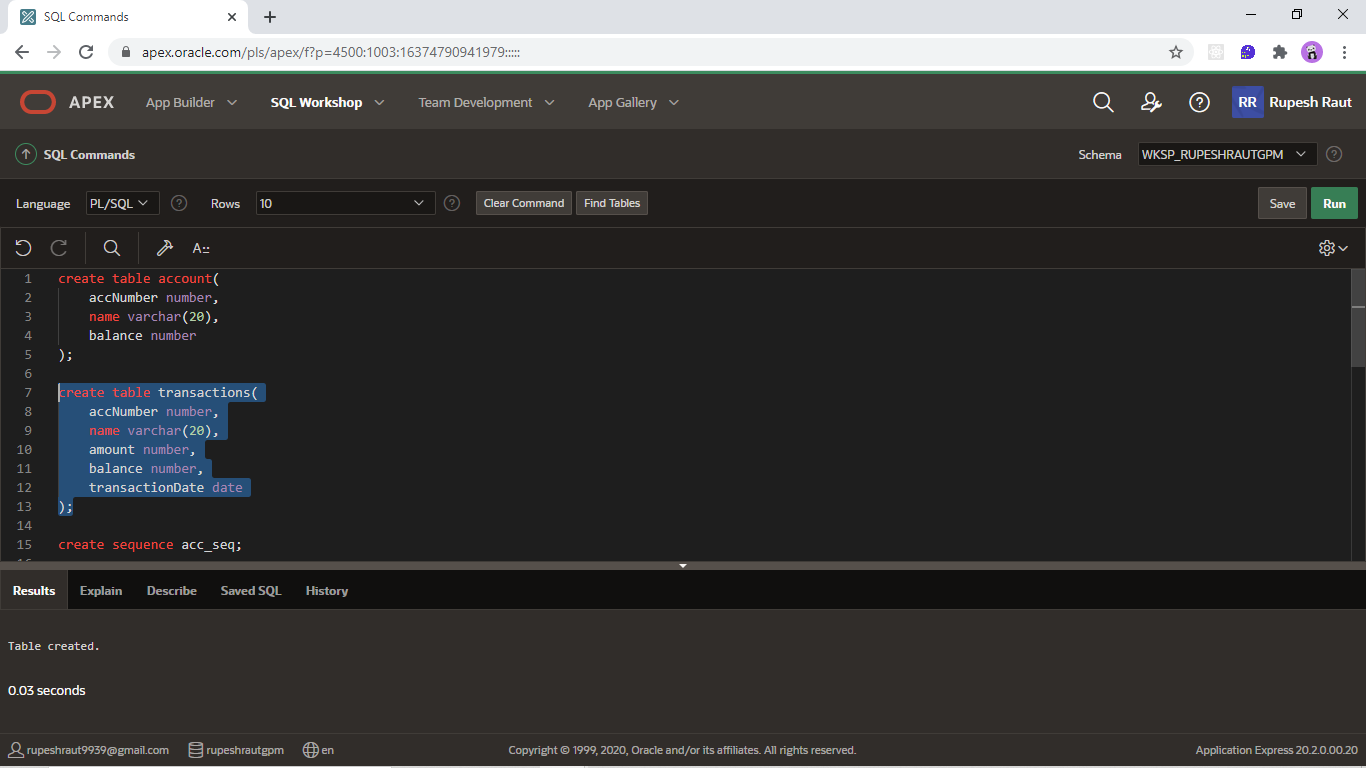
**END;**

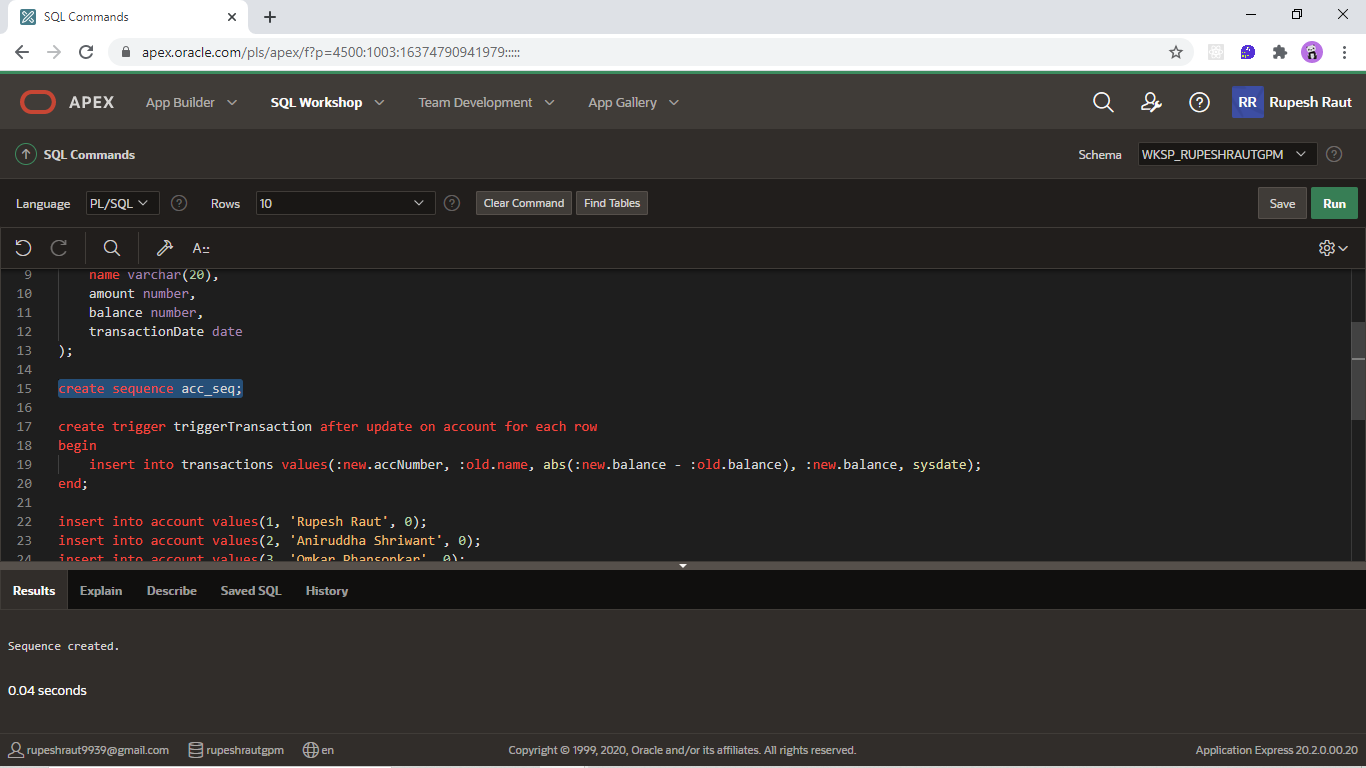
**Output :**

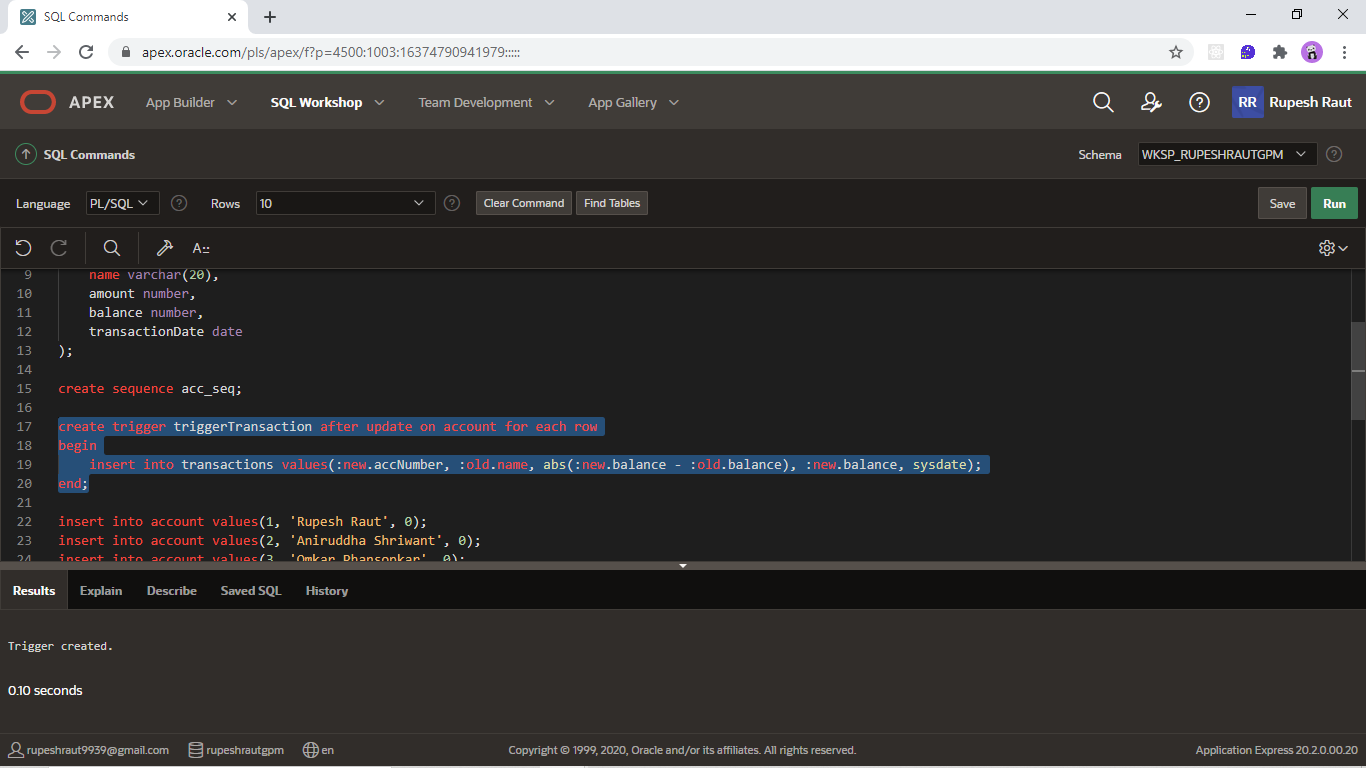
****

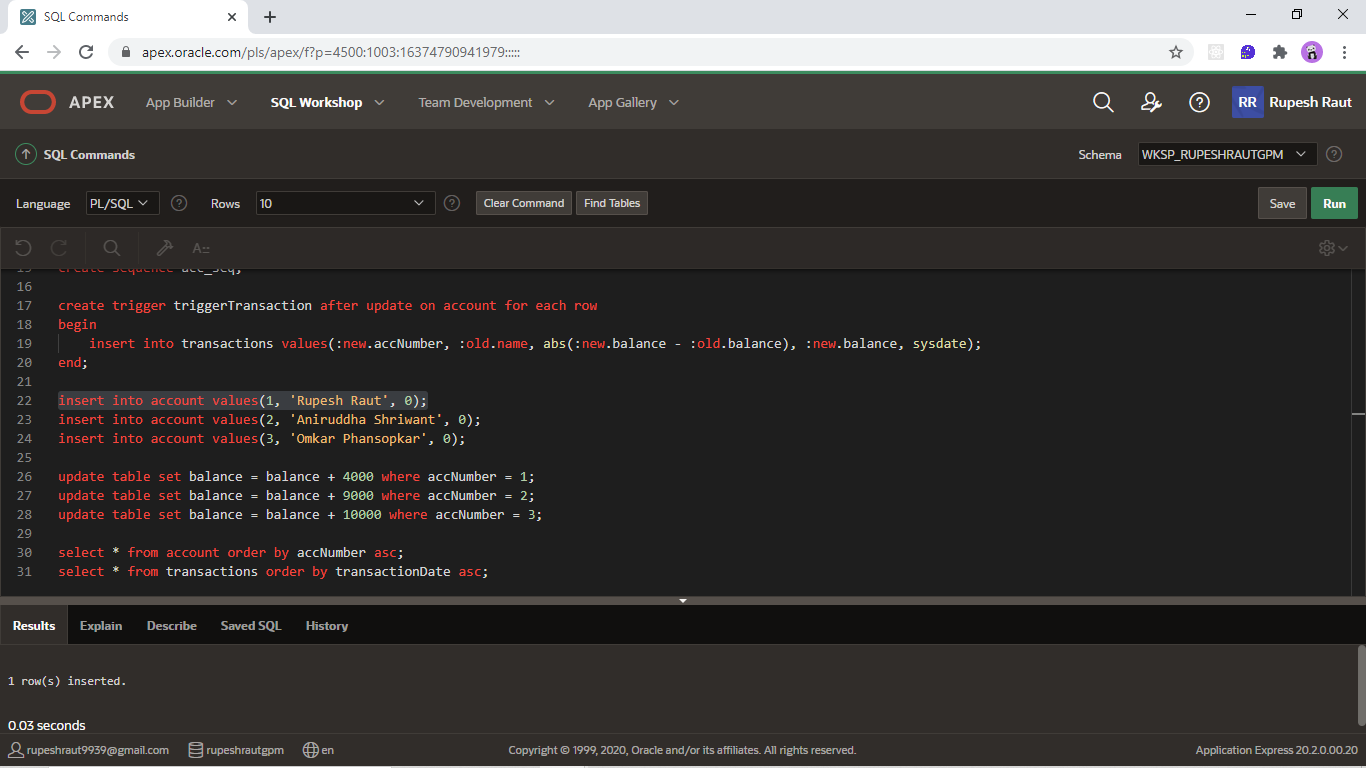
****

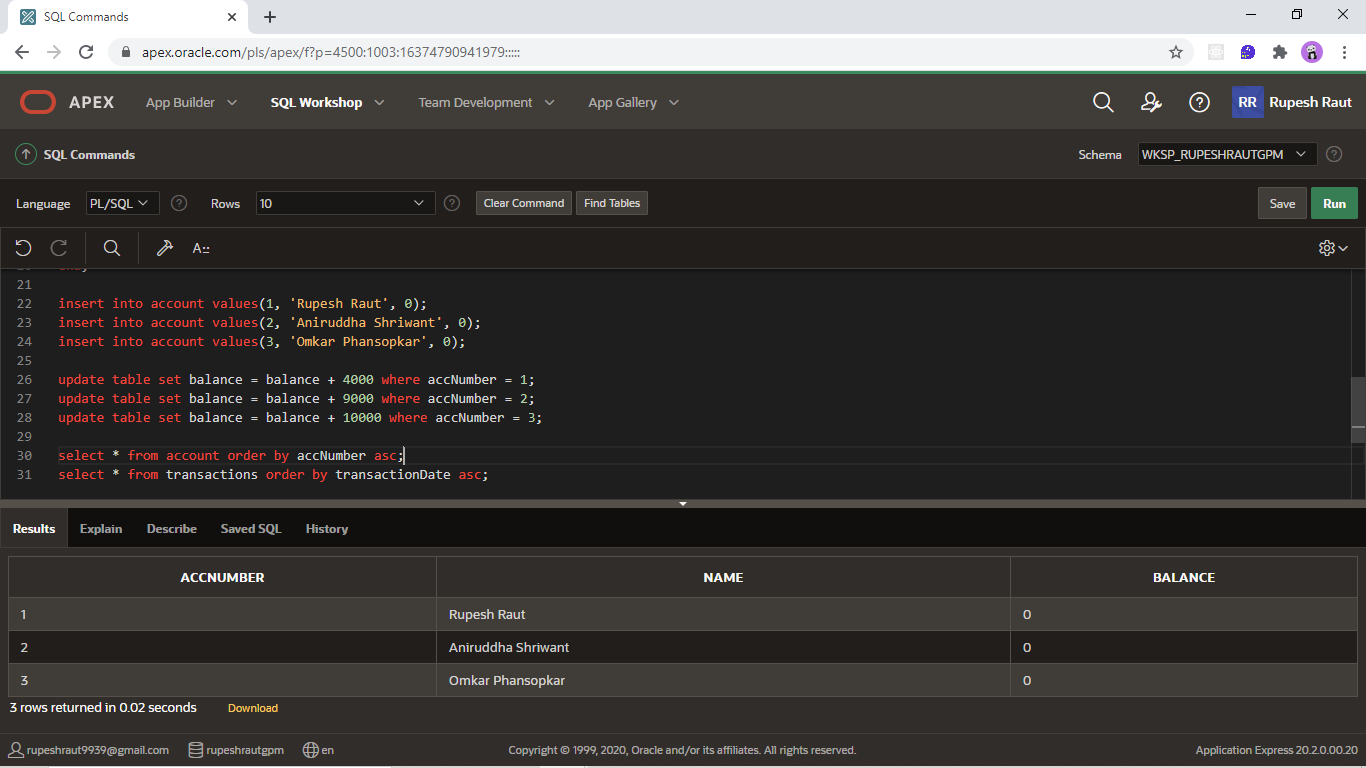
****

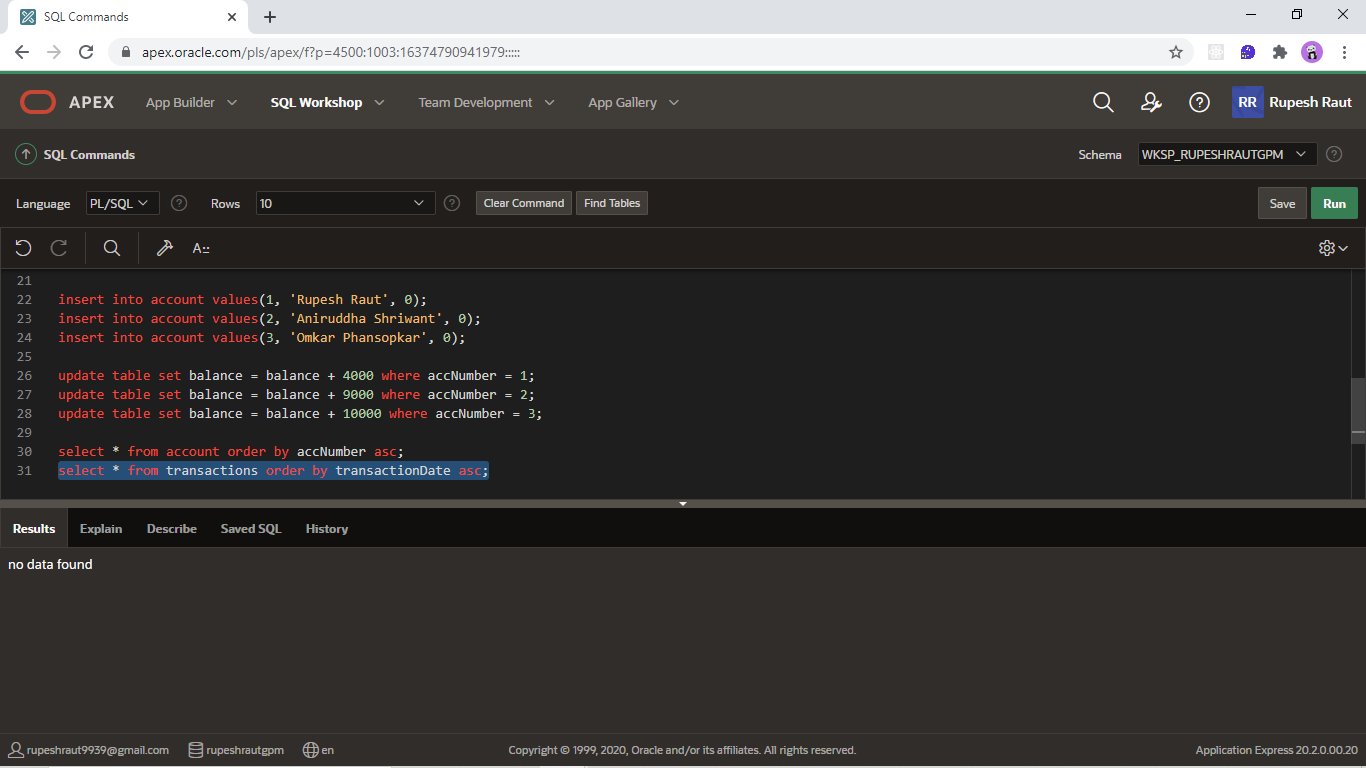
****

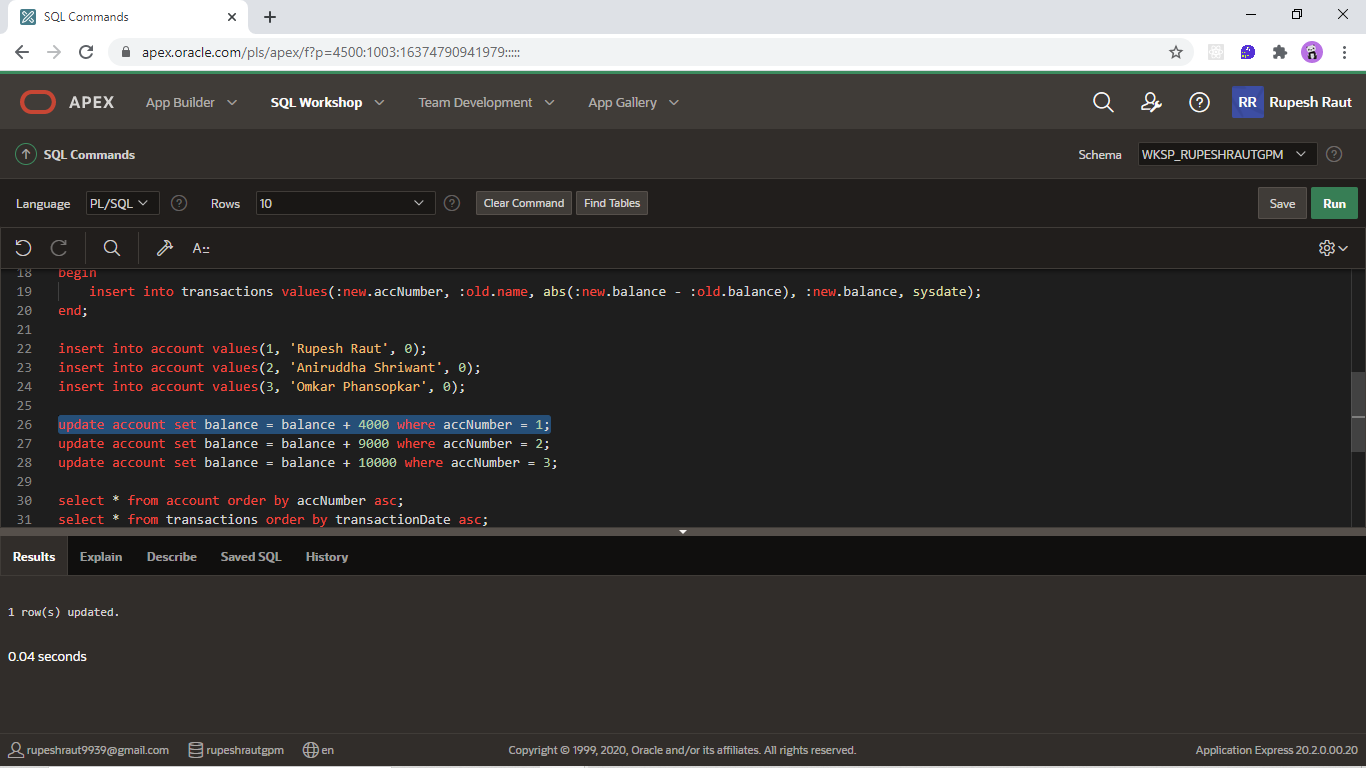
****

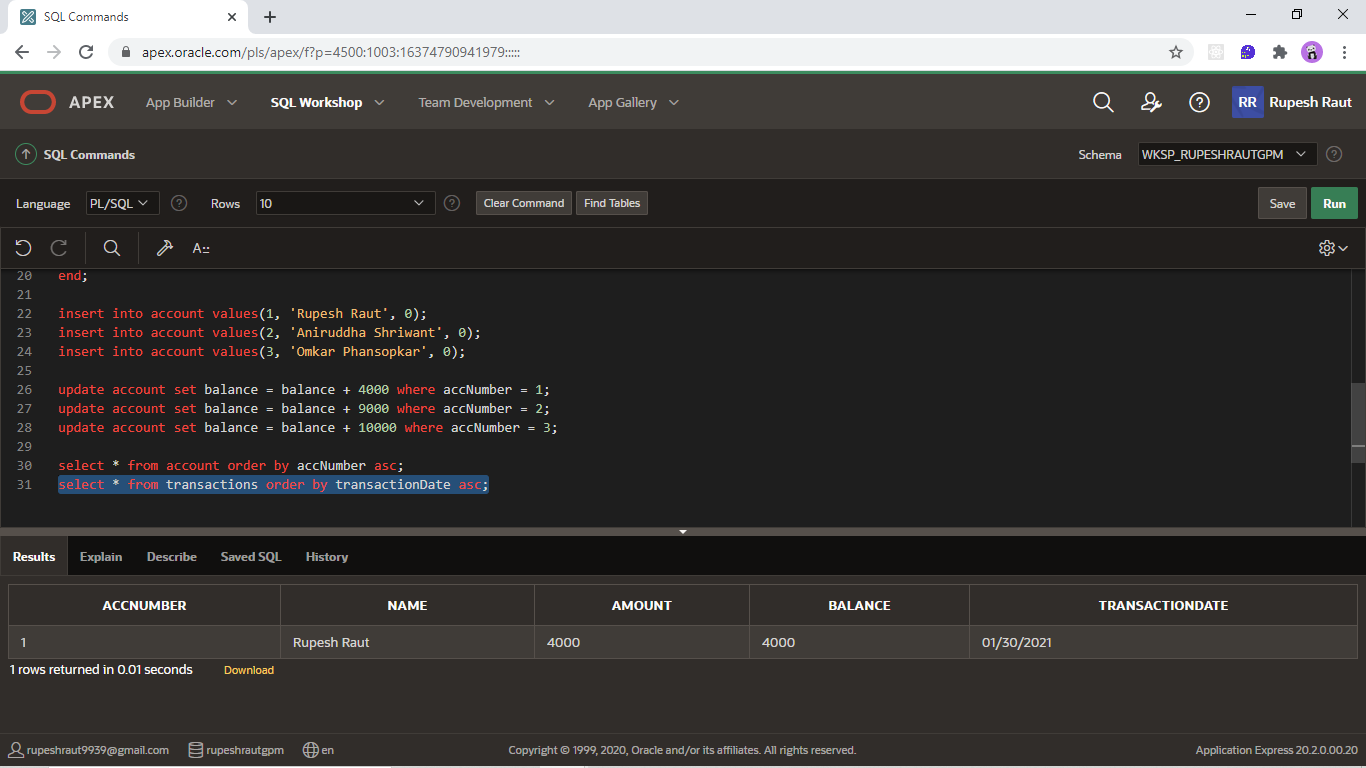
****

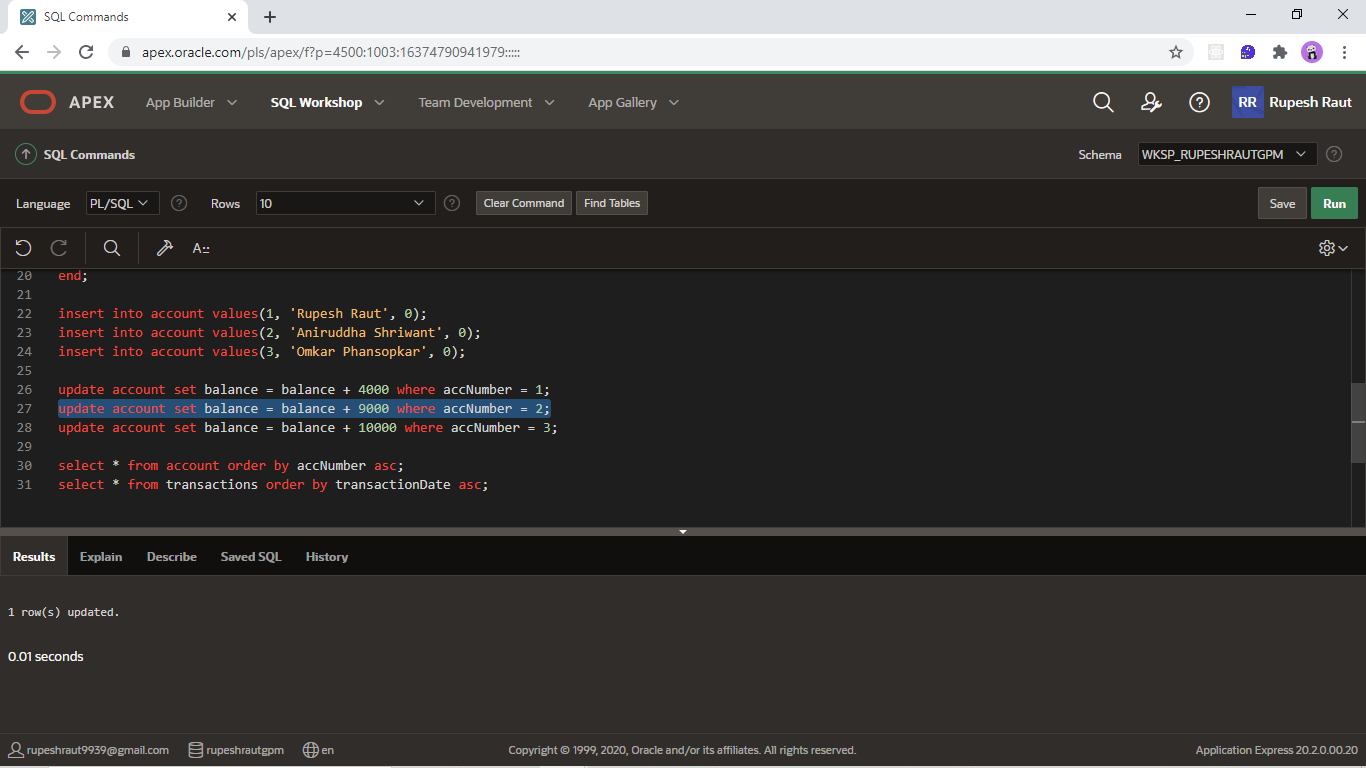
****

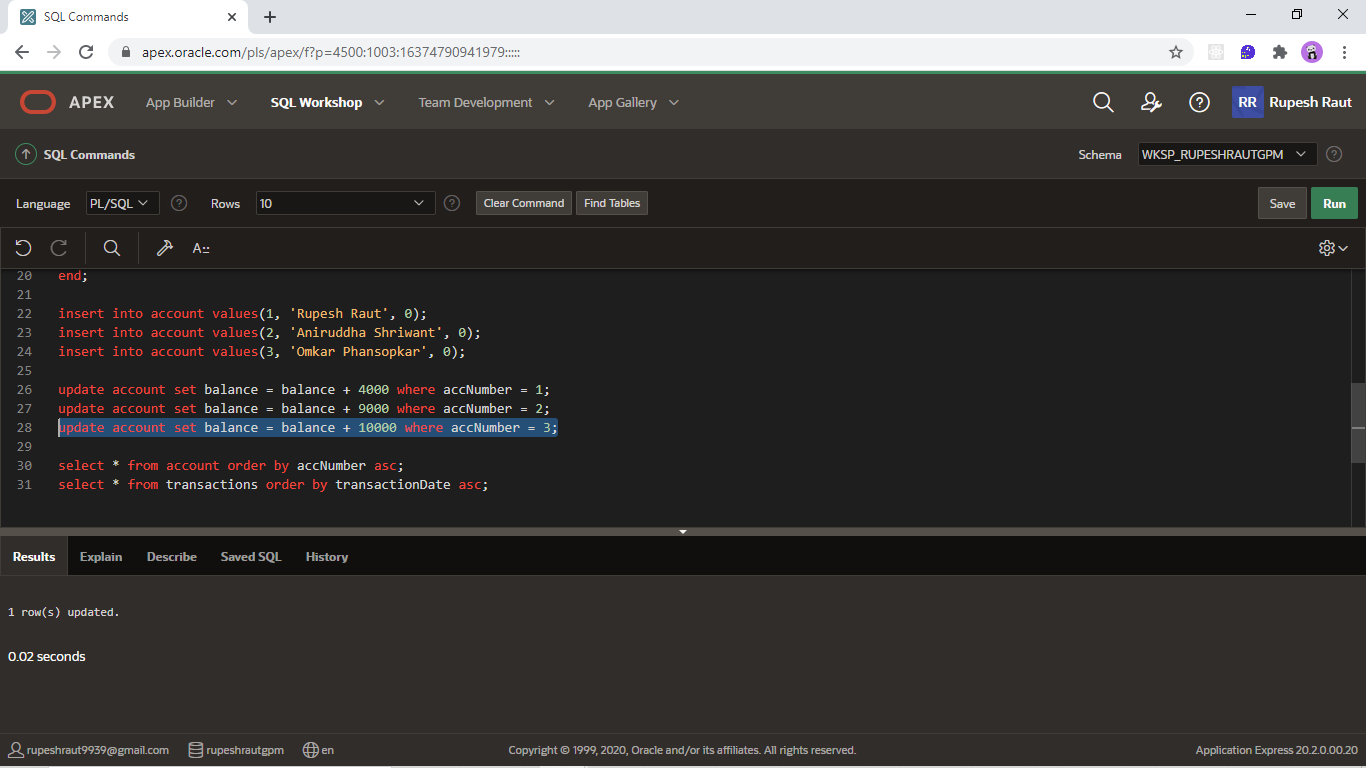
****

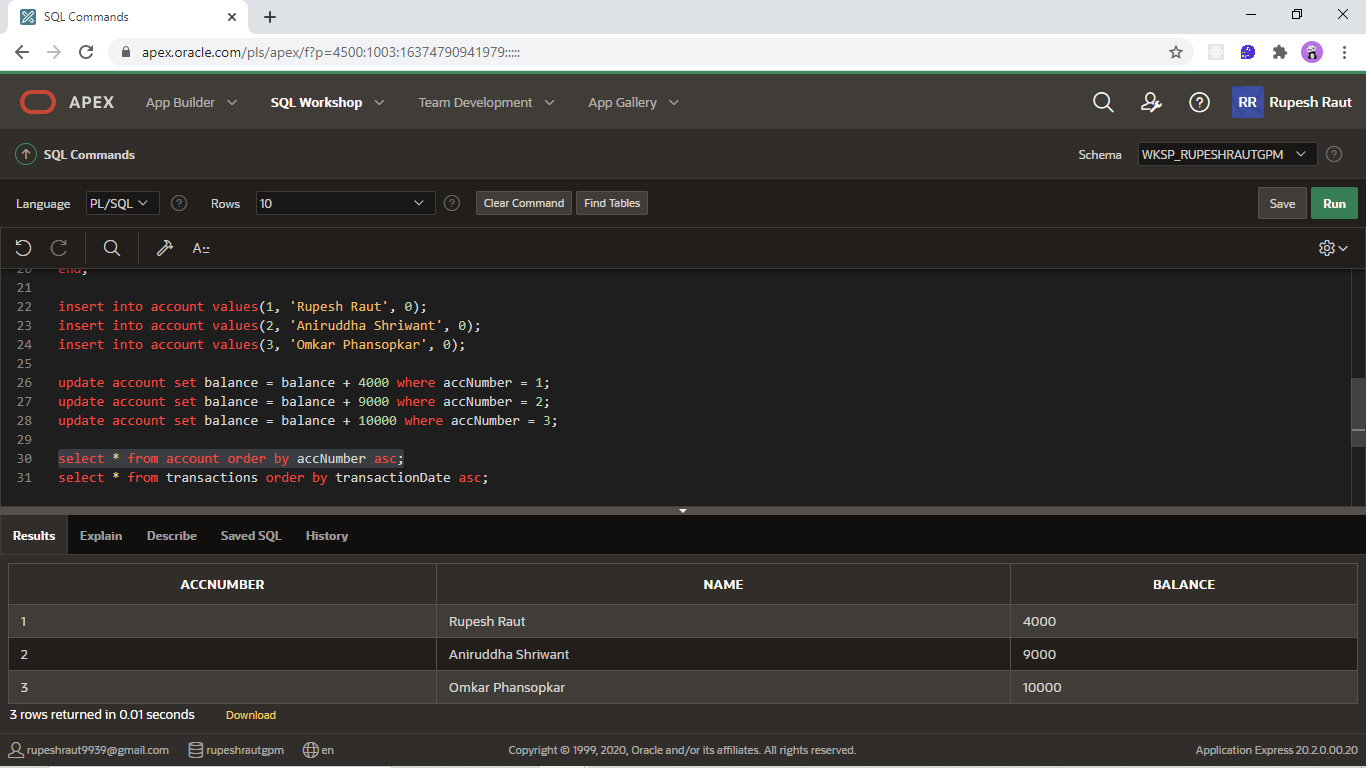
****

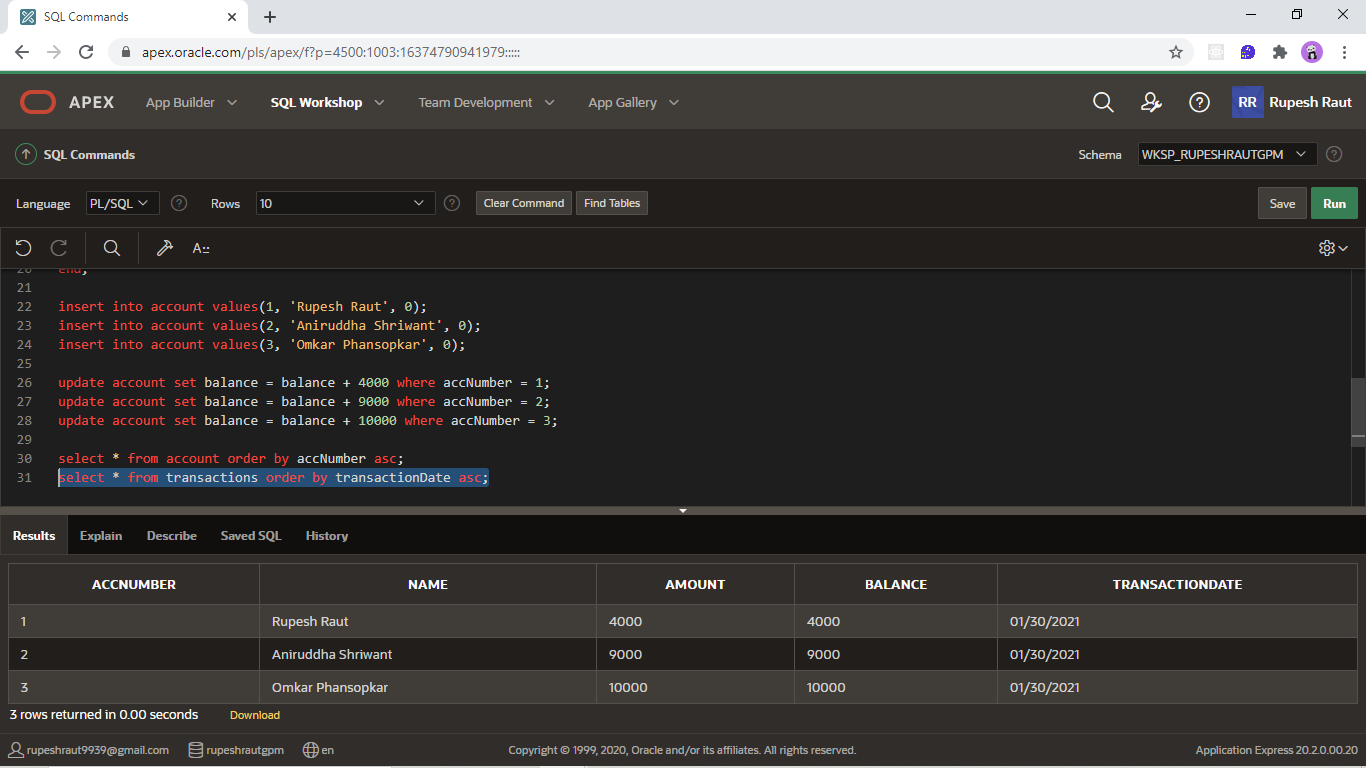
****

****

****

****

****

****

**Conclusion: Thus, we created triggers on database.**