Practical no. 8

FS19CO042

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';
```

2. 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:

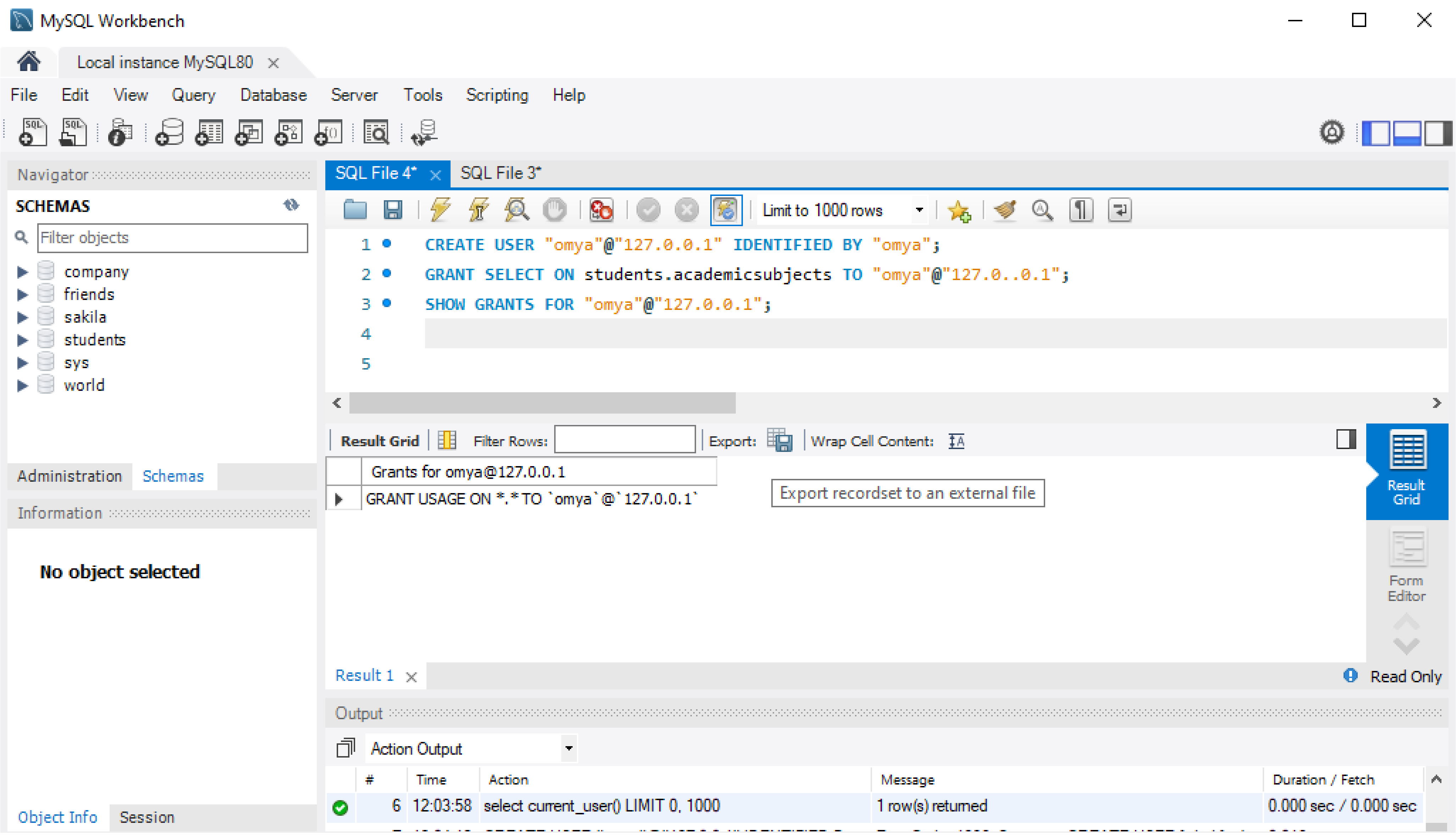
```
SAVEPOINT savepoint_name;
```

OUTPUT :

1. **Practical no. 9 DCL**
   1) **GRANT**

01. **Creating user**

02. **Granting permissions**

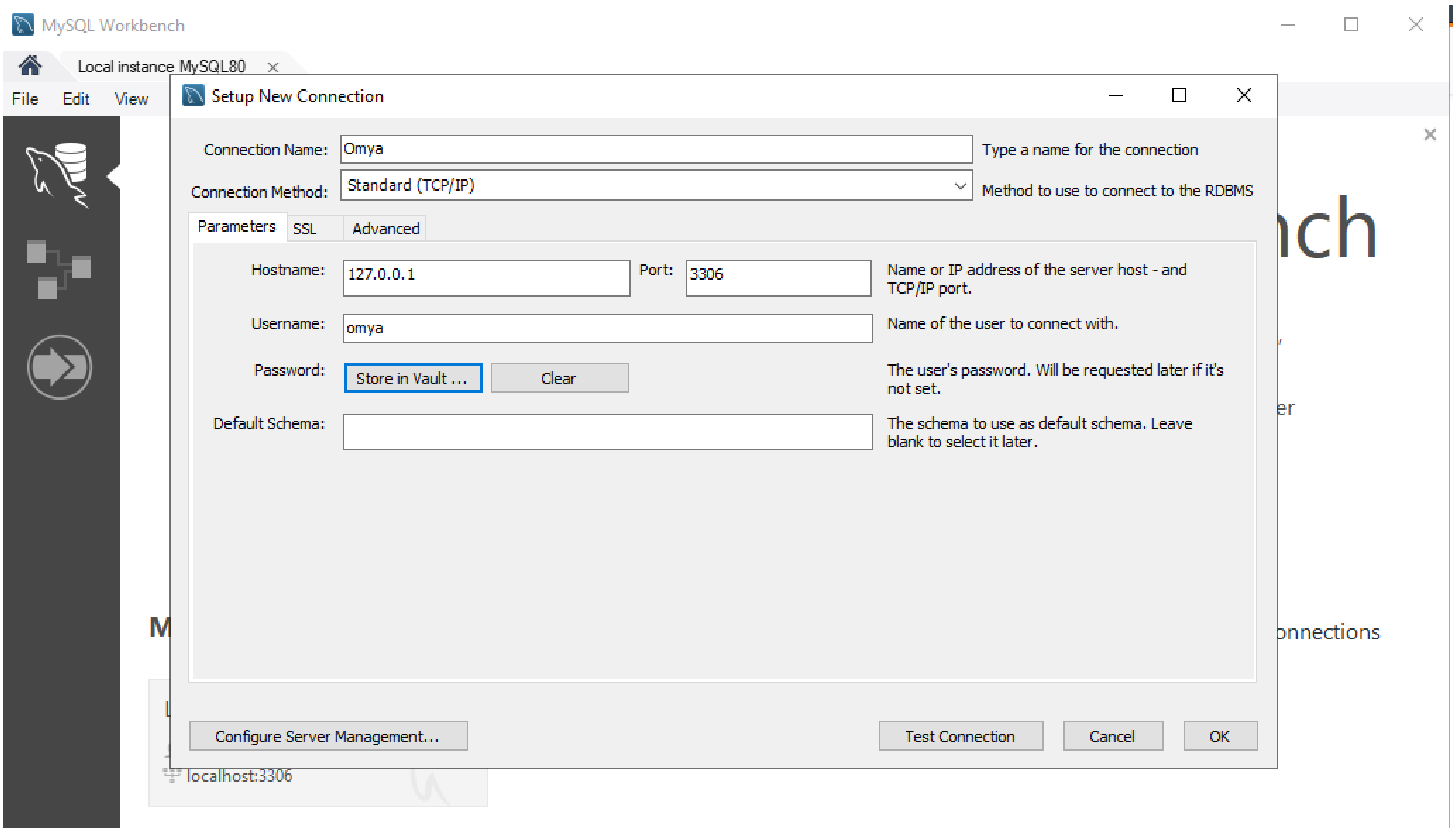


```
CREATE USER "omya"@"127.0.0.1" IDENTIFIED BY "omya";
GRANT SELECT ON students.academicsubjects TO "omya"@"127.0..0.1";
SHOW GRANTS FOR "omya"@"127.0.0.1";
```

Grants for omya@127.0.0.1
GRANT USAGE ON *.* TO \`omya\`@\`127.0.0.1\`

	#	Time	Action	Message	Duration / Fetch
✓	6	12:03:58	select current_user() LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec

03. **Making new connection**



Connection Name: Omya — Type a name for the connection

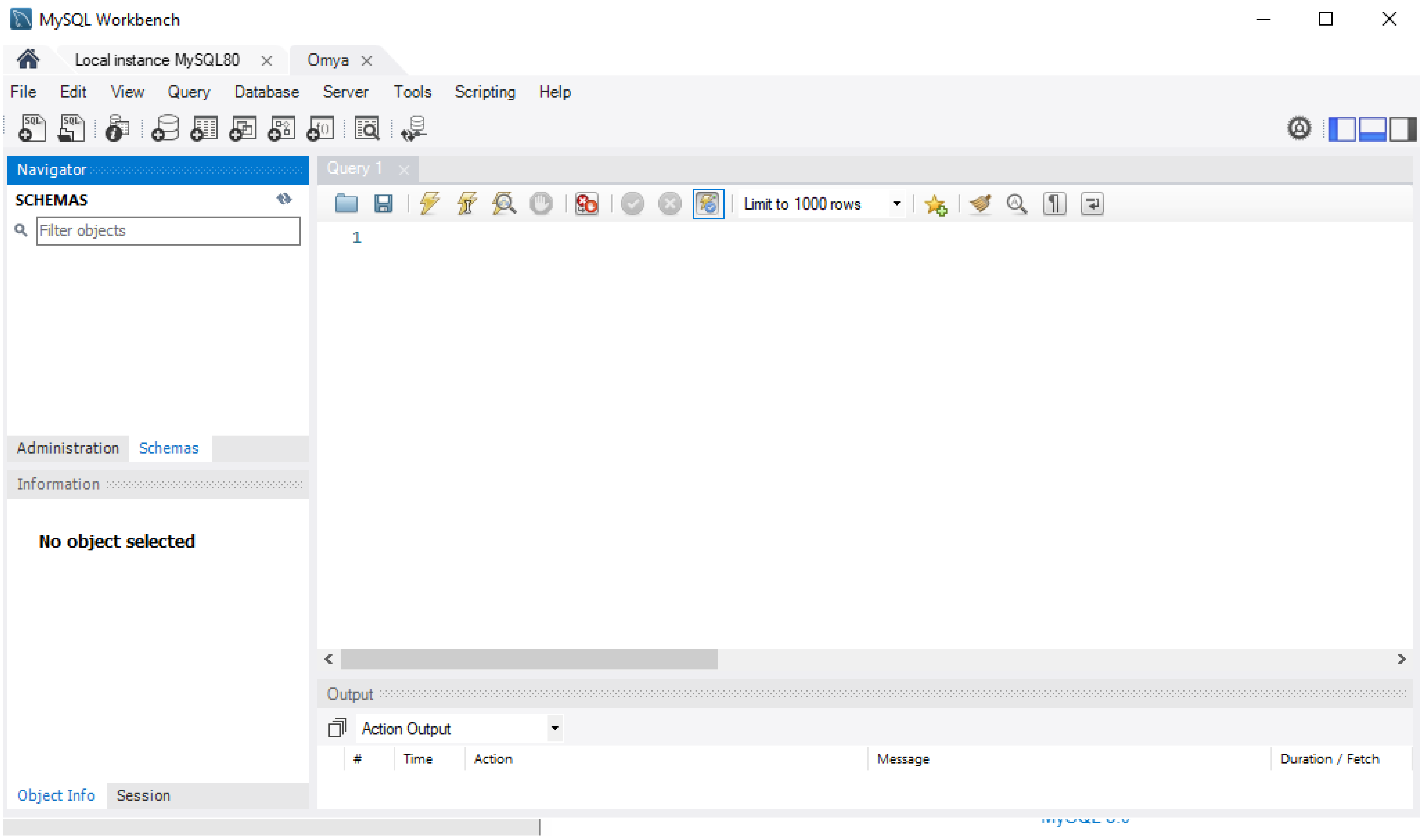
Connection Method: Standard (TCP/IP) — Method to use to connect to the RDBMS

Hostname: 127.0.0.1 Port: 3306 — Name or IP address of the server host - and TCP/IP port.

Username: omya — Name of the user to connect with.

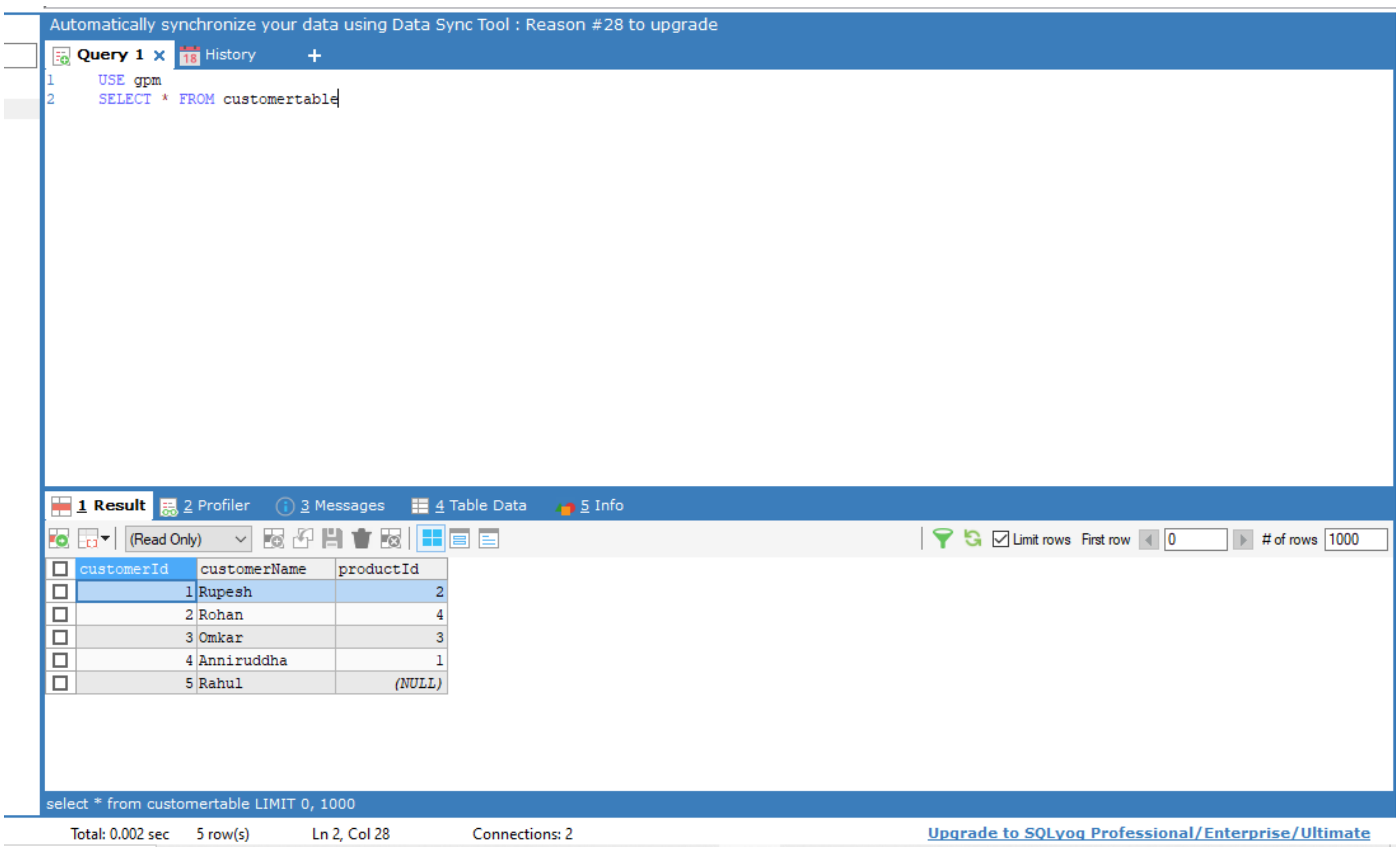
Password: — The user's password. Will be requested later if it's not set.

Default Schema: — The schema to use as default schema. Leave blank to select it later.



Omkar Phansopkar
FS19CO042

04. Use allowed commands


Automatically synchronize your data using Data Sync Tool : Reason #28 to upgrade
Query 1
History
1 Result
2 Profiler
3 Messages
4 Table Data
5 Info
(Read Only)
Limit rows
First row
# of rows
Upgrade to SQLyog Professional/Enterprise/Ultimate

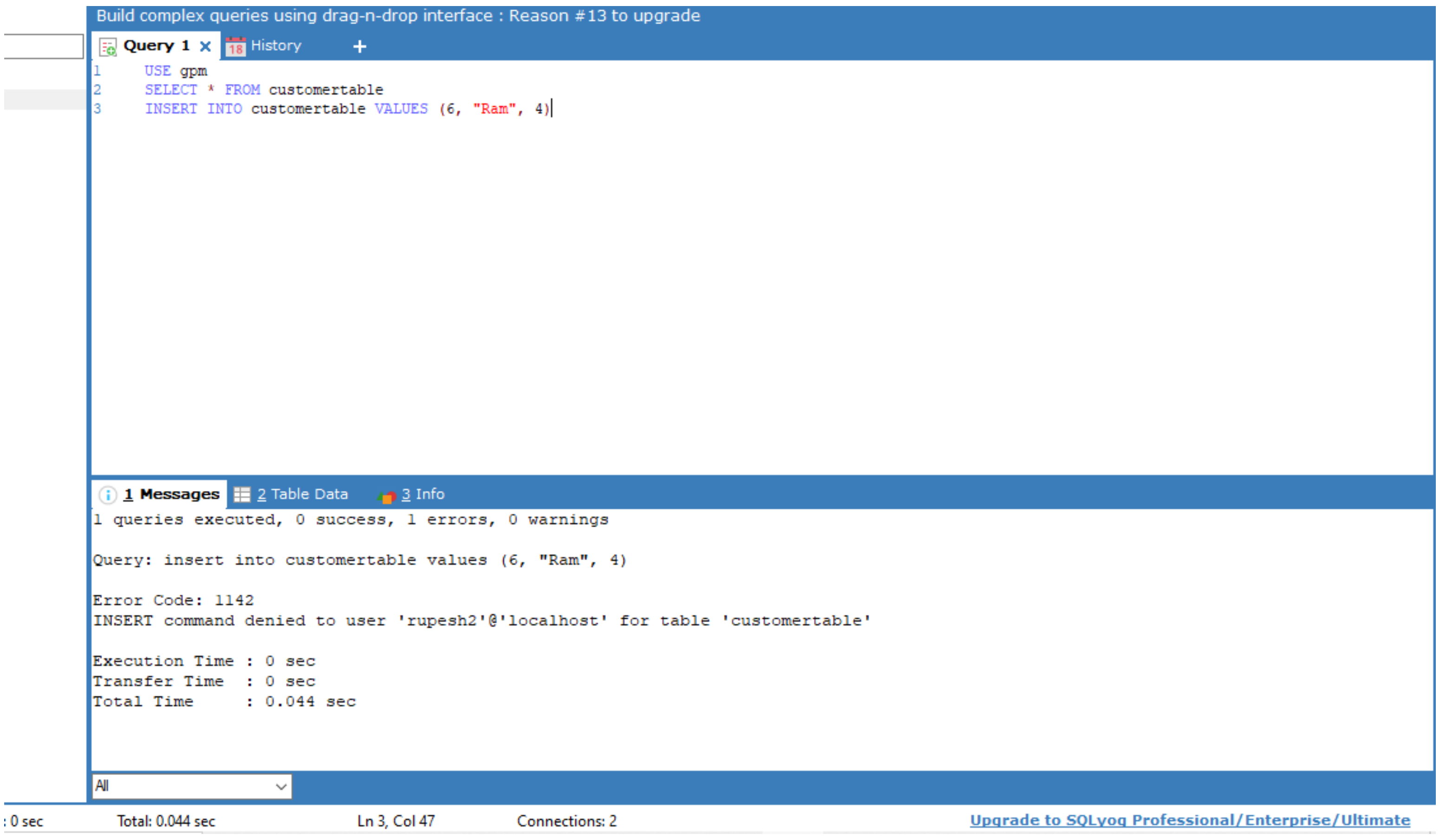
```
USE gpm
SELECT * FROM customertable
```

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)

☑ Limit rows First row 0 # of rows 1000

select * from customertable LIMIT 0, 1000

Total: 0.002 sec 5 row(s) Ln 2, Col 28 Connections: 2

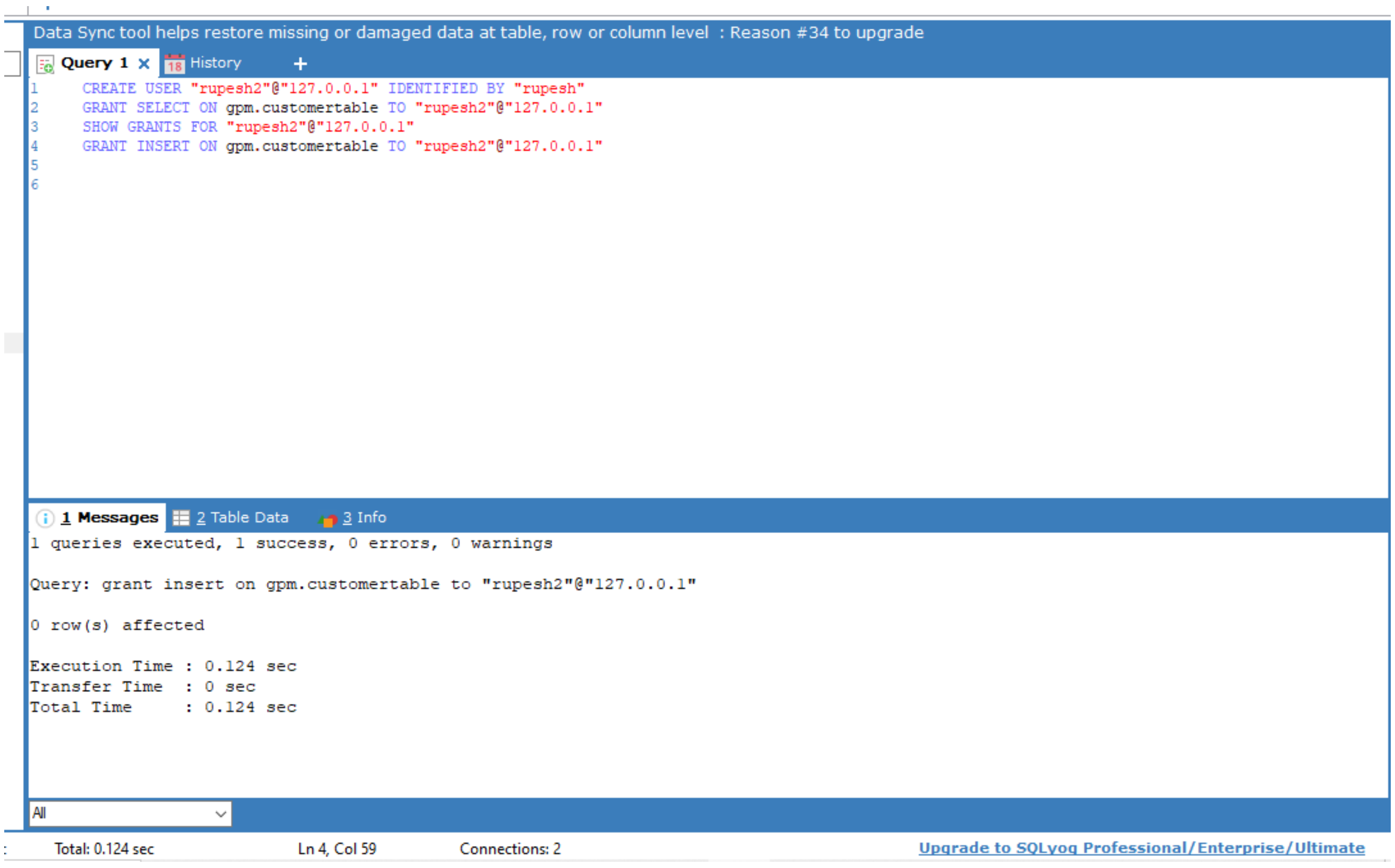

Build complex queries using drag-n-drop interface : Reason #13 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
INSERT INTO customertable VALUES (6, "Ram", 4)
```

```
1 queries executed, 0 success, 1 errors, 0 warnings

Query: insert into customertable values (6, "Ram", 4)

Error Code: 1142
INSERT command denied to user 'rupesh2'@'localhost' for table 'customertable'

Execution Time : 0 sec
Transfer Time  : 0 sec
Total Time     : 0.044 sec
```

: 0 sec Total: 0.044 sec Ln 3, Col 47 Connections: 2

Omkar Phansopkar
FS19CO042

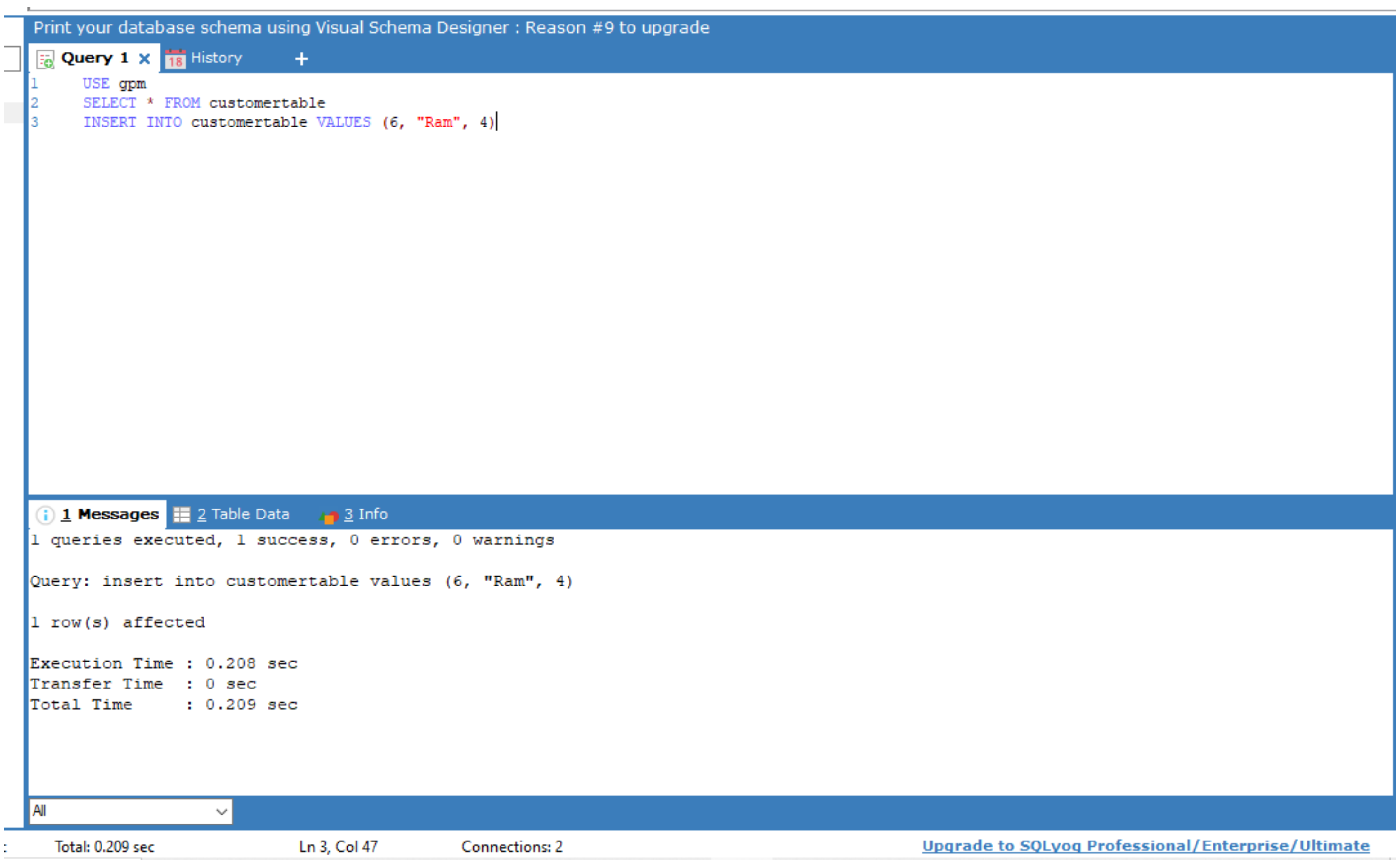


Data Sync tool helps restore missing or damaged data at table, row or column level : Reason #34 to upgrade

```
CREATE USER "rupesh2"@"127.0.0.1" IDENTIFIED BY "rupesh"
GRANT SELECT ON gpm.customertable TO "rupesh2"@"127.0.0.1"
SHOW GRANTS FOR "rupesh2"@"127.0.0.1"
GRANT INSERT ON gpm.customertable TO "rupesh2"@"127.0.0.1"
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: grant insert on gpm.customertable to "rupesh2"@"127.0.0.1"

0 row(s) affected

Execution Time : 0.124 sec
Transfer Time  : 0 sec
Total Time     : 0.124 sec
```

Total: 0.124 sec Ln 4, Col 59 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate



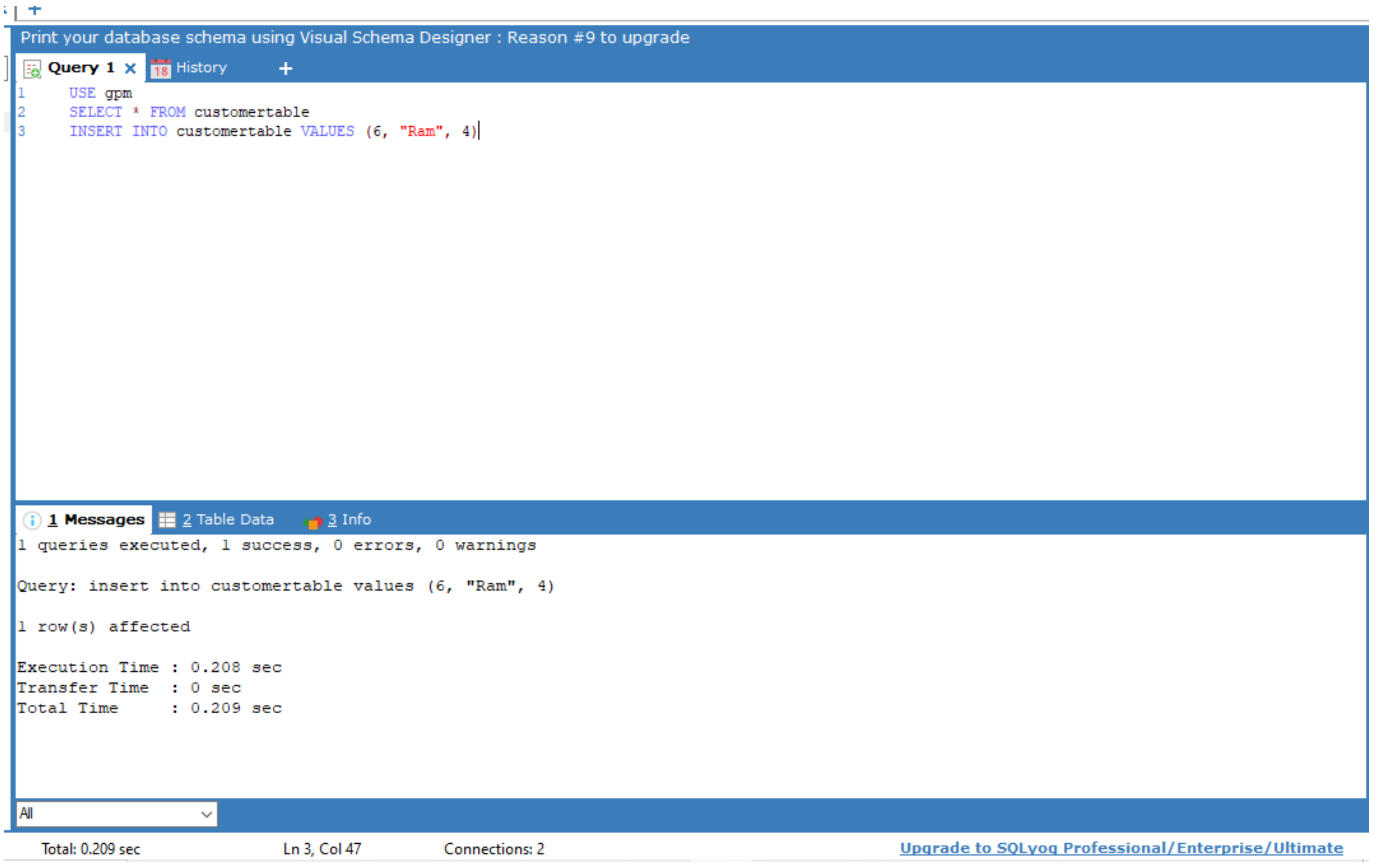
Print your database schema using Visual Schema Designer : Reason #9 to upgrade

```
USE gpm
SELECT * FROM customertable
INSERT INTO customertable VALUES (6, "Ram", 4)
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: insert into customertable values (6, "Ram", 4)

1 row(s) affected

Execution Time : 0.208 sec
Transfer Time  : 0 sec
Total Time     : 0.209 sec
```

Total: 0.209 sec Ln 3, Col 47 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

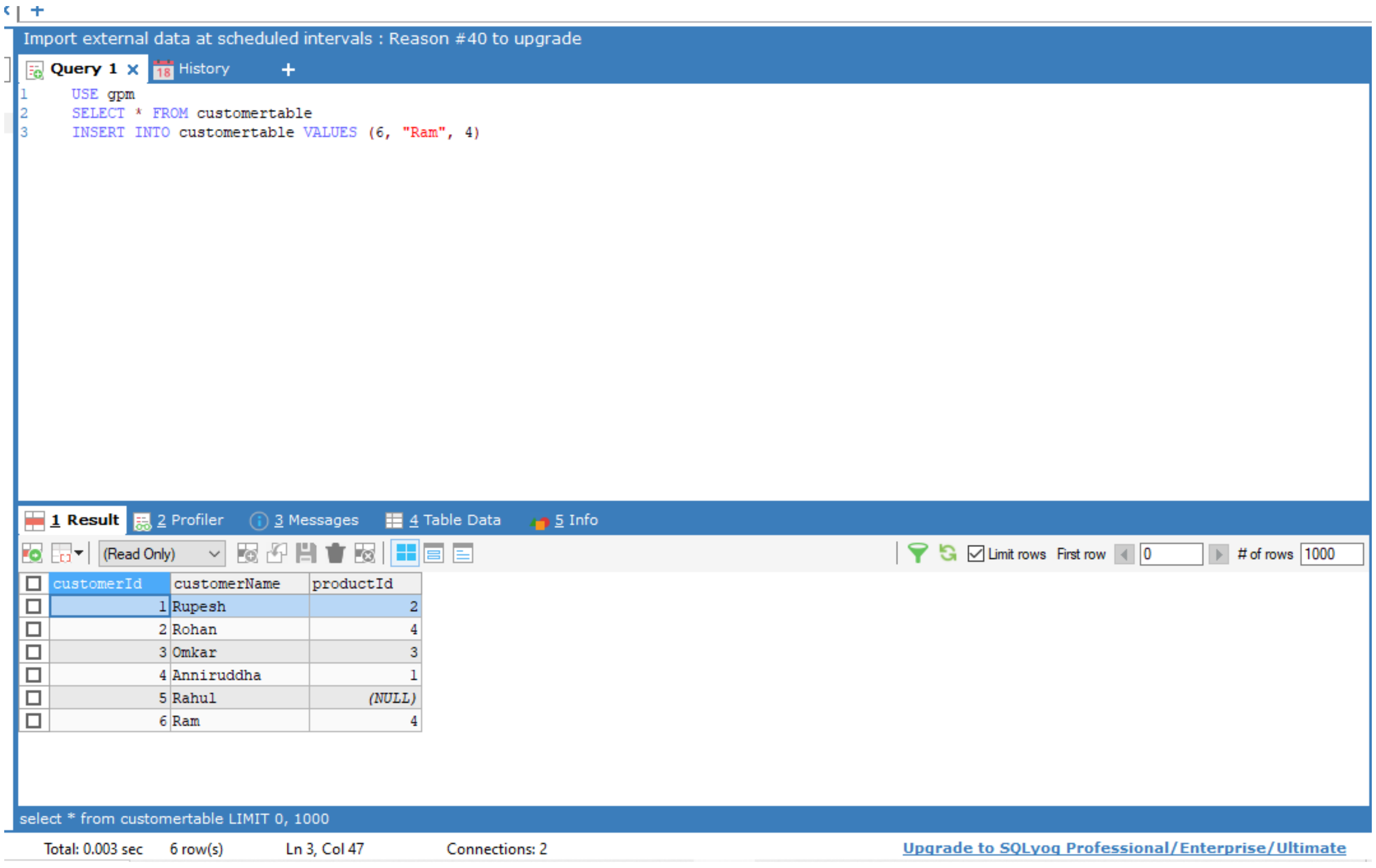
Omkar Phansopkar
FS19CO042


Print your database schema using Visual Schema Designer : Reason #9 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
INSERT INTO customertable VALUES (6, "Ram", 4)
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: insert into customertable values (6, "Ram", 4)

1 row(s) affected

Execution Time : 0.208 sec
Transfer Time  : 0 sec
Total Time     : 0.209 sec
```

Total: 0.209 sec Ln 3, Col 47 Connections: 2


Import external data at scheduled intervals : Reason #40 to upgrade
Query 1
History
1 Result
2 Profiler
3 Messages
4 Table Data
5 Info
(Read Only)
Limit rows
First row
0
# of rows
1000
Upgrade to SQLyog Professional/Enterprise/Ultimate

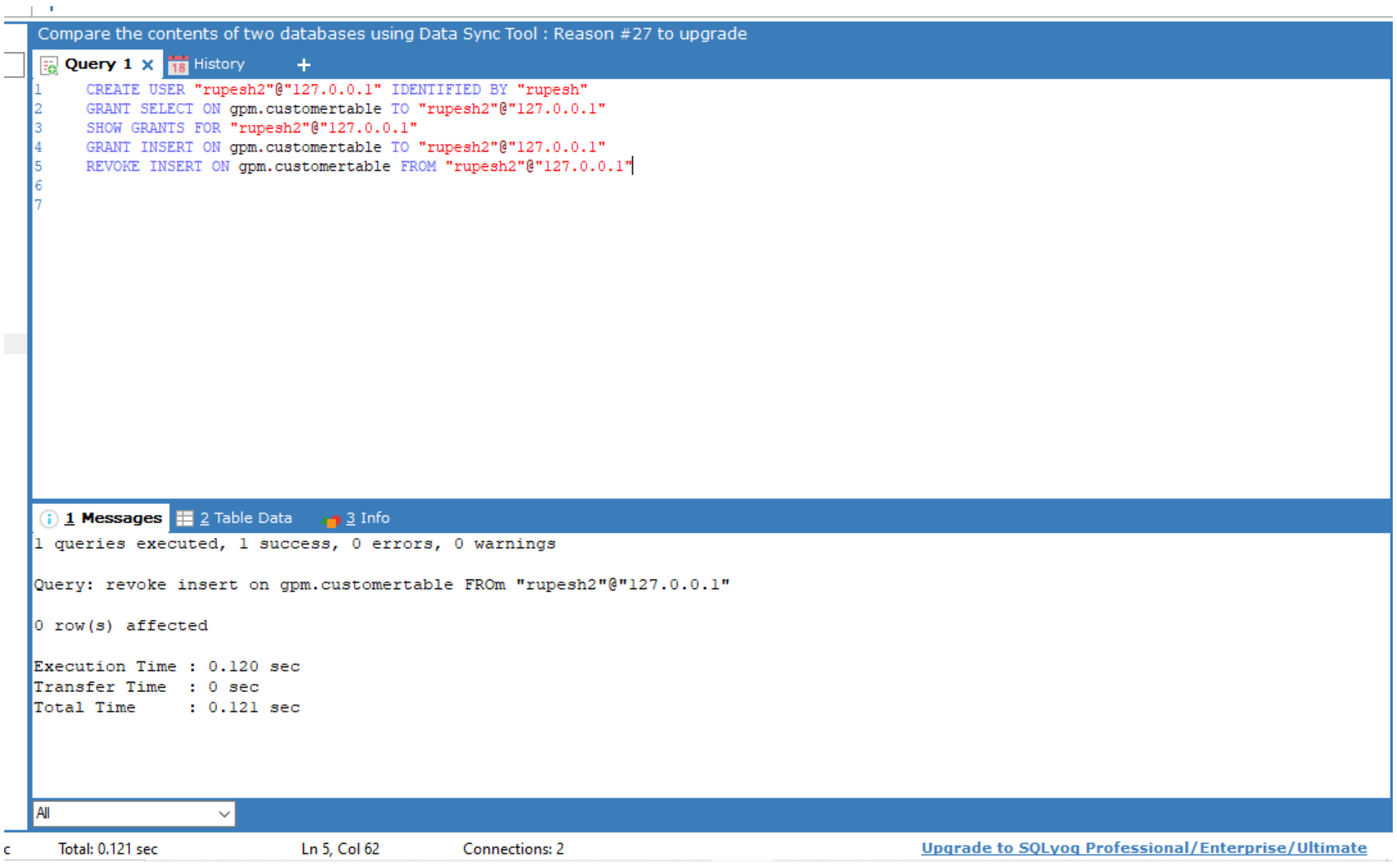
```
USE gpm
SELECT * FROM customertable
INSERT INTO customertable VALUES (6, "Ram", 4)
```

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)
6	Ram	4

select * from customertable LIMIT 0, 1000

Total: 0.003 sec 6 row(s) Ln 3, Col 47 Connections: 2

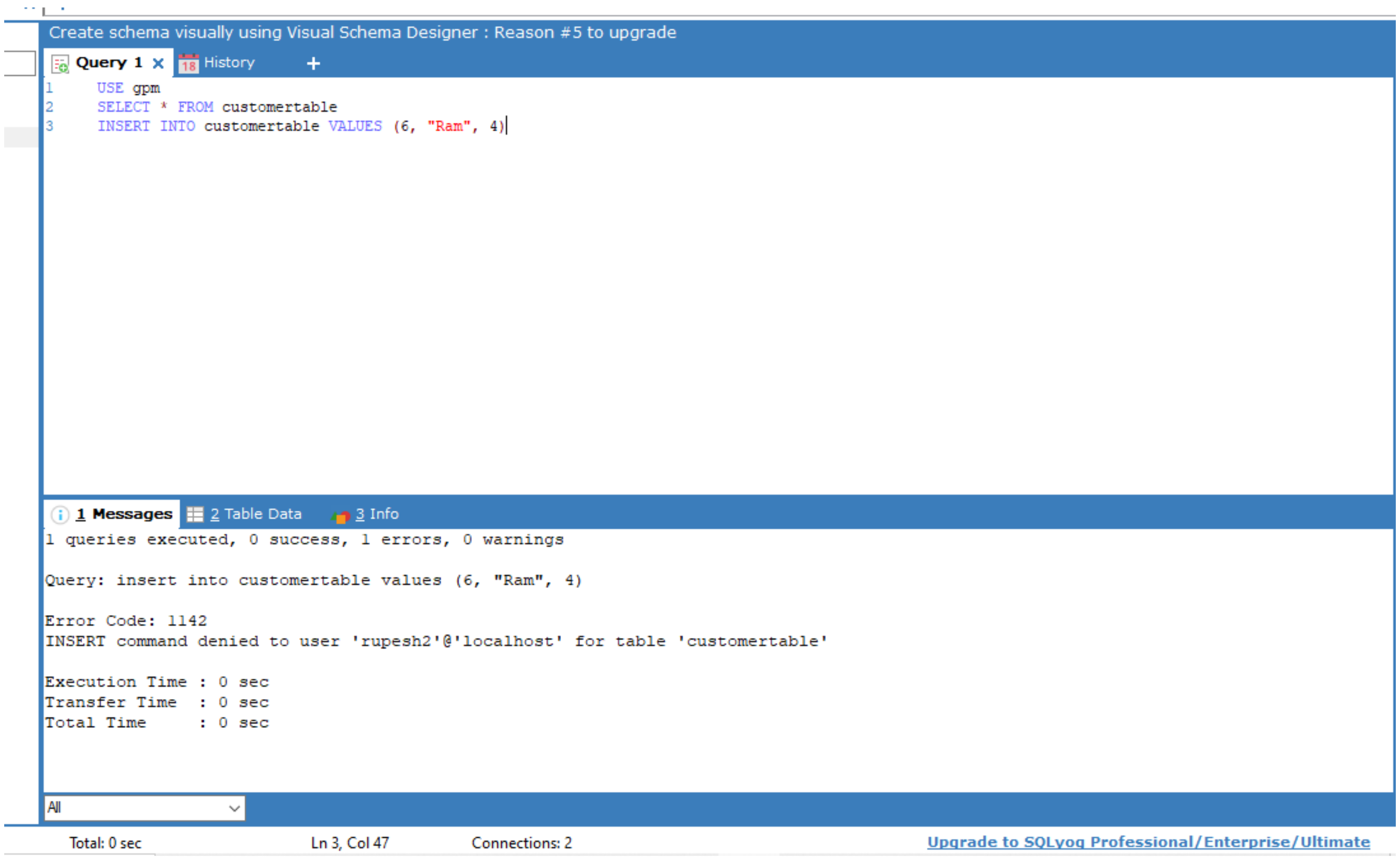
2. REVOKE


Compare the contents of two databases using Data Sync Tool : Reason #27 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
CREATE USER "rupesh2"@"127.0.0.1" IDENTIFIED BY "rupesh"
GRANT SELECT ON gpm.customertable TO "rupesh2"@"127.0.0.1"
SHOW GRANTS FOR "rupesh2"@"127.0.0.1"
GRANT INSERT ON gpm.customertable TO "rupesh2"@"127.0.0.1"
REVOKE INSERT ON gpm.customertable FROM "rupesh2"@"127.0.0.1"
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: revoke insert on gpm.customertable FROm "rupesh2"@"127.0.0.1"

0 row(s) affected

Execution Time : 0.120 sec
Transfer Time  : 0 sec
Total Time     : 0.121 sec
```

Total: 0.121 sec   Ln 5, Col 62   Connections: 2


Create schema visually using Visual Schema Designer : Reason #5 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Upgrade to SQLyog Professional/Enterprise/Ultimate

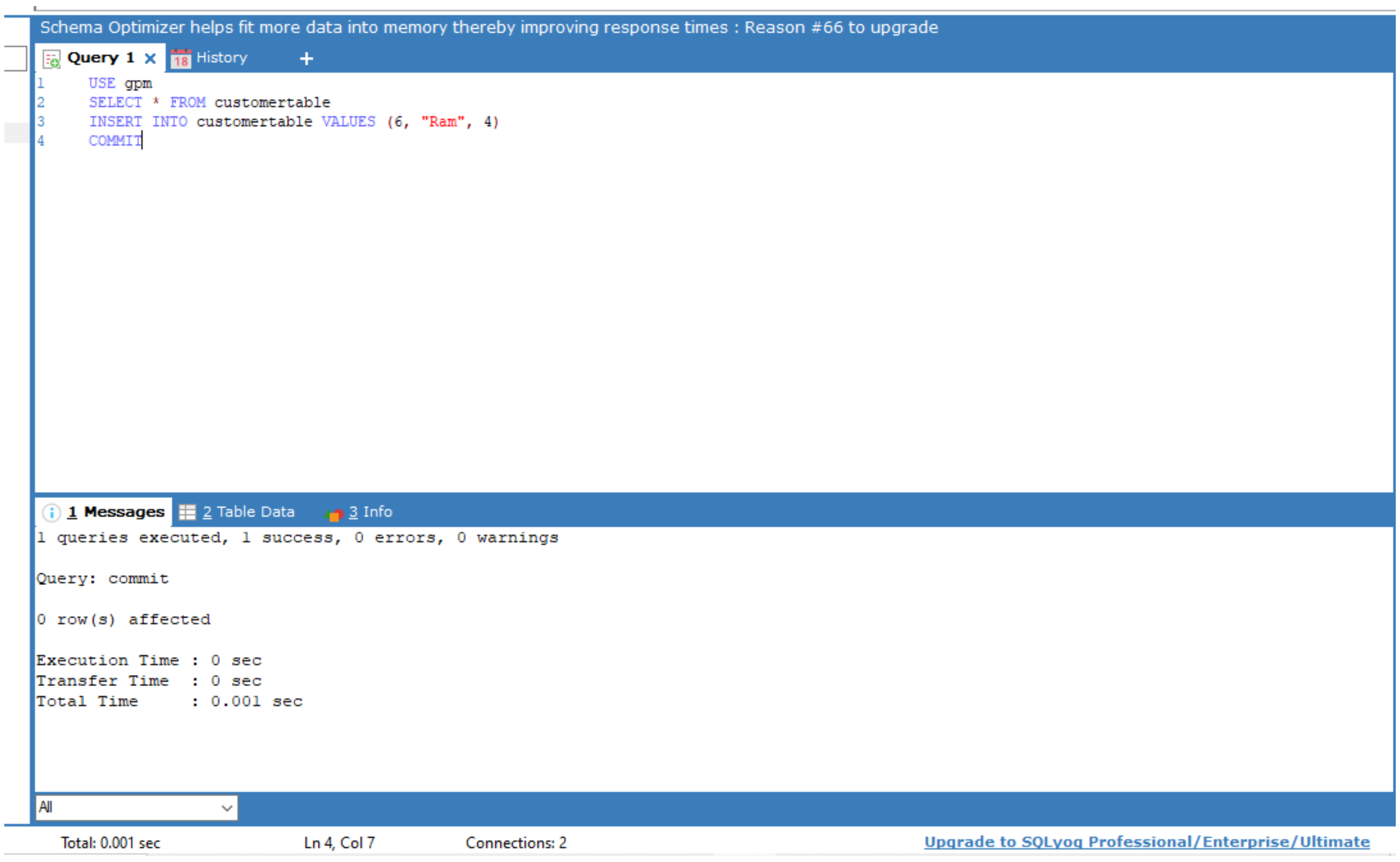
```
USE gpm
SELECT * FROM customertable
INSERT INTO customertable VALUES (6, "Ram", 4)
```

```
1 queries executed, 0 success, 1 errors, 0 warnings

Query: insert into customertable values (6, "Ram", 4)

Error Code: 1142
INSERT command denied to user 'rupesh2'@'localhost' for table 'customertable'

Execution Time : 0 sec
Transfer Time  : 0 sec
Total Time     : 0 sec
```

Total: 0 sec   Ln 3, Col 47   Connections: 2

2. TCL

1) COMMIT



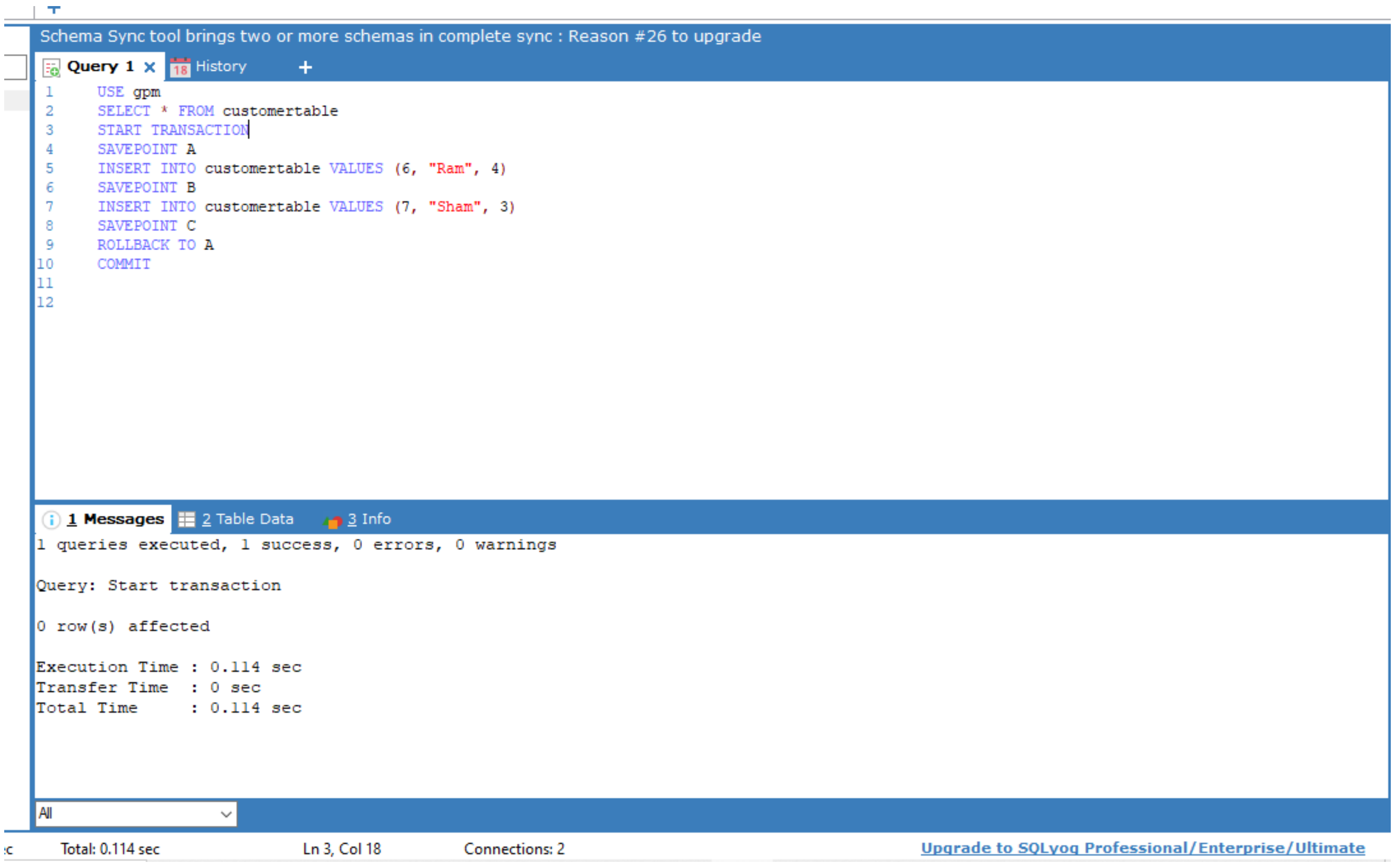
Schema Optimizer helps fit more data into memory thereby improving response times : Reason #66 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All

```
USE gpm
SELECT * FROM customertable
INSERT INTO customertable VALUES (6, "Ram", 4)
COMMIT
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: commit

0 row(s) affected

Execution Time : 0 sec
Transfer Time  : 0 sec
Total Time     : 0.001 sec
```

Total: 0.001 sec
Ln 4, Col 7
Connections: 2
Upgrade to SQLyog Professional/Enterprise/Ultimate

2) SAVEPOINT



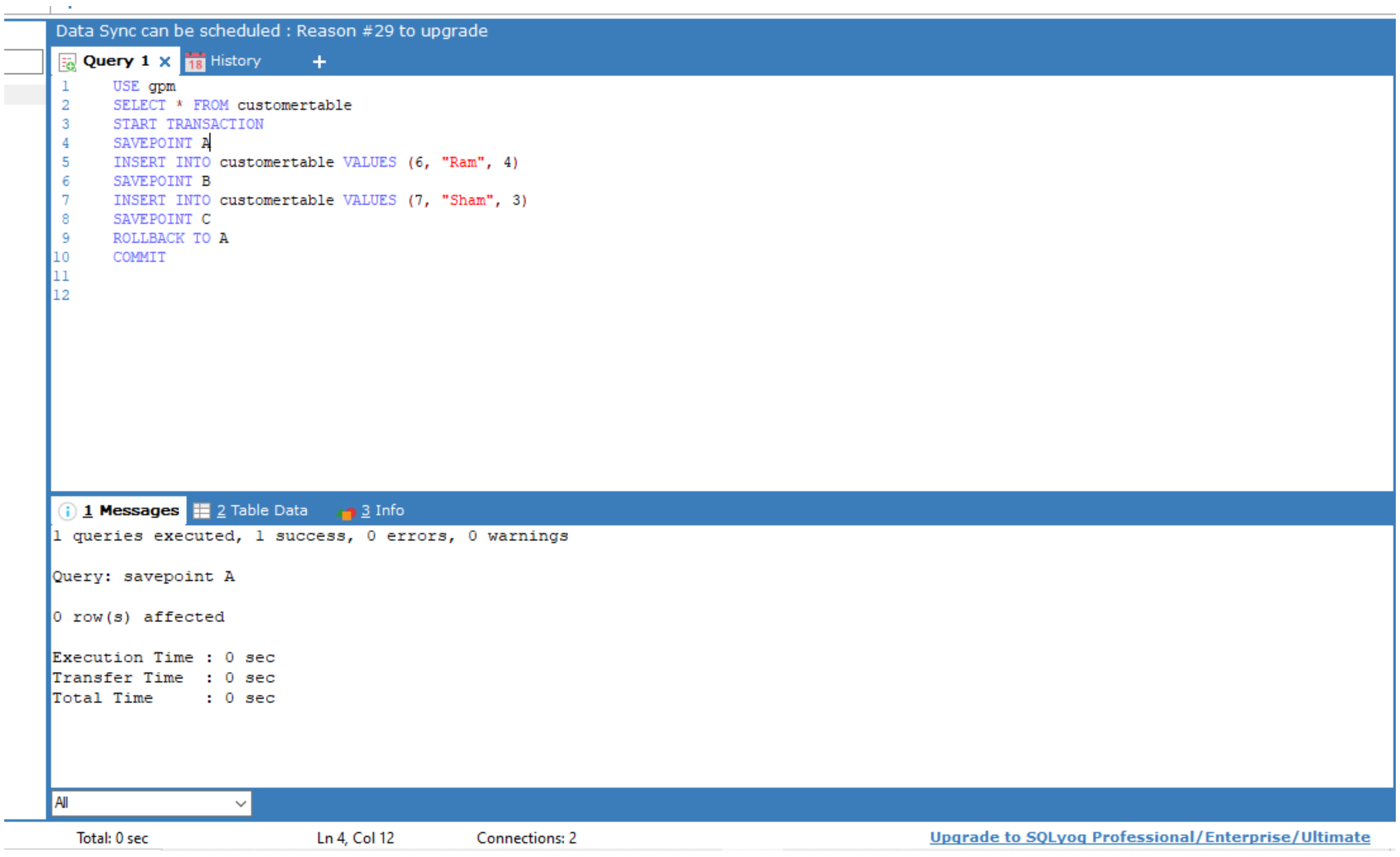
Schema Sync tool brings two or more schemas in complete sync : Reason #26 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All

```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: Start transaction

0 row(s) affected

Execution Time : 0.114 sec
Transfer Time  : 0 sec
Total Time     : 0.114 sec
```

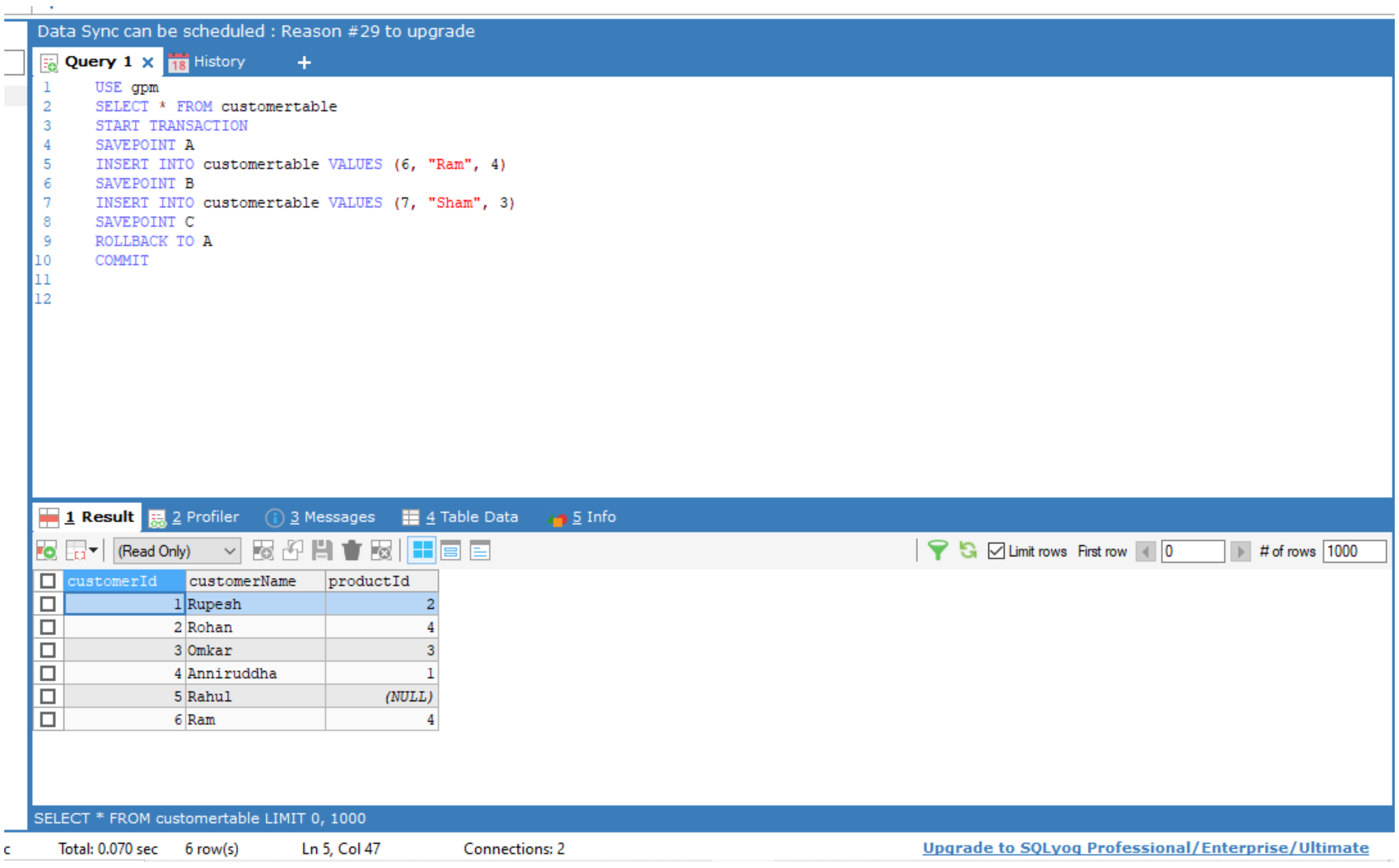
Total: 0.114 sec
Ln 3, Col 18
Connections: 2
Upgrade to SQLyog Professional/Enterprise/Ultimate

Omkar Phansopkar
FS19CO042


Data Sync can be scheduled : Reason #29 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Total: 0 sec
Ln 4, Col 12
Connections: 2
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

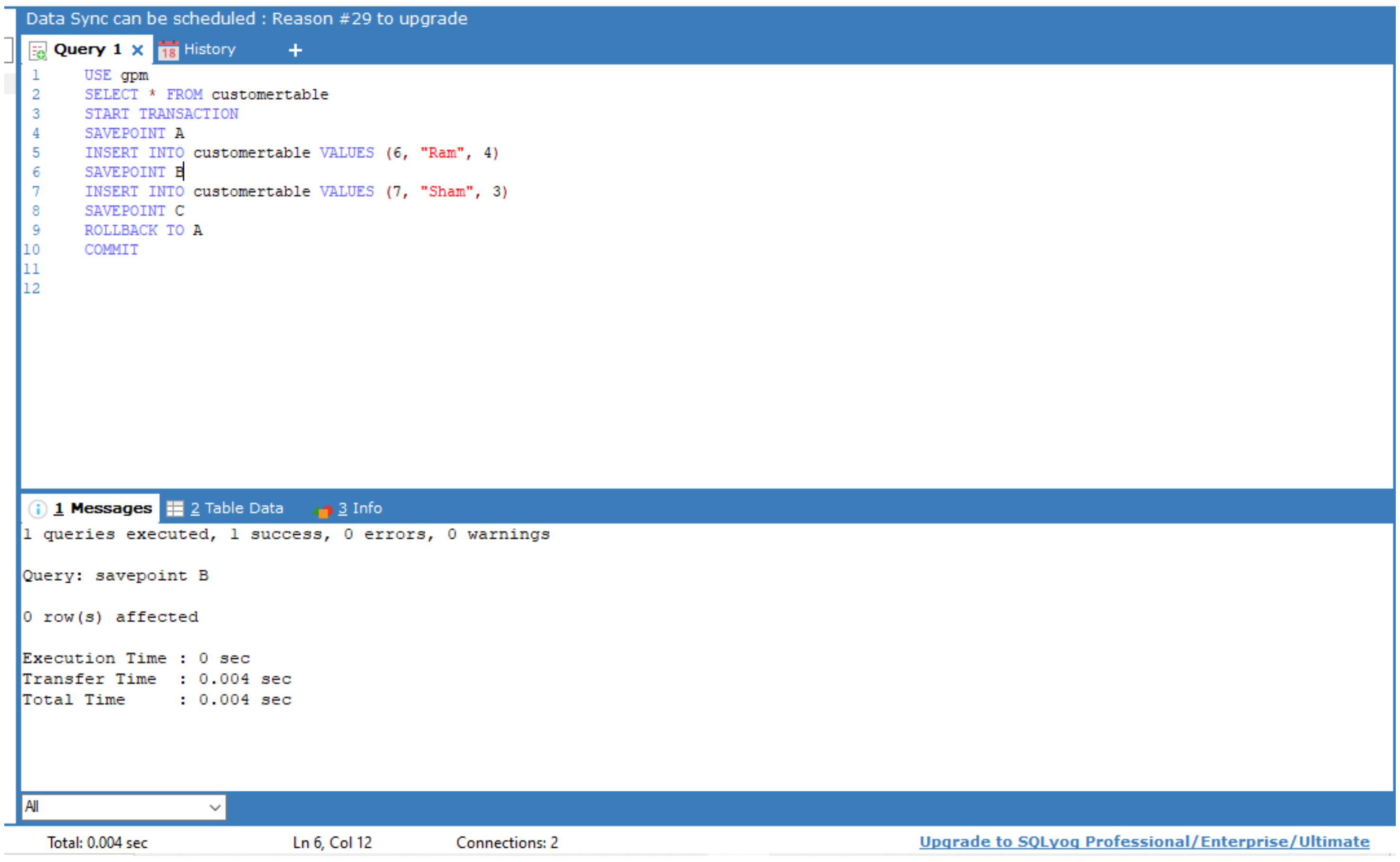
```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint A

0 row(s) affected

Execution Time : 0 sec
Transfer Time  : 0 sec
Total Time     : 0 sec
```



```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

□	customerId	customerName	productId
□	1	Rupesh	2
□	2	Rohan	4
□	3	Omkar	3
□	4	Anniruddha	1
□	5	Rahul	(NULL)
□	6	Ram	4

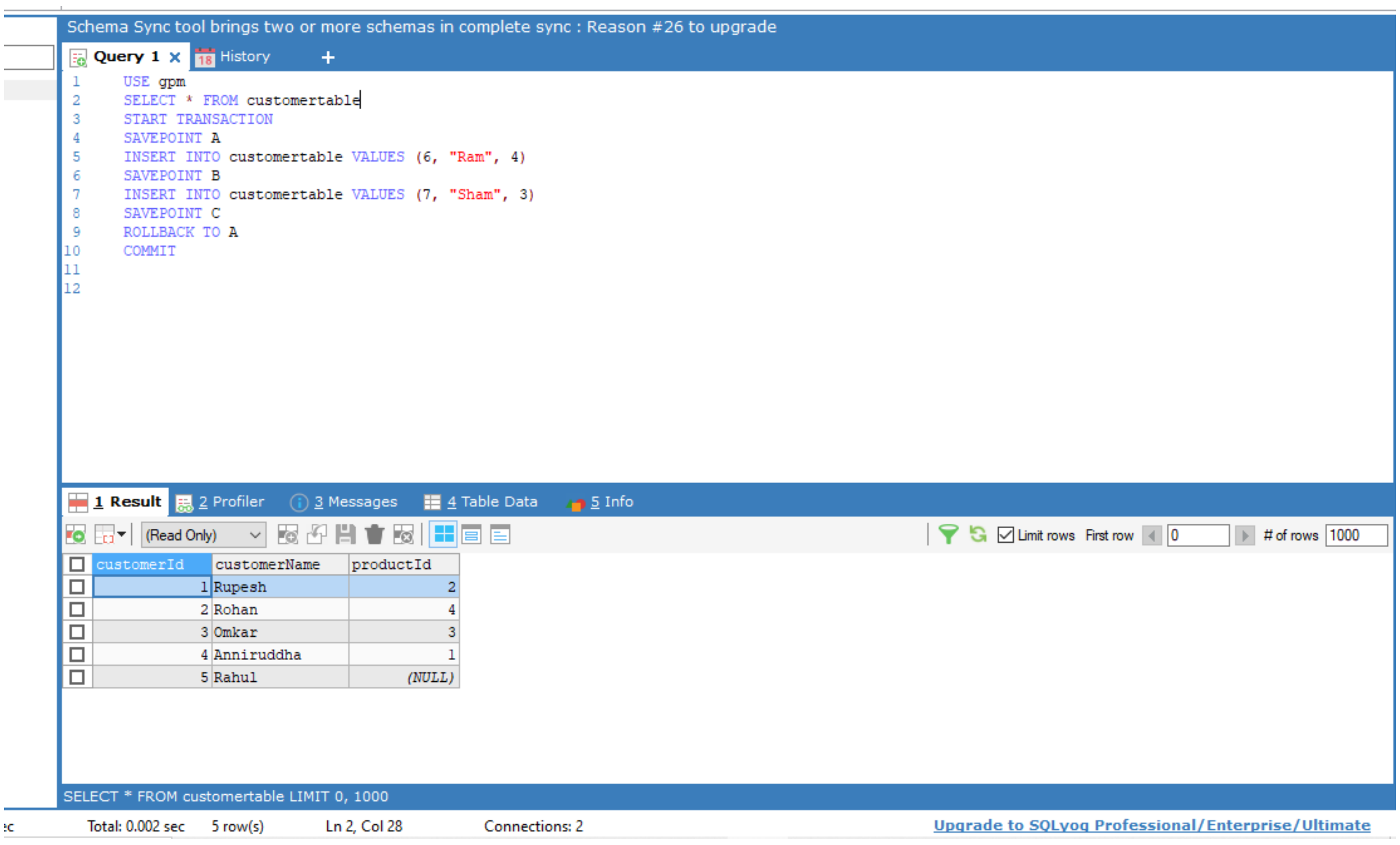
SELECT * FROM customertable LIMIT 0, 1000


Data Sync can be scheduled : Reason #29 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Total: 0.004 sec
Ln 6, Col 12
Connections: 2
Upgrade to SQLyog Professional/Enterprise/Ultimate

```sql
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

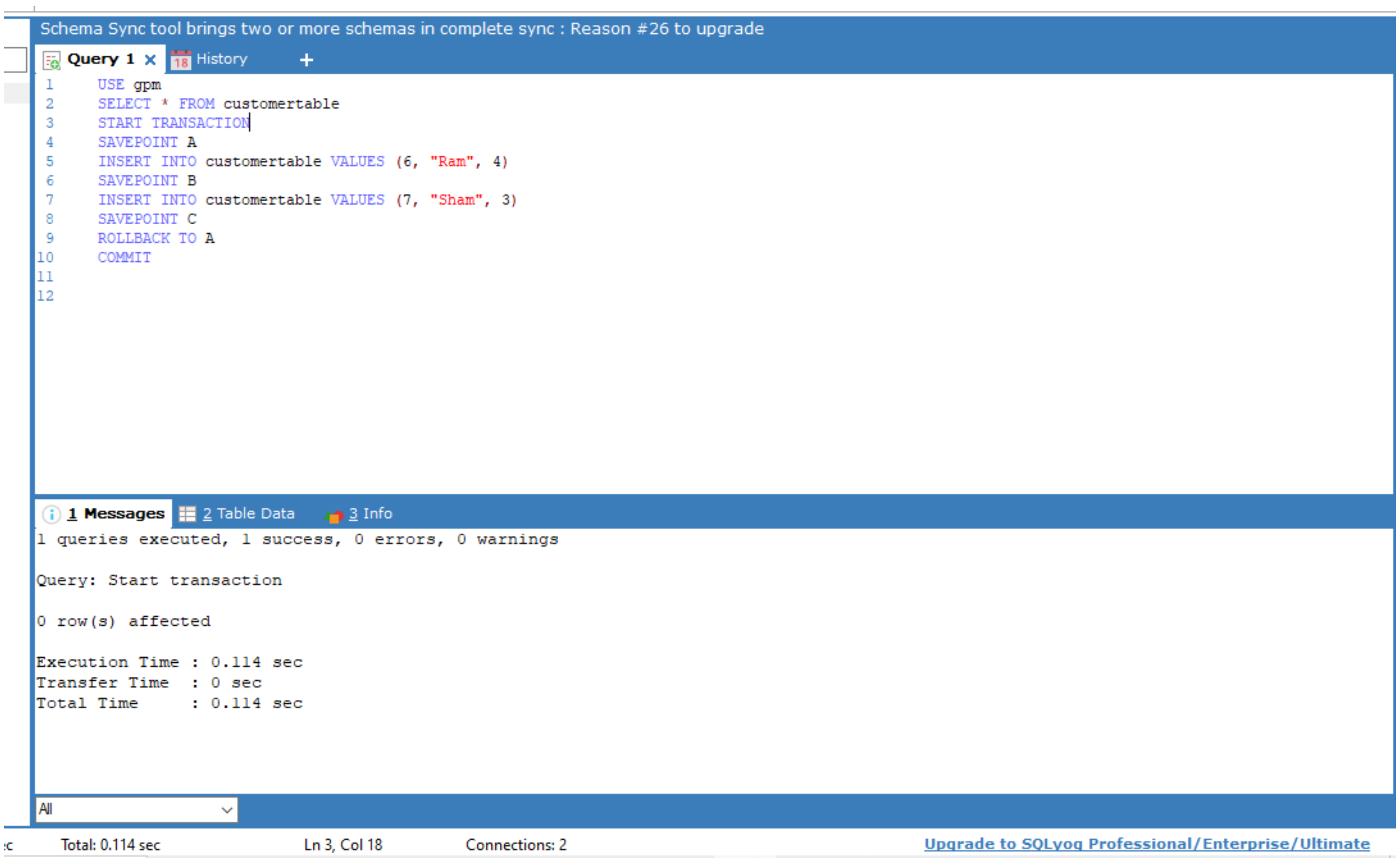
```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint B

0 row(s) affected

Execution Time : 0 sec
Transfer Time  : 0.004 sec
Total Time     : 0.004 sec
```

3) ROLLBACK


Schema Sync tool brings two or more schemas in complete sync : Reason #26 to upgrade
Query 1
History
1 Result
2 Profiler
3 Messages
4 Table Data
5 Info
(Read Only)
Limit rows
First row
0
# of rows
1000
SELECT * FROM customertable LIMIT 0, 1000
Total: 0.002 sec
5 row(s)
Ln 2, Col 28
Connections: 2
Upgrade to SQLyog Professional/Enterprise/Ultimate

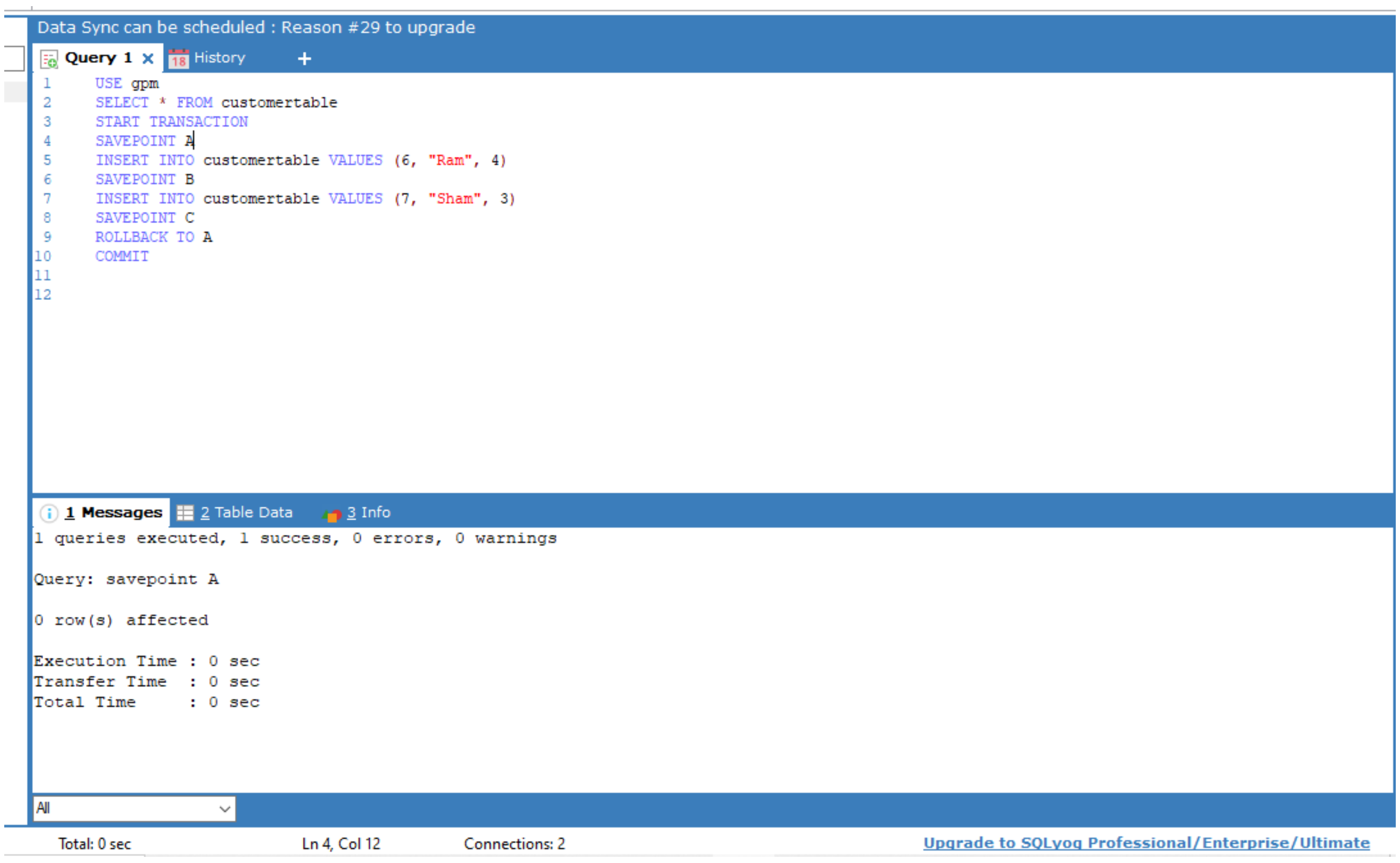
```sql
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)


Schema Sync tool brings two or more schemas in complete sync : Reason #26 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Total: 0.114 sec
Ln 3, Col 18
Connections: 2
Upgrade to SQLyog Professional/Enterprise/Ultimate

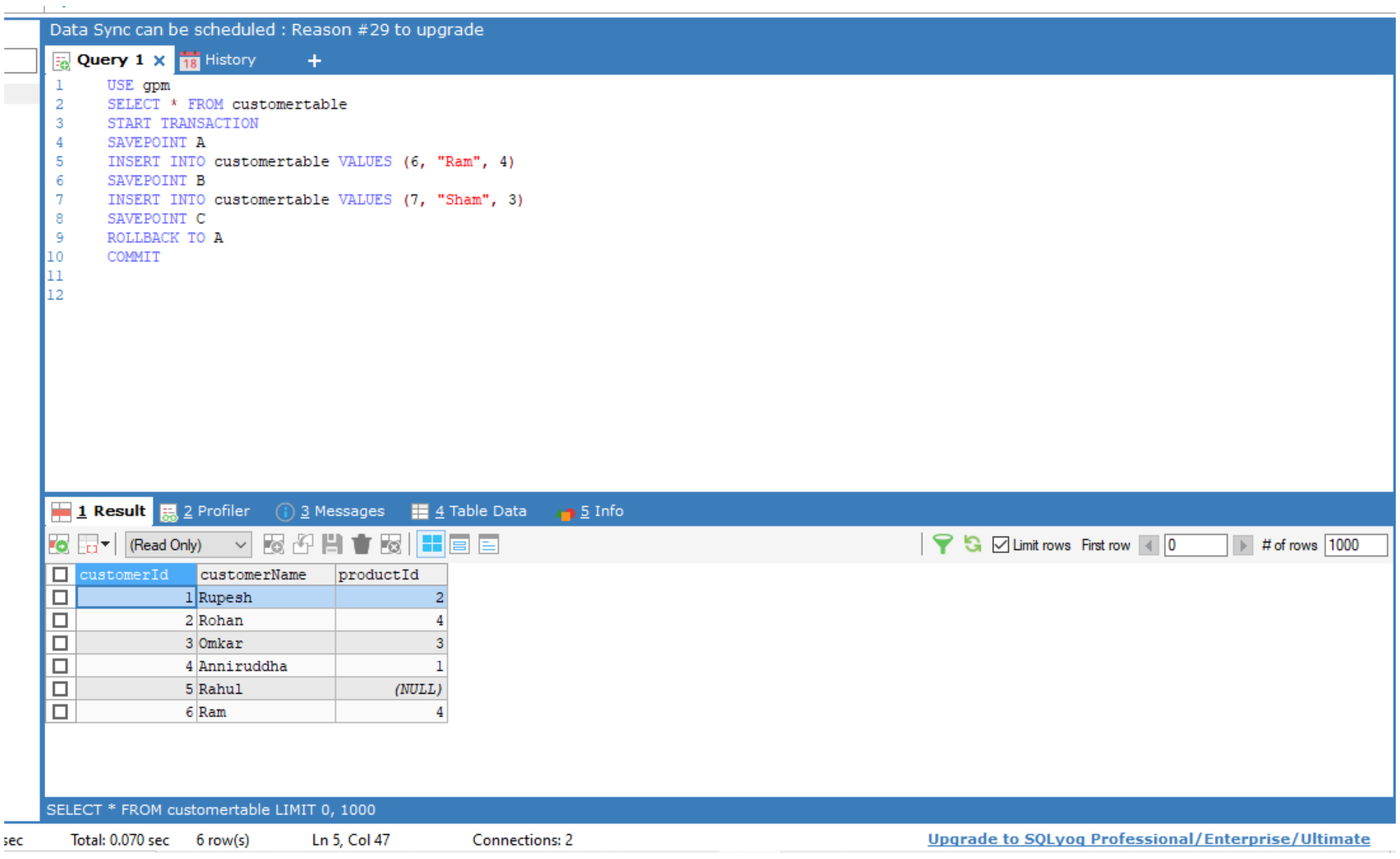
```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: Start transaction

0 row(s) affected

Execution Time : 0.114 sec
Transfer Time  : 0 sec
Total Time     : 0.114 sec
```


Data Sync can be scheduled : Reason #29 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Total: 0 sec
Ln 4, Col 12
Connections: 2
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint A

0 row(s) affected

Execution Time : 0 sec
Transfer Time  : 0 sec
Total Time     : 0 sec
```


Data Sync can be scheduled : Reason #29 to upgrade
Query 1
History
1 Result
2 Profiler
3 Messages
4 Table Data
5 Info
(Read Only)
Limit rows
First row
# of rows
Upgrade to SQLyog Professional/Enterprise/Ultimate

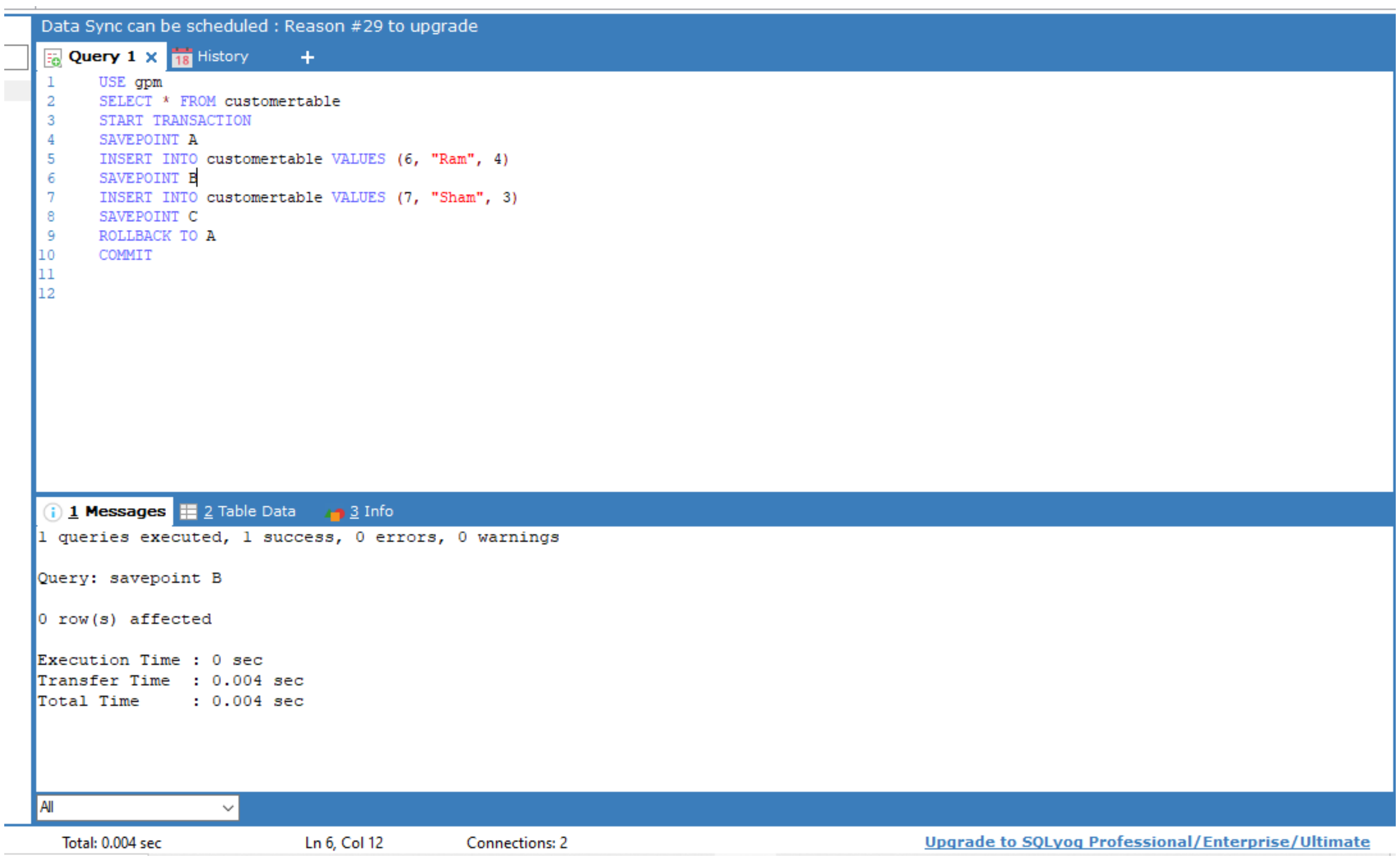
```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

First row: 0 · # of rows: 1000

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)
6	Ram	4

```
SELECT * FROM customertable LIMIT 0, 1000
```

sec Total: 0.070 sec 6 row(s) Ln 5, Col 47 Connections: 2

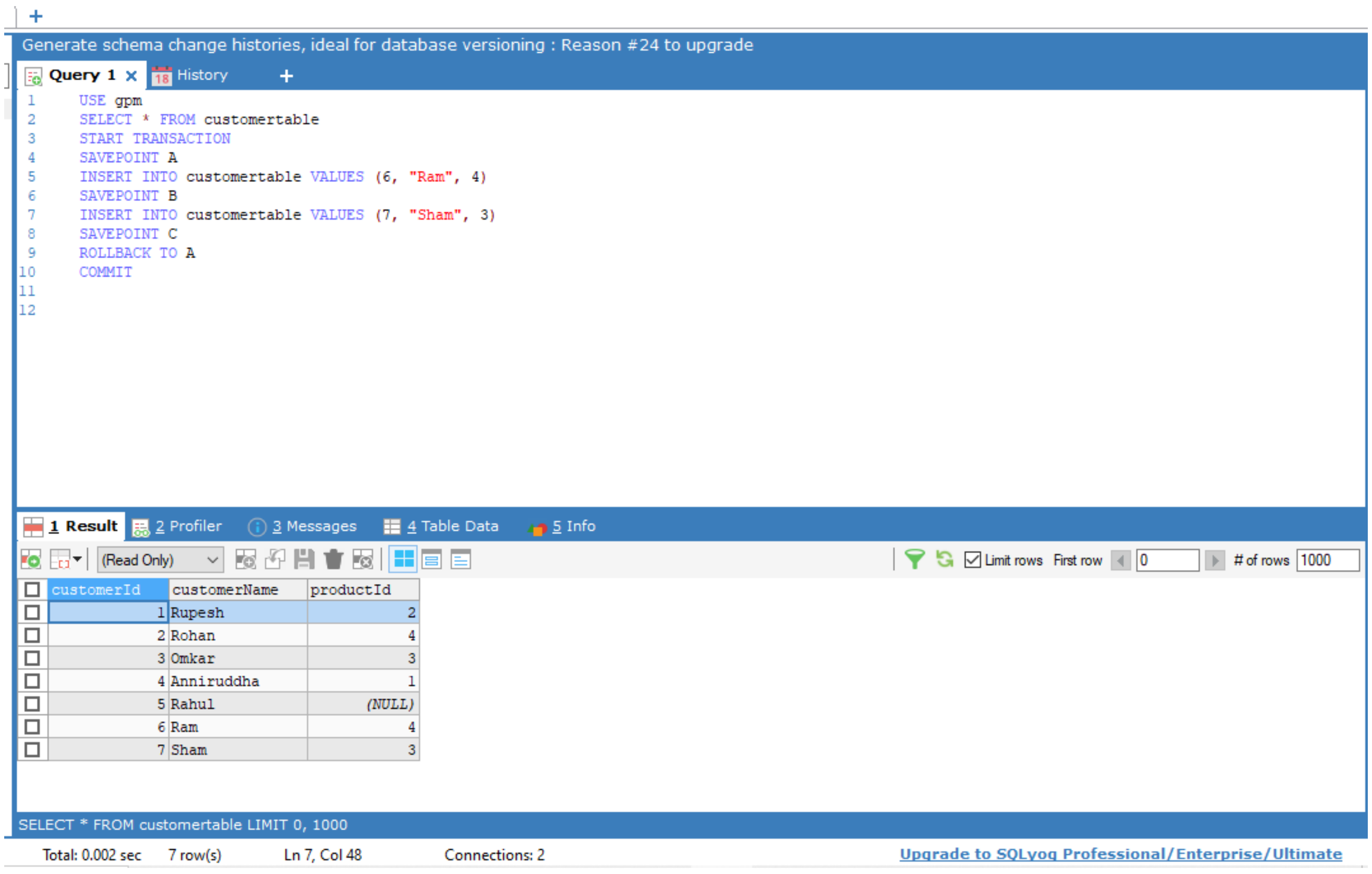

Data Sync can be scheduled : Reason #29 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint B

0 row(s) affected

Execution Time : 0 sec
Transfer Time  : 0.004 sec
Total Time     : 0.004 sec
```

Total: 0.004 sec Ln 6, Col 12 Connections: 2

Omkar Phansopkar
FS19CO042


Generate schema change histories, ideal for database versioning : Reason #24 to upgrade
Query 1
History
1 Result
2 Profiler
3 Messages
4 Table Data
5 Info
(Read Only)
Limit rows
First row
# of rows
Upgrade to SQLyog Professional/Enterprise/Ultimate

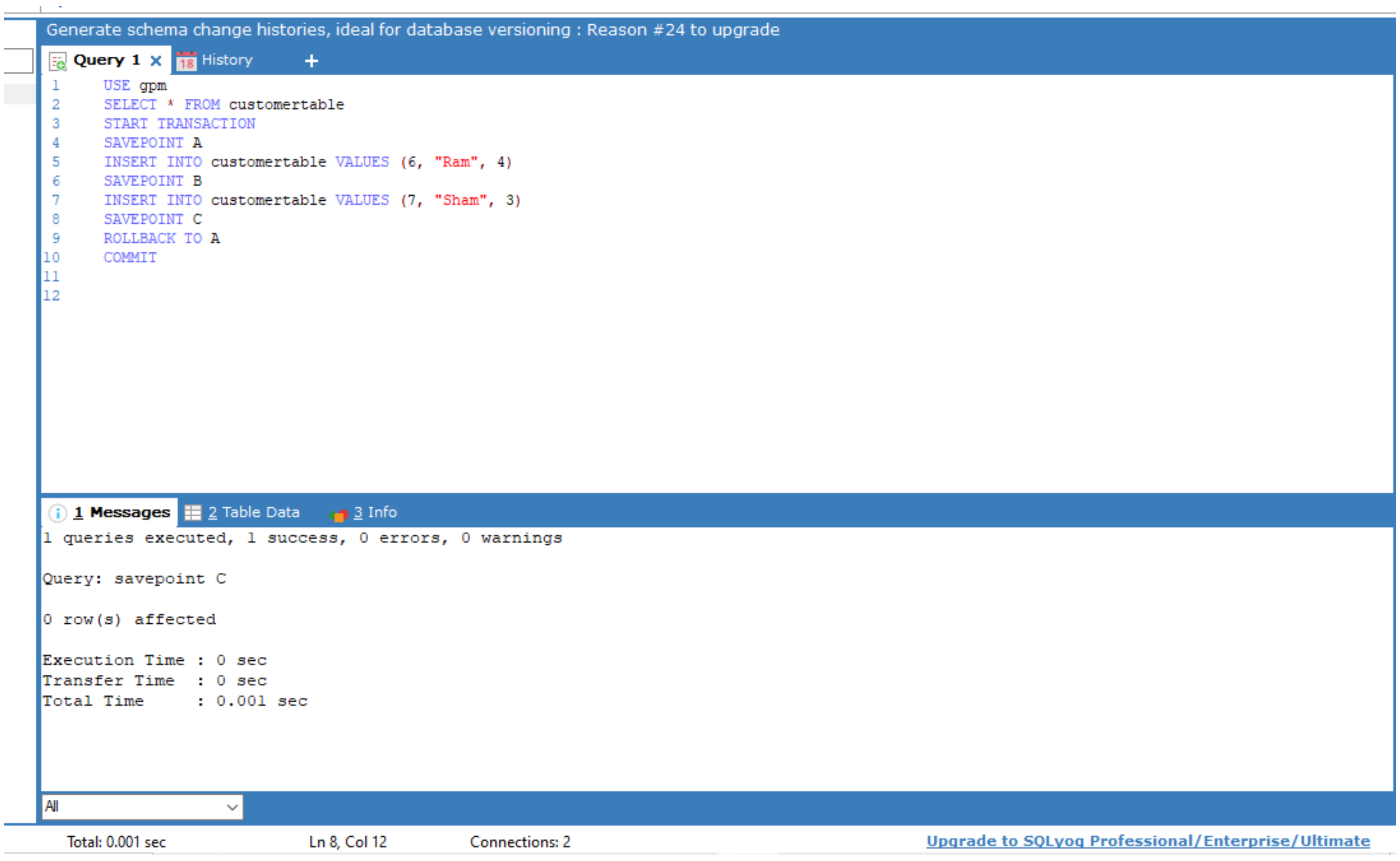
```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

☑ Limit rows First row 0 # of rows 1000

	customerId	customerName	productId
□	1	Rupesh	2
□	2	Rohan	4
□	3	Omkar	3
□	4	Anniruddha	1
□	5	Rahul	(NULL)
□	6	Ram	4
□	7	Sham	3

SELECT * FROM customertable LIMIT 0, 1000

Total: 0.002 sec 7 row(s) Ln 7, Col 48 Connections: 2

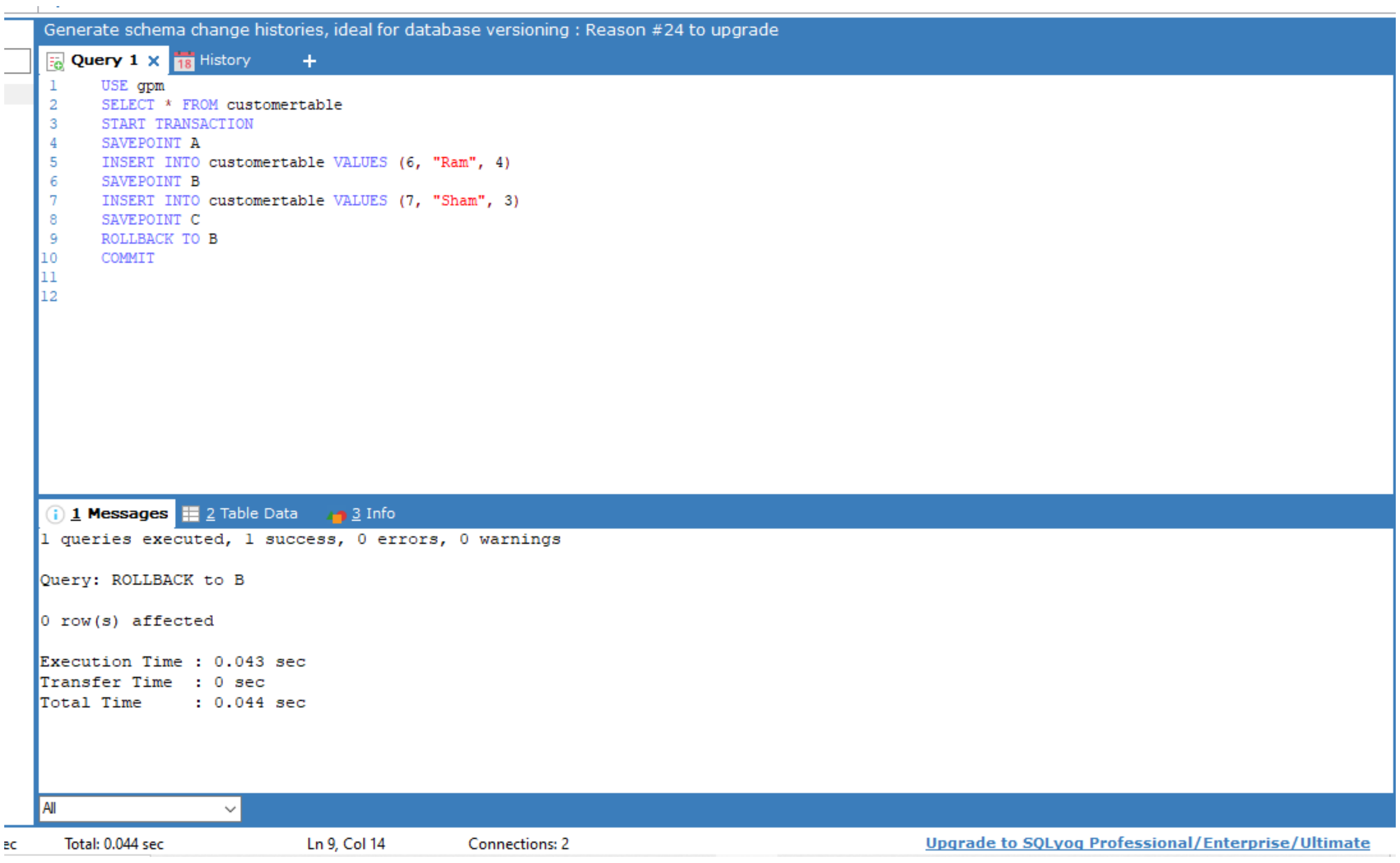

Generate schema change histories, ideal for database versioning : Reason #24 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint C

0 row(s) affected

Execution Time : 0 sec
Transfer Time  : 0 sec
Total Time     : 0.001 sec
```

Total: 0.001 sec Ln 8, Col 12 Connections: 2

Omkar Phansopkar
FS19CO042

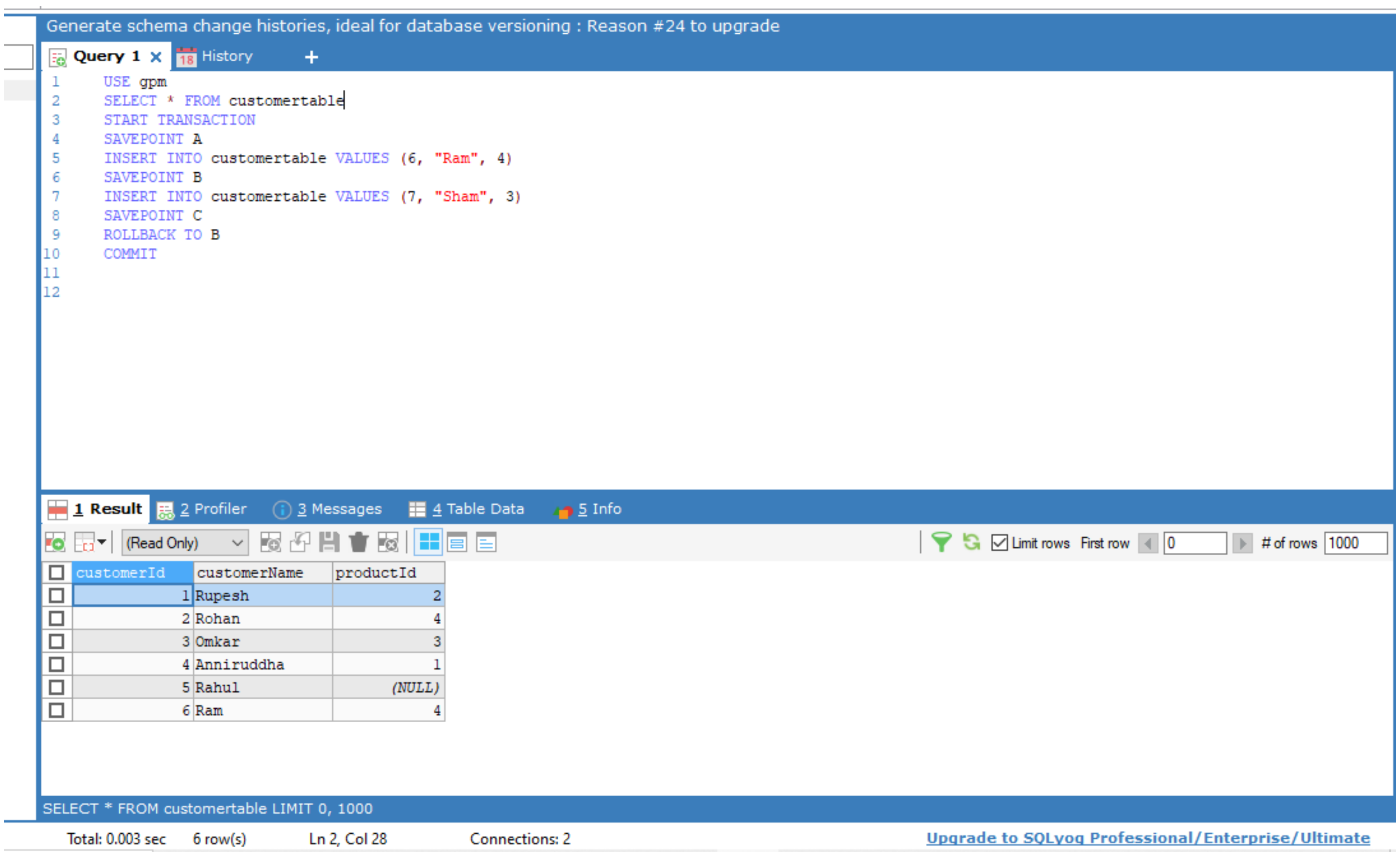


Generate schema change histories, ideal for database versioning : Reason #24 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO B
COMMIT
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: ROLLBACK to B

0 row(s) affected

Execution Time : 0.043 sec
Transfer Time  : 0 sec
Total Time     : 0.044 sec
```

Total: 0.044 sec   Ln 9, Col 14   Connections: 2



Generate schema change histories, ideal for database versioning : Reason #24 to upgrade
Query 1
History
1 Result
2 Profiler
3 Messages
4 Table Data
5 Info
(Read Only)
Limit rows
First row
# of rows
Upgrade to SQLyog Professional/Enterprise/Ultimate

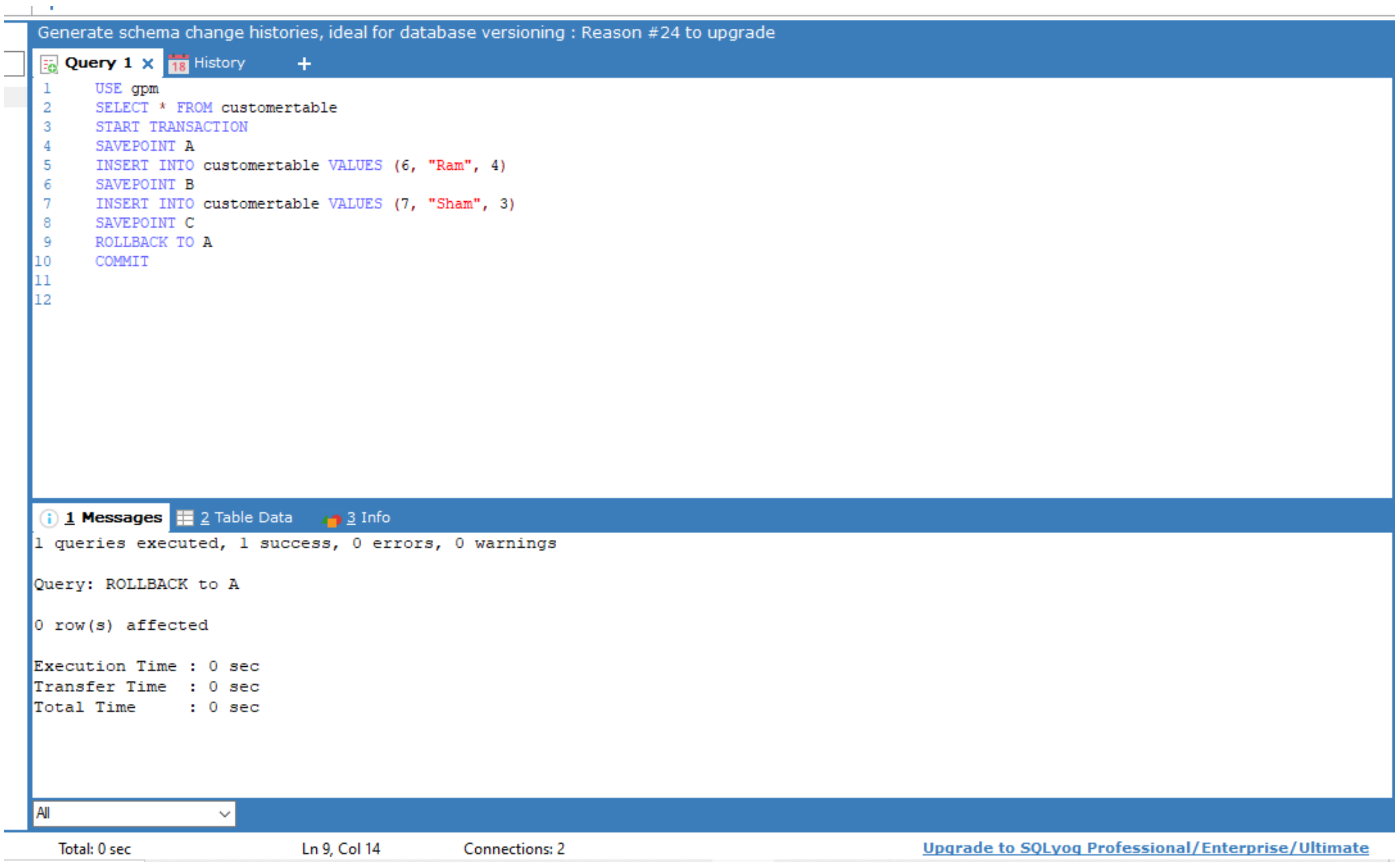
```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO B
COMMIT
```

☑ Limit rows   First row 0   # of rows 1000

□	customerId	customerName	productId
□	1	Rupesh	2
□	2	Rohan	4
□	3	Omkar	3
□	4	Anniruddha	1
□	5	Rahul	(NULL)
□	6	Ram	4

SELECT * FROM customertable LIMIT 0, 1000

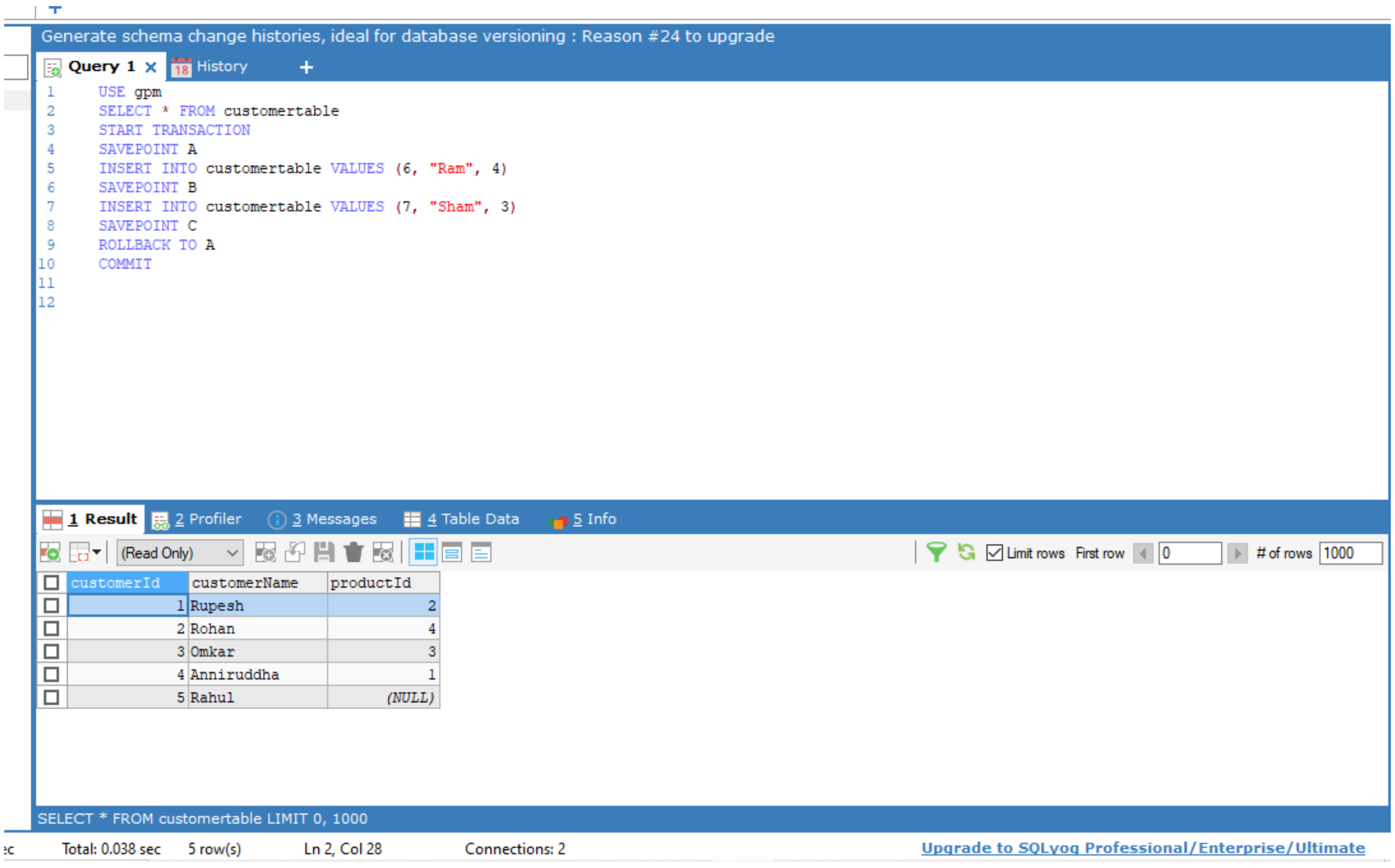
Total: 0.003 sec   6 row(s)   Ln 2, Col 28   Connections: 2


Generate schema change histories, ideal for database versioning : Reason #24 to upgrade
Query 1
History
1 Messages
2 Table Data
3 Info
All
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

```
1 queries executed, 1 success, 0 errors, 0 warnings

Query: ROLLBACK to A

0 row(s) affected

Execution Time : 0 sec
Transfer Time  : 0 sec
Total Time     : 0 sec
```

Total: 0 sec   Ln 9, Col 14   Connections: 2


Generate schema change histories, ideal for database versioning : Reason #24 to upgrade
Query 1
History
1 Result
2 Profiler
3 Messages
4 Table Data
5 Info
(Read Only)
Limit rows
First row
# of rows
Upgrade to SQLyog Professional/Enterprise/Ultimate

```
USE gpm
SELECT * FROM customertable
START TRANSACTION
SAVEPOINT A
INSERT INTO customertable VALUES (6, "Ram", 4)
SAVEPOINT B
INSERT INTO customertable VALUES (7, "Sham", 3)
SAVEPOINT C
ROLLBACK TO A
COMMIT
```

☑ Limit rows First row 0 # of rows 1000

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)

SELECT * FROM customertable LIMIT 0, 1000

Total: 0.038 sec   5 row(s)   Ln 2, Col 28   Connections: 2

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

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;
```

2. 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;
```

3. 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;
```

4. 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;
```

5. 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;
```

6. Nested Loops

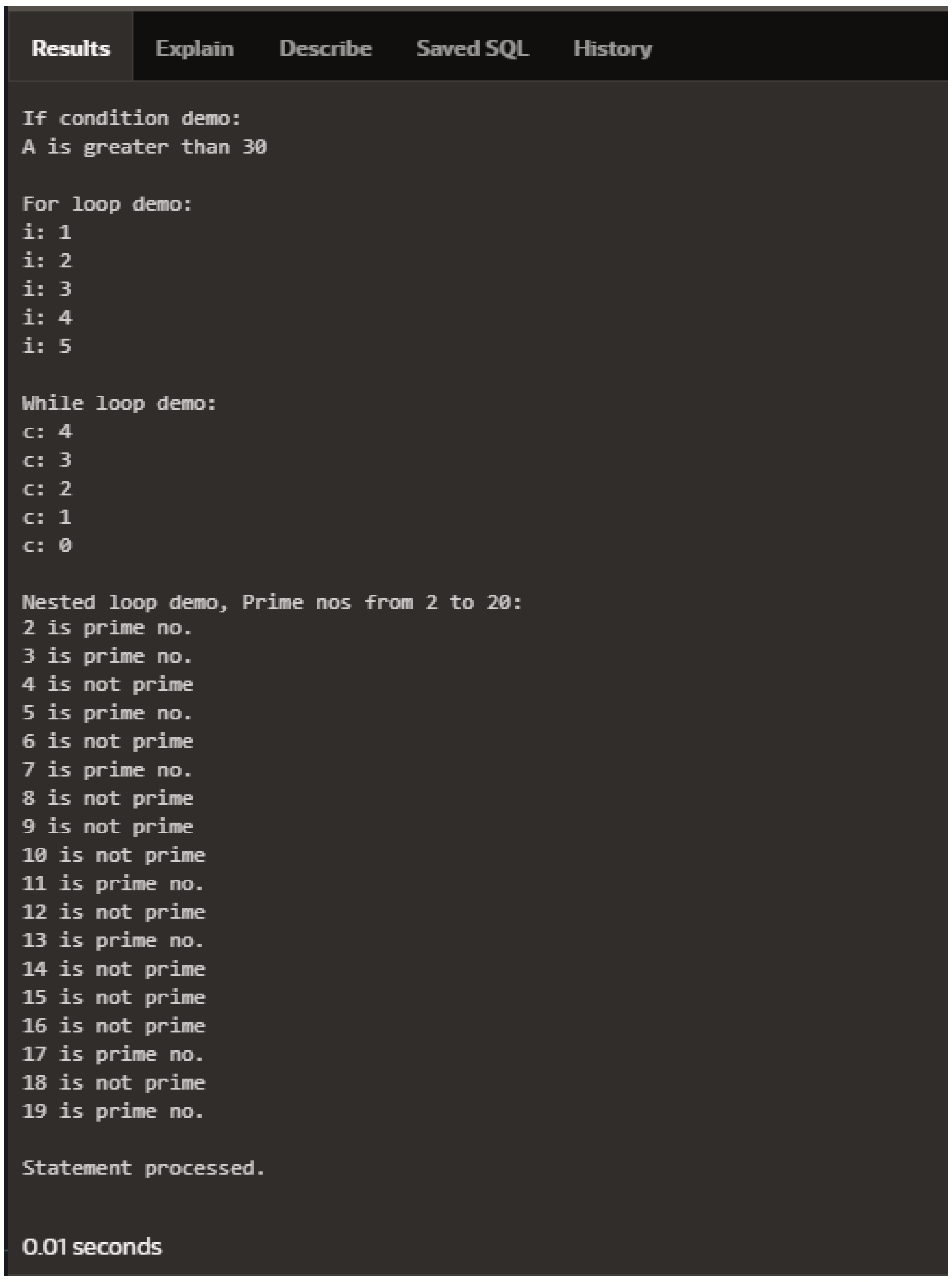
PL/SQL allows using one loop inside another loop. Following section shows a few examples to illustrate the concept.

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;
```



```
If condition demo:
A is greater than 30

For loop demo:
i: 1
i: 2
i: 3
i: 4
i: 5

While loop demo:
c: 4
c: 3
c: 2
c: 1
c: 0

Nested loop demo, Prime nos from 2 to 20:
2 is prime no.
3 is prime no.
4 is not prime
5 is prime no.
6 is not prime
7 is prime no.
8 is not prime
9 is not prime
10 is not prime
11 is prime no.
12 is not prime
13 is prime no.
14 is not prime
15 is not prime
16 is not prime
17 is prime no.
18 is not prime
19 is prime no.

Statement processed.
```

0.01 seconds

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

Practical no. 10

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.

2. 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.

3. 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.

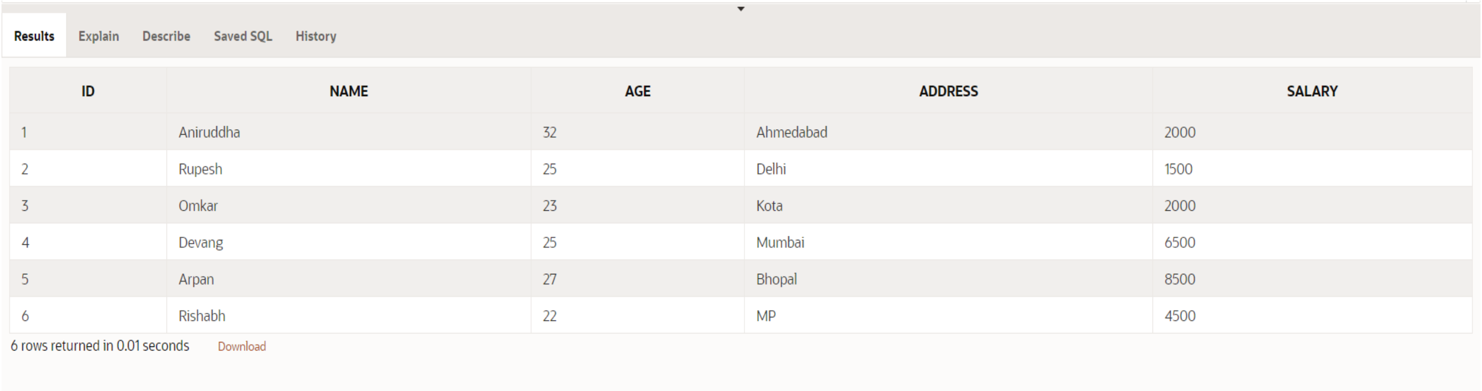
Syntax :

CURSOR cursor_name IS select_statement;

Output :

1. Implicit Cursor :

CUSTOMERS table :



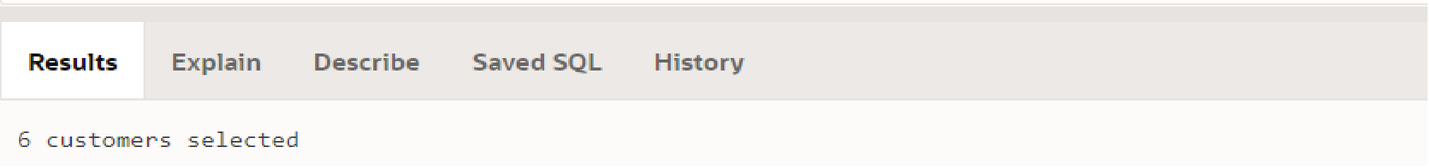
ID	NAME	AGE	ADDRESS	SALARY
1	Aniruddha	32	Ahmedabad	2000
2	Rupesh	25	Delhi	1500
3	Omkar	23	Kota	2000
4	Devang	25	Mumbai	6500
5	Arpan	27	Bhopal	8500
6	Rishabh	22	MP	4500

6 rows returned in 0.01 seconds Download

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;
```

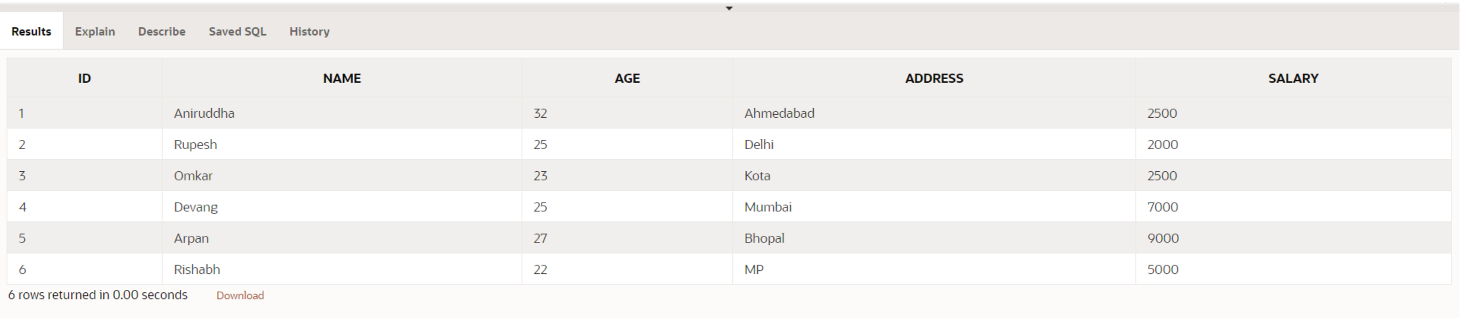
When the above code is executed at the SQL prompt, it produces the following result:



```
6 customers selected
```

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

Customers Table :



ID	NAME	AGE	ADDRESS	SALARY
1	Aniruddha	32	Ahmedabad	2500
2	Rupesh	25	Delhi	2000
3	Omkar	23	Kota	2500
4	Devang	25	Mumbai	7000
5	Arpan	27	Bhopal	9000
6	Rishabh	22	MP	5000

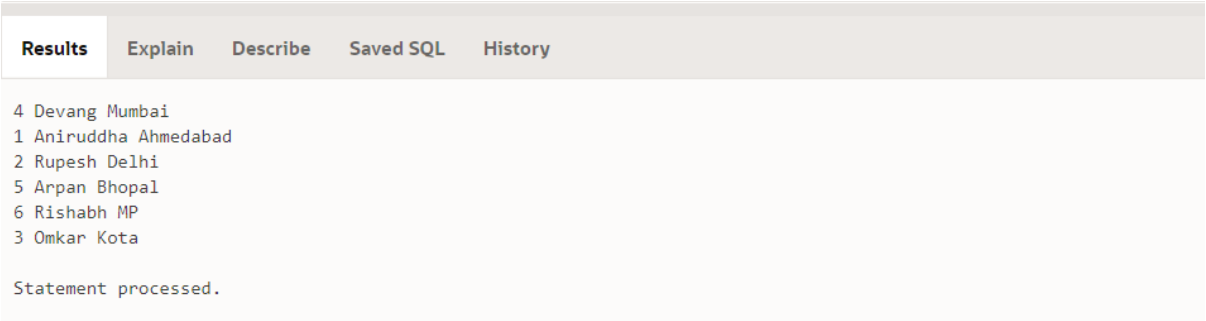
6 rows returned in 0.00 seconds

2. 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



```
4 Devang Mumbai
1 Aniruddha Ahmedabad
2 Rupesh Delhi
5 Arpan Bhopal
6 Rishabh MP
3 Omkar Kota

Statement processed.
```

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

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.
2. **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;
```

2. 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.

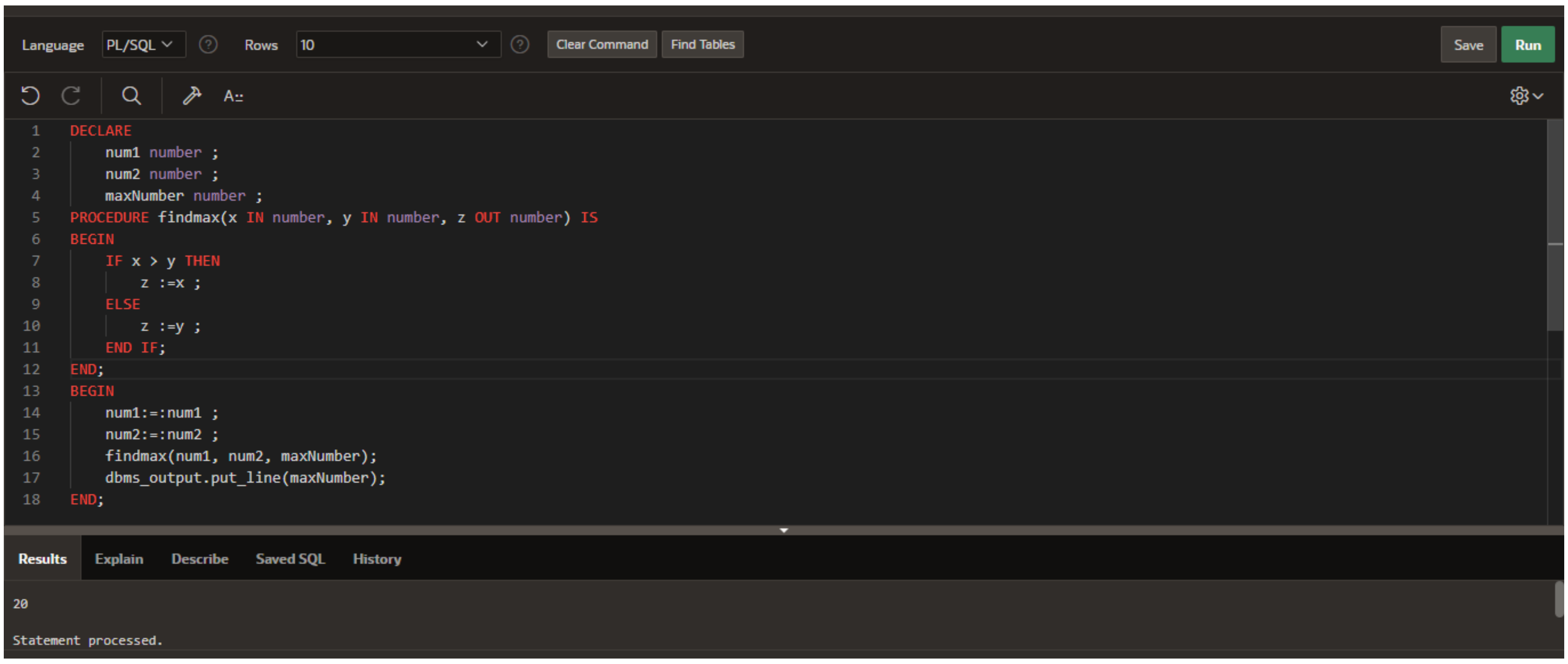
The AS keyword is used instead of the IS keyword for creating a standalone function

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 :**

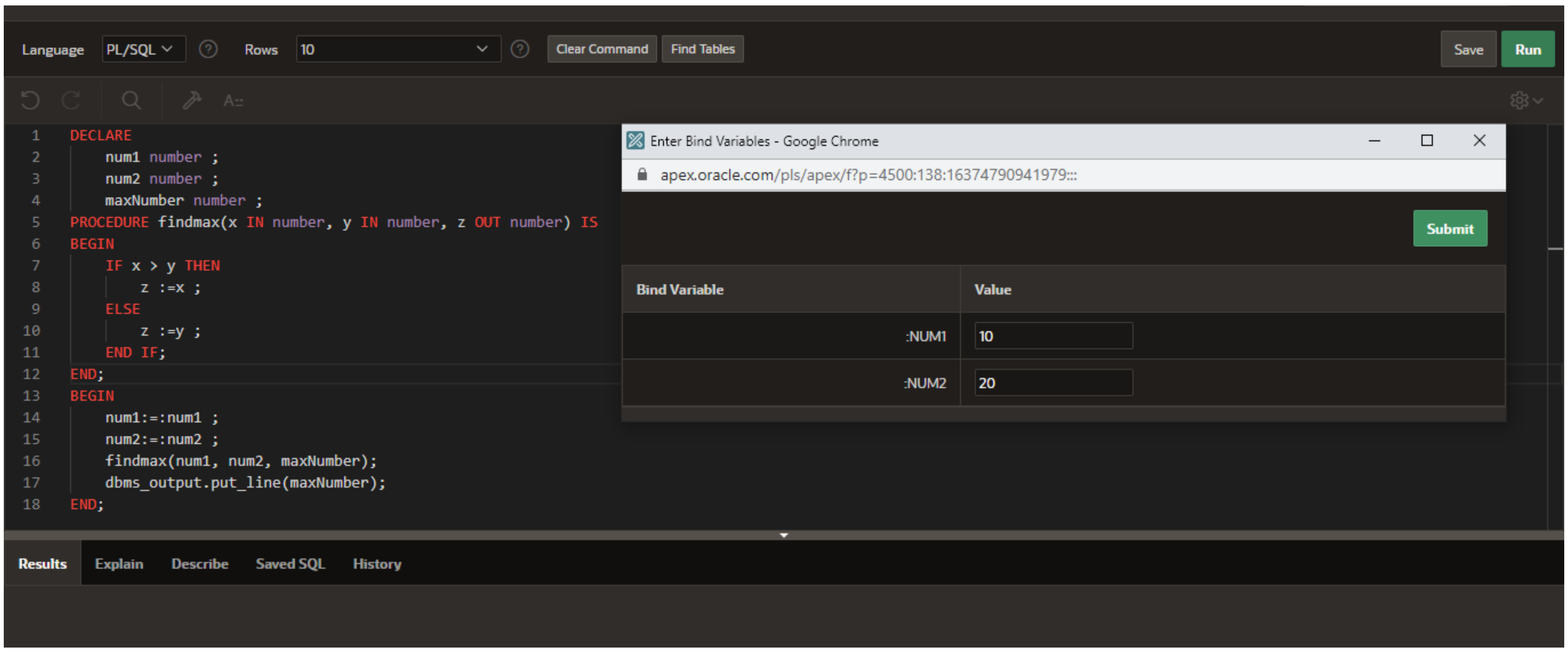


Language PL/SQL

Rows 10

```
DECLARE
    num1 number ;
    num2 number ;
    maxNumber number ;
PROCEDURE findmax(x IN number, y IN number, z OUT number) IS
BEGIN
    IF x > y THEN
        z :=x ;
    ELSE
        z :=y ;
    END IF;
END;
BEGIN
    num1:=:num1 ;
    num2:=:num2 ;
    findmax(num1, num2, maxNumber);
    dbms_output.put_line(maxNumber);
END;
```

```
20

Statement processed.
```



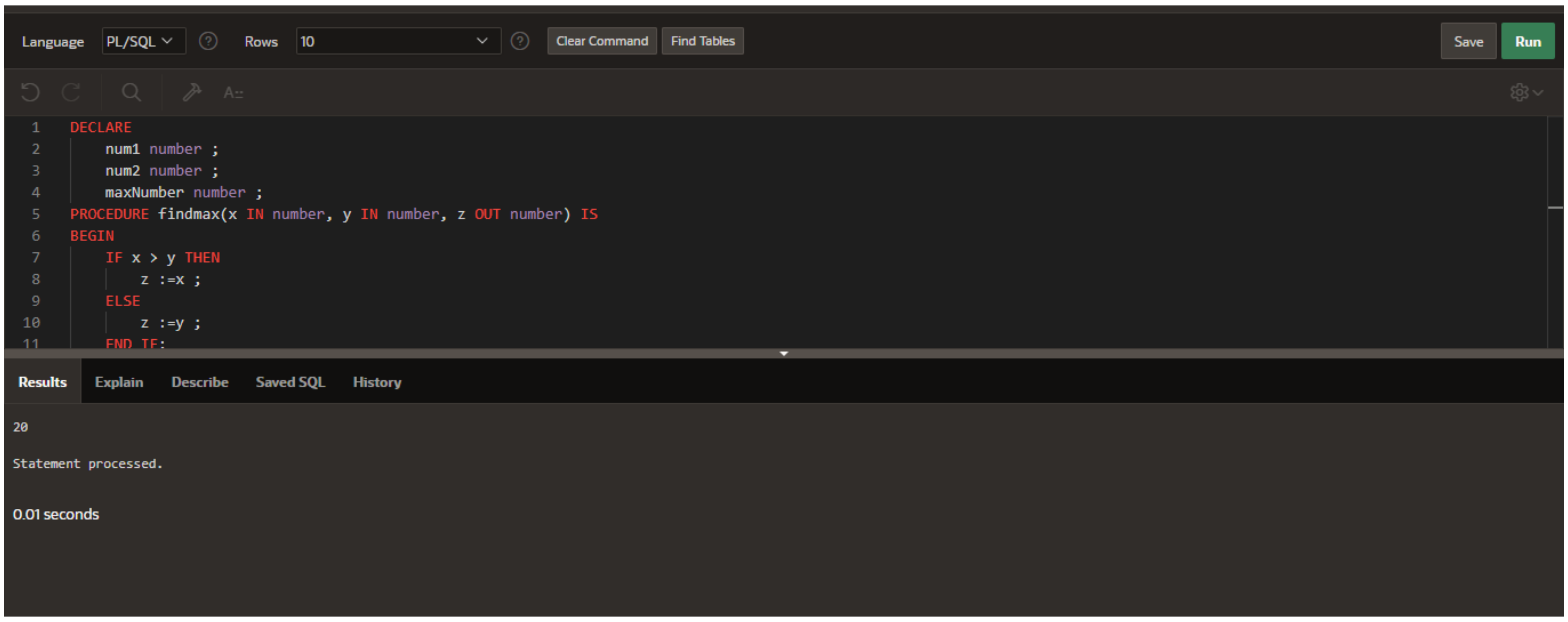
```
DECLARE
    num1 number ;
    num2 number ;
    maxNumber number ;
PROCEDURE findmax(x IN number, y IN number, z OUT number) IS
BEGIN
    IF x > y THEN
        z :=x ;
    ELSE
        z :=y ;
    END IF;
END;
BEGIN
    num1:=:num1 ;
    num2:=:num2 ;
    findmax(num1, num2, maxNumber);
    dbms_output.put_line(maxNumber);
END;
```

Enter Bind Variables - Google Chrome

apex.oracle.com/pls/apex/f?p=4500:138:16374790941979:::

Submit

Bind Variable	Value
:NUM1	10
:NUM2	20

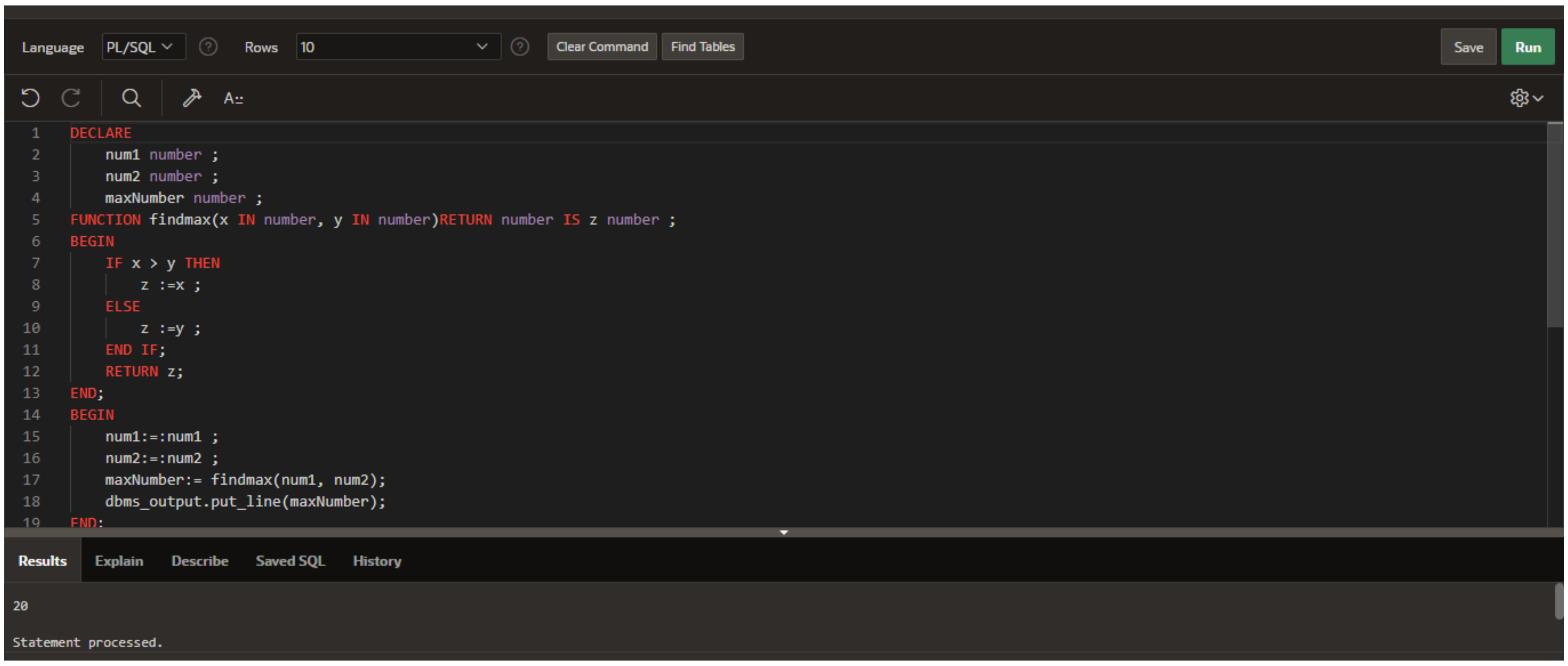


```
DECLARE
    num1 number ;
    num2 number ;
    maxNumber number ;
PROCEDURE findmax(x IN number, y IN number, z OUT number) IS
BEGIN
    IF x > y THEN
        z :=x ;
    ELSE
        z :=y ;
    END IF;
```

```
20

Statement processed.
```

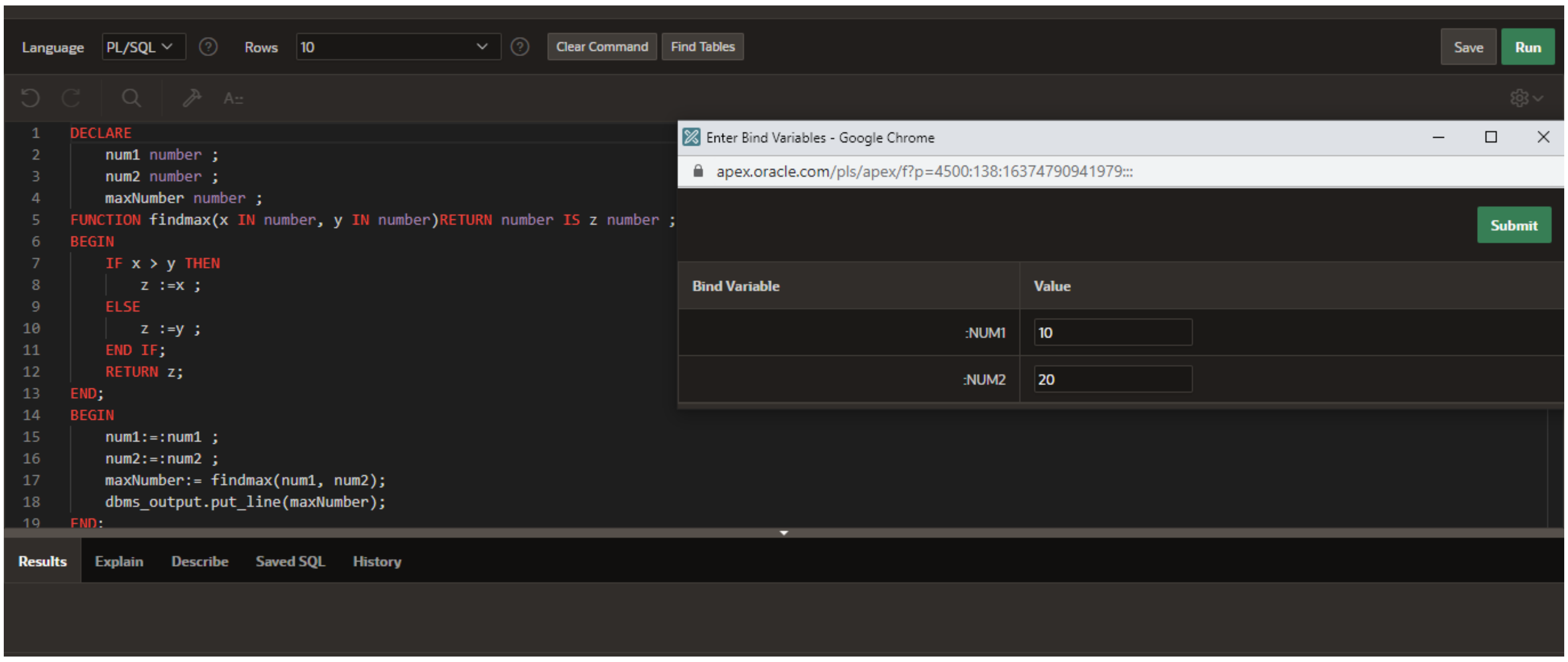
0.01 seconds

2. Function :



```
DECLARE
    num1 number ;
    num2 number ;
    maxNumber number ;
FUNCTION findmax(x IN number, y IN number)RETURN number IS z number ;
BEGIN
    IF x > y THEN
        z :=x ;
    ELSE
        z :=y ;
    END IF;
    RETURN z;
END;
BEGIN
    num1:=:num1 ;
    num2:=:num2 ;
    maxNumber:= findmax(num1, num2);
    dbms_output.put_line(maxNumber);
END;
```

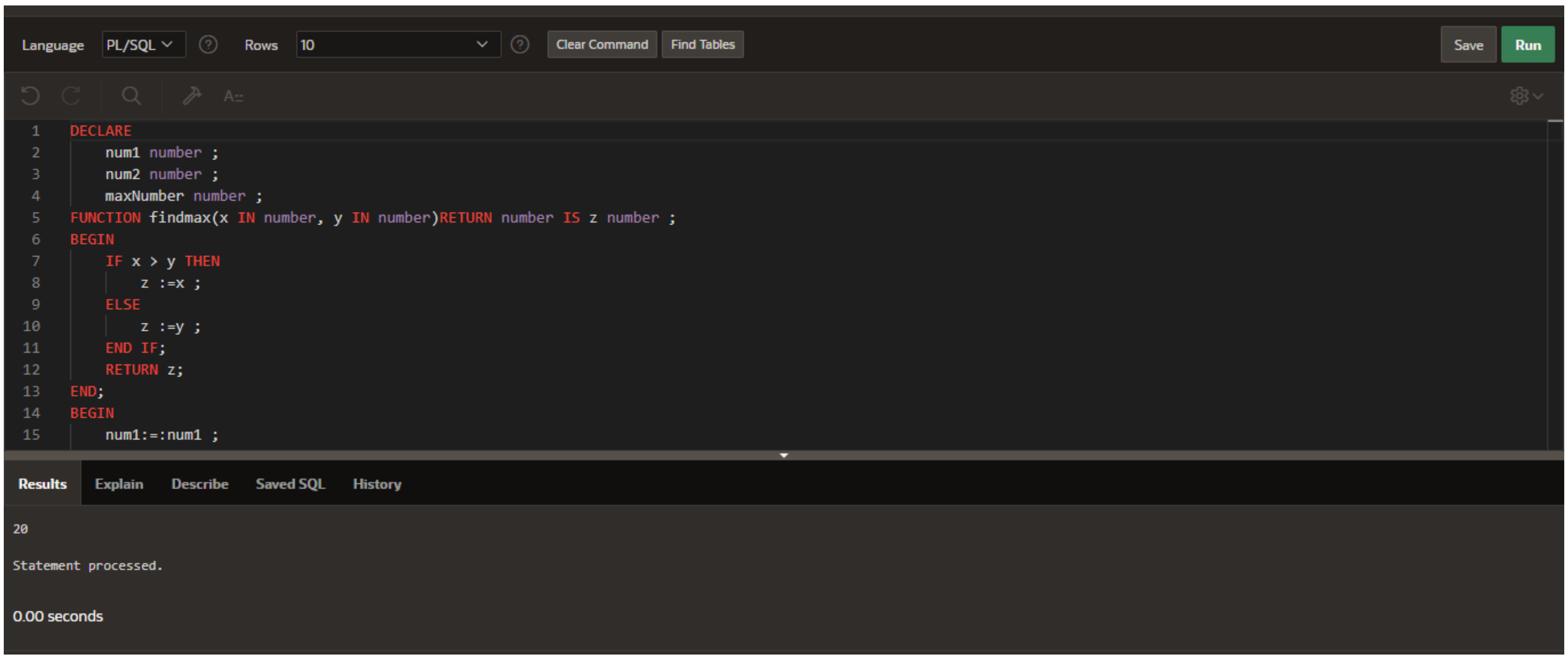
```
20

Statement processed.
```



```
DECLARE
    num1 number ;
    num2 number ;
    maxNumber number ;
FUNCTION findmax(x IN number, y IN number)RETURN number IS z number ;
BEGIN
    IF x > y THEN
        z :=x ;
    ELSE
        z :=y ;
    END IF;
    RETURN z;
END;
BEGIN
    num1:=:num1 ;
    num2:=:num2 ;
    maxNumber:= findmax(num1, num2);
    dbms_output.put_line(maxNumber);
END;
```

Bind Variable	Value
:NUM1	10
:NUM2	20

Omkar Phansopkar
FS19CO042



Language PL/SQL

Rows 10

```
DECLARE
    num1 number ;
    num2 number ;
    maxNumber number ;
FUNCTION findmax(x IN number, y IN number)RETURN number IS z number ;
BEGIN
    IF x > y THEN
        z :=x ;
    ELSE
        z :=y ;
    END IF;
    RETURN z;
END;
BEGIN
    num1:=:num1 ;
```

```
20

Statement processed.

0.00 seconds
```

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

Practical no. 12

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

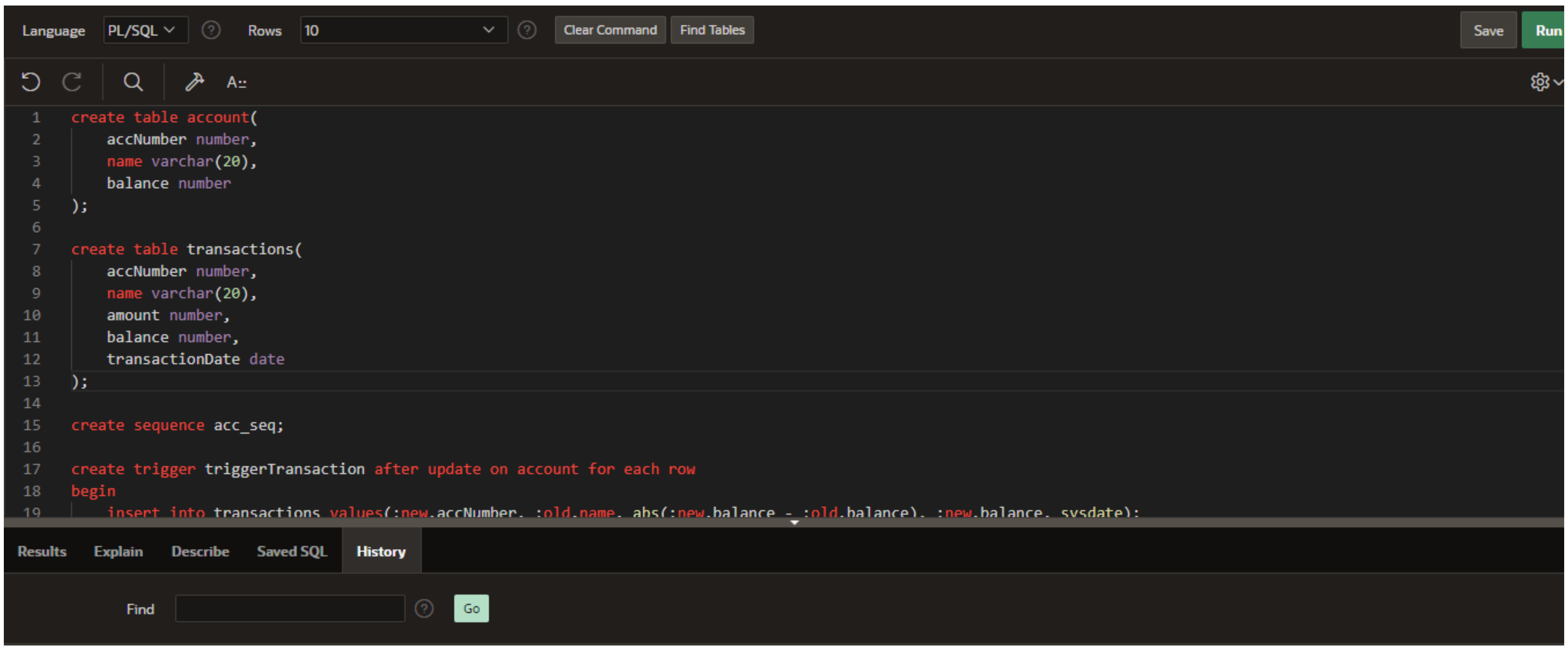
Triggers can be written for the following purposes –

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**

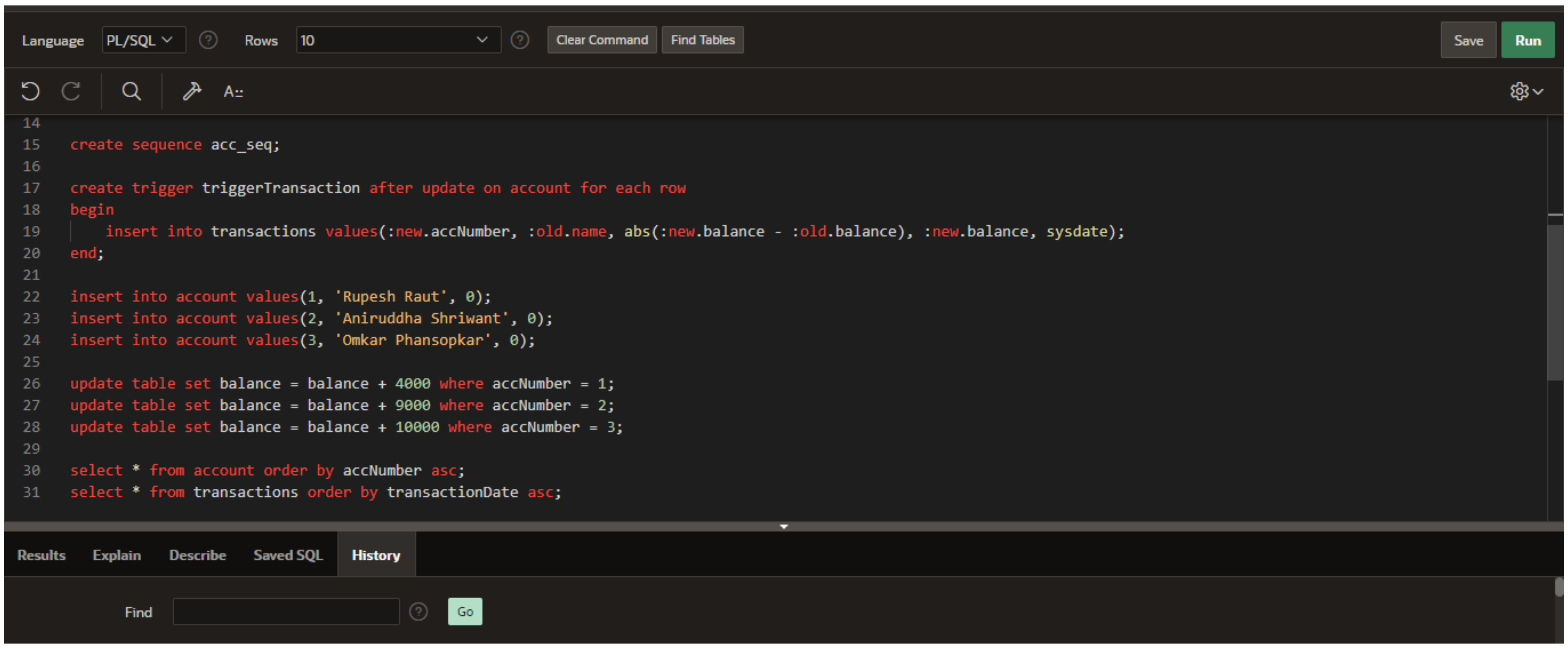
Syntax :

```
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;
```

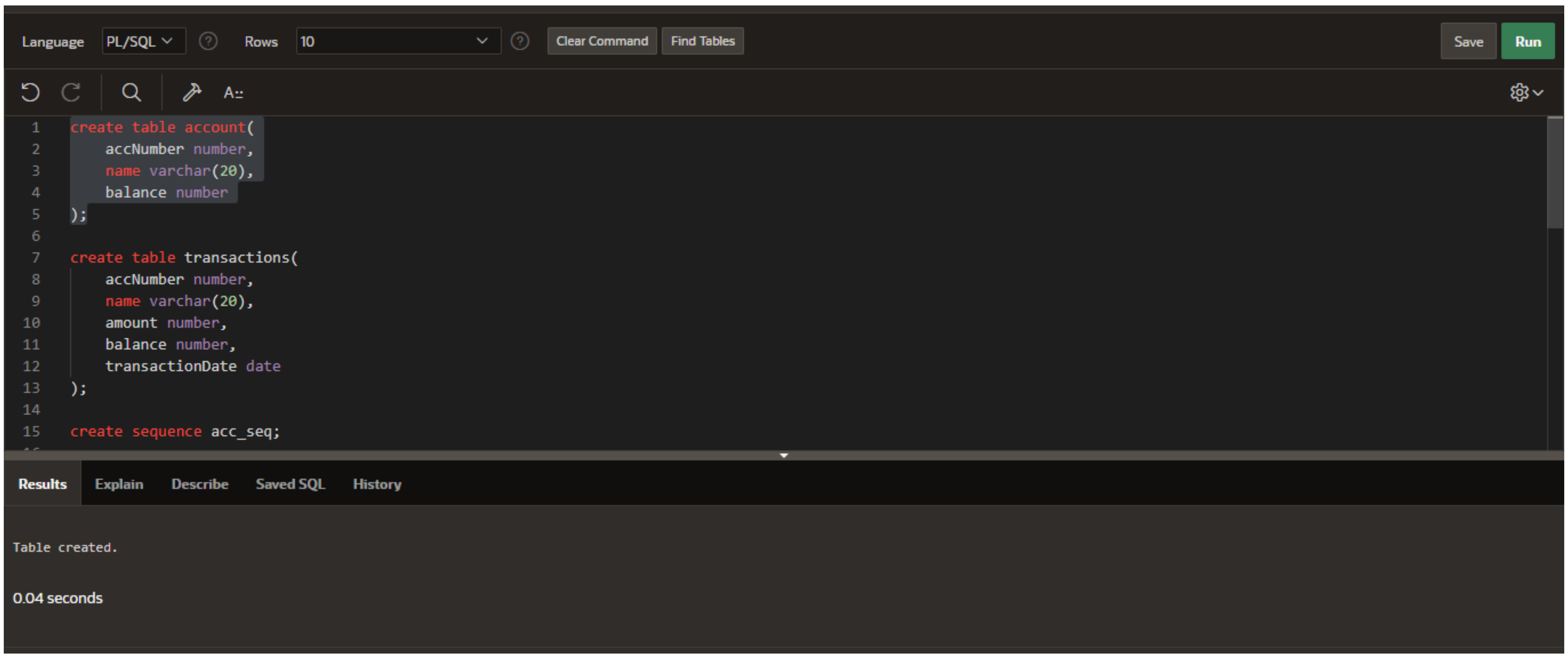
Output :



```
create table account(
    accNumber number,
    name varchar(20),
    balance number
);

create table transactions(
    accNumber number,
    name varchar(20),
    amount number,
    balance number,
    transactionDate date
);

create sequence acc_seq;

create trigger triggerTransaction after update on account for each row
begin
    insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
```



```

create sequence acc_seq;

create trigger triggerTransaction after update on account for each row
begin
    insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update table set balance = balance + 4000 where accNumber = 1;
update table set balance = balance + 9000 where accNumber = 2;
update table set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```



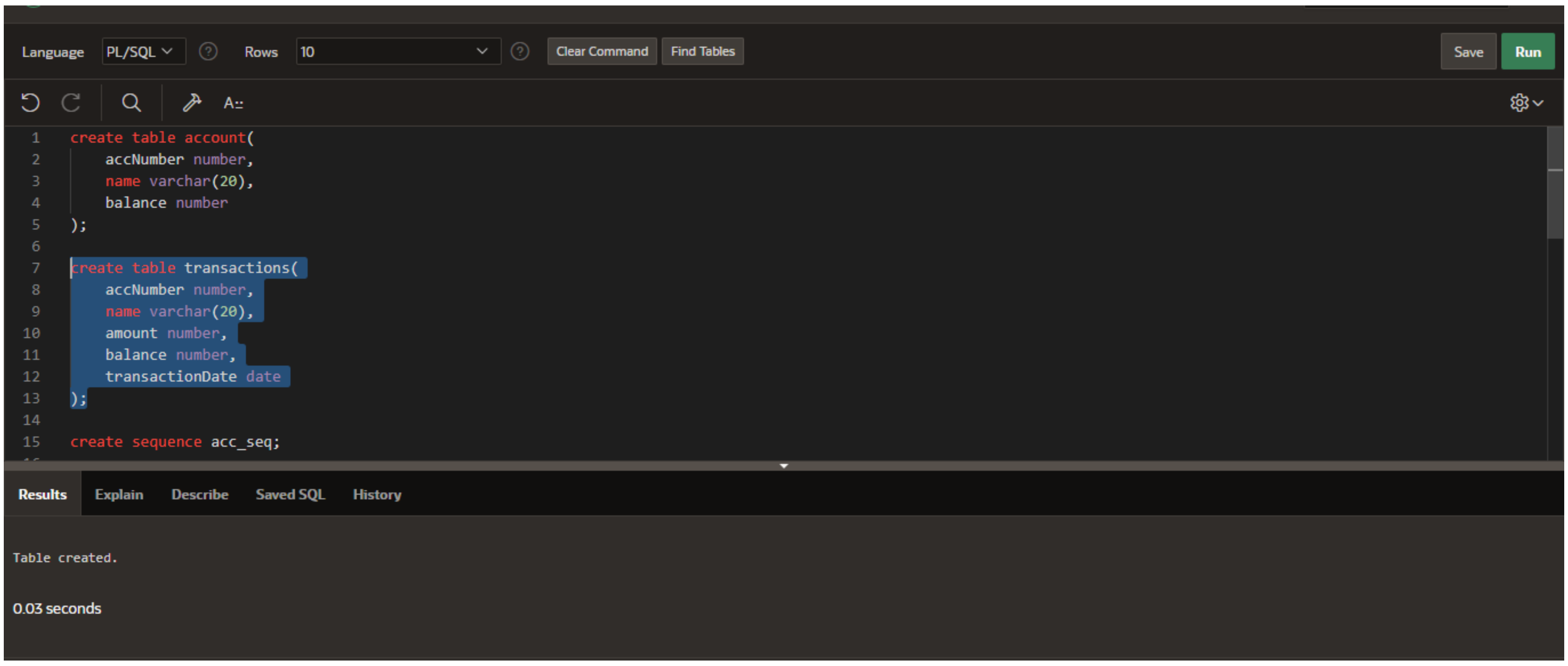
Language PL/SQL

Rows 10

```
create table account(
    accNumber number,
    name varchar(20),
    balance number
);

create table transactions(
    accNumber number,
    name varchar(20),
    amount number,
    balance number,
    transactionDate date
);

create sequence acc_seq;
```

```
Table created.
```

0.04 seconds



Language PL/SQL

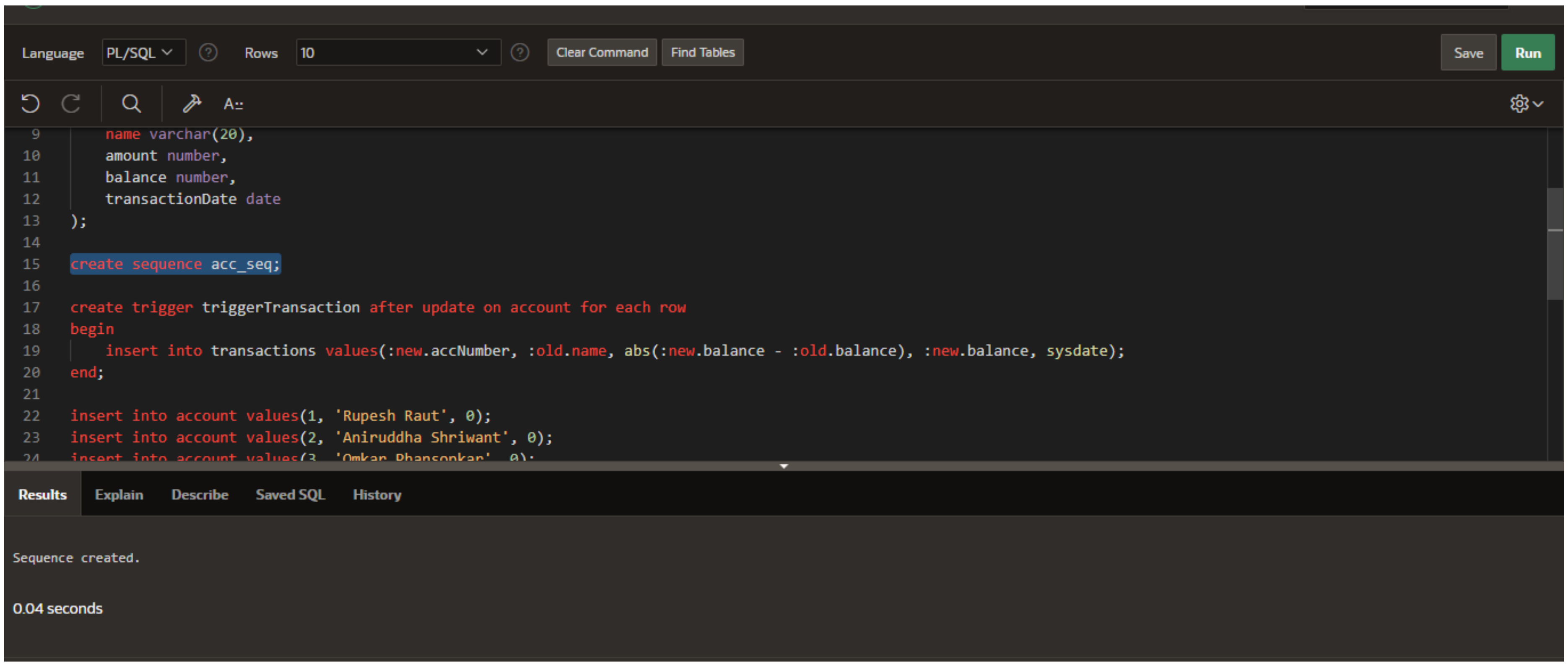
Rows 10

```
create table account(
    accNumber number,
    name varchar(20),
    balance number
);

create table transactions(
    accNumber number,
    name varchar(20),
    amount number,
    balance number,
    transactionDate date
);

create sequence acc_seq;
```

```
Table created.
```

0.03 seconds

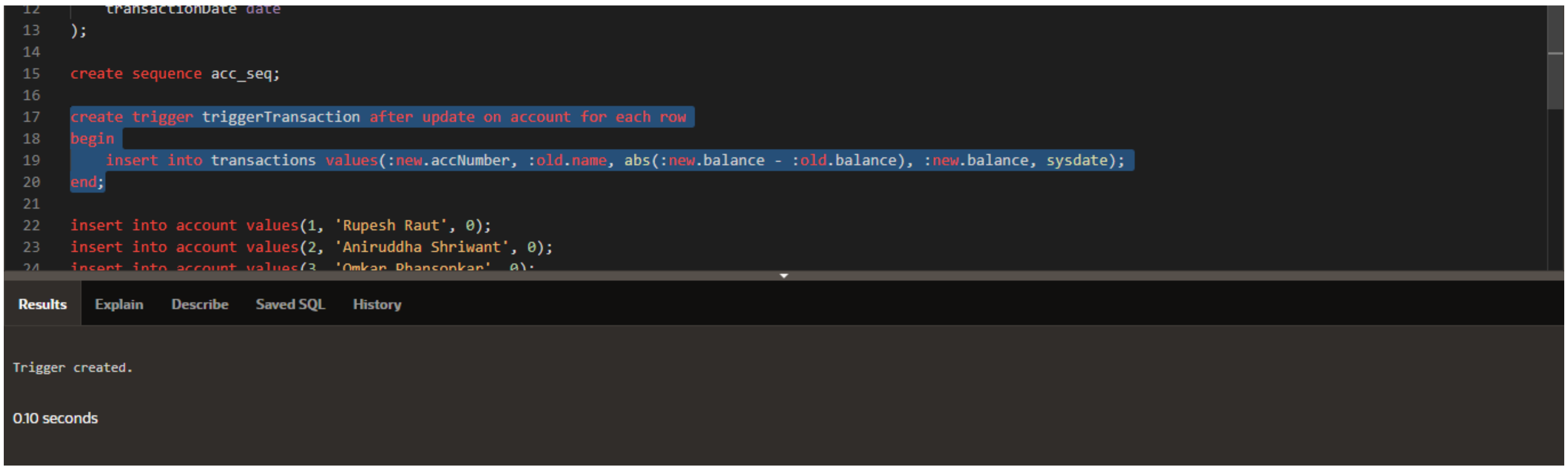
Omkar Phansopkar
FS19CO042



```
    name varchar(20),
    amount number,
    balance number,
    transactionDate date
);

create sequence acc_seq;

create trigger triggerTransaction after update on account for each row
begin
    insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);
```

```
Sequence created.
```

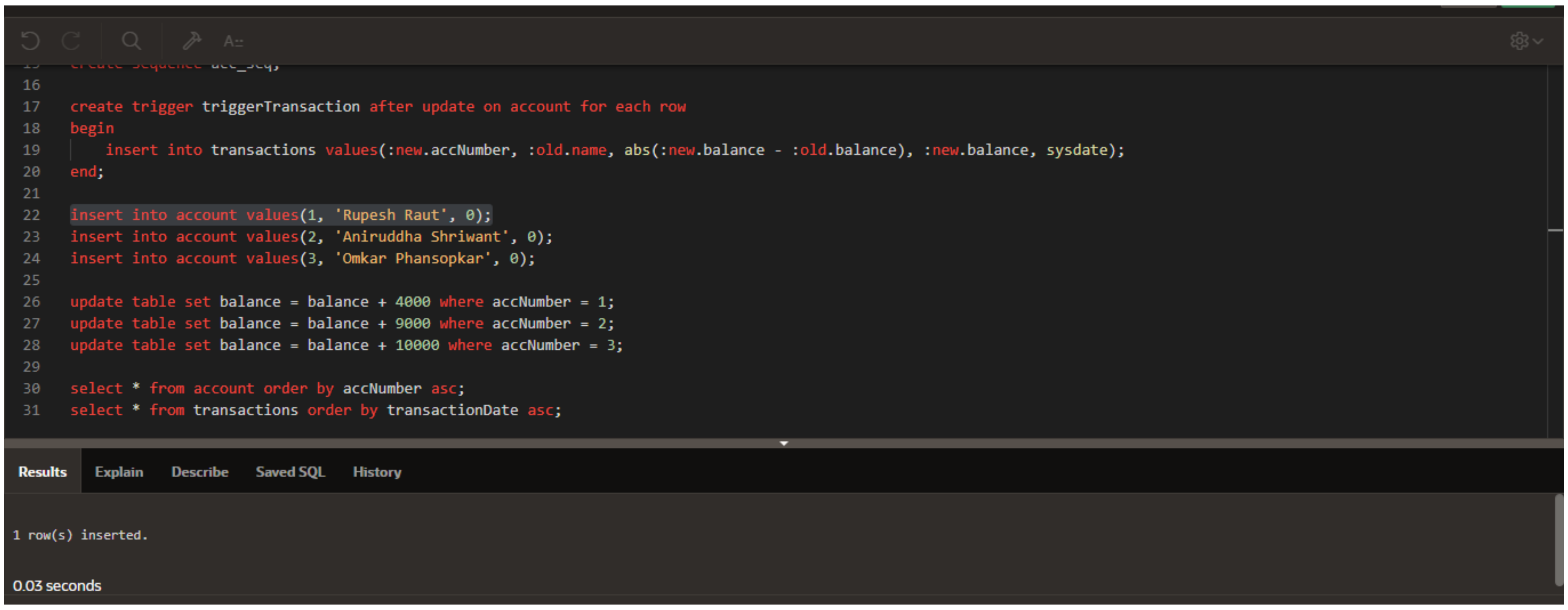
0.04 seconds



```
    transactionDate date
);

create sequence acc_seq;

create trigger triggerTransaction after update on account for each row
begin
    insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);
```

```
Trigger created.
```

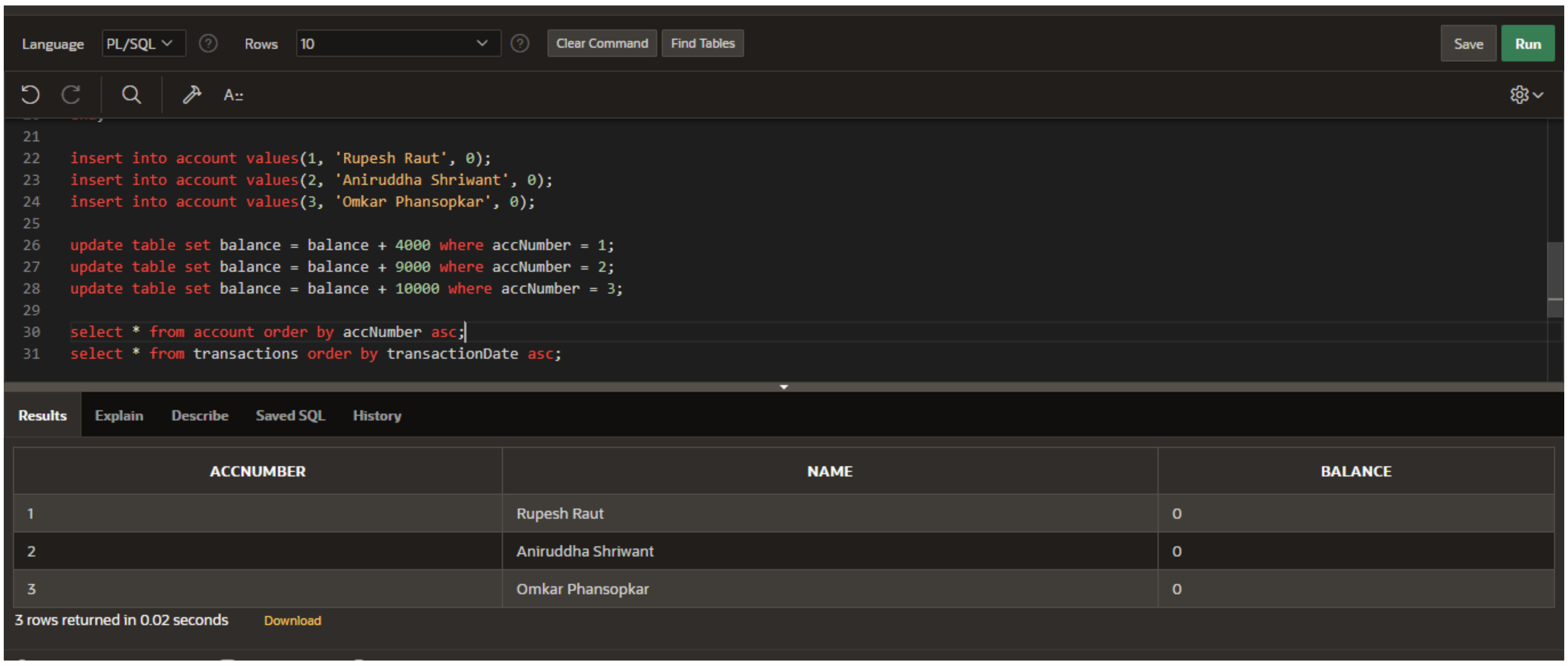
0.10 seconds



```
create sequence acc_seq;

create trigger triggerTransaction after update on account for each row
begin
    insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update table set balance = balance + 4000 where accNumber = 1;
update table set balance = balance + 9000 where accNumber = 2;
update table set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

```
1 row(s) inserted.
```

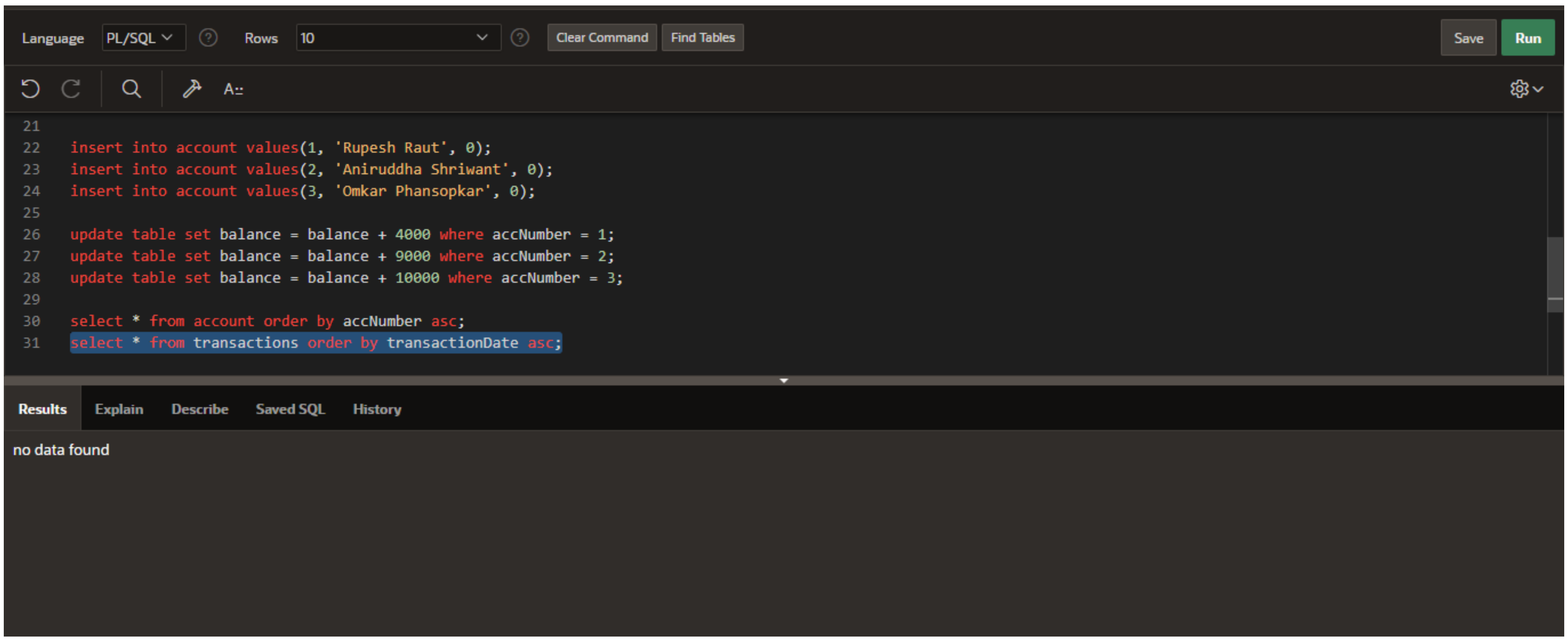
0.03 seconds



```sql

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update table set balance = balance + 4000 where accNumber = 1;
update table set balance = balance + 9000 where accNumber = 2;
update table set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

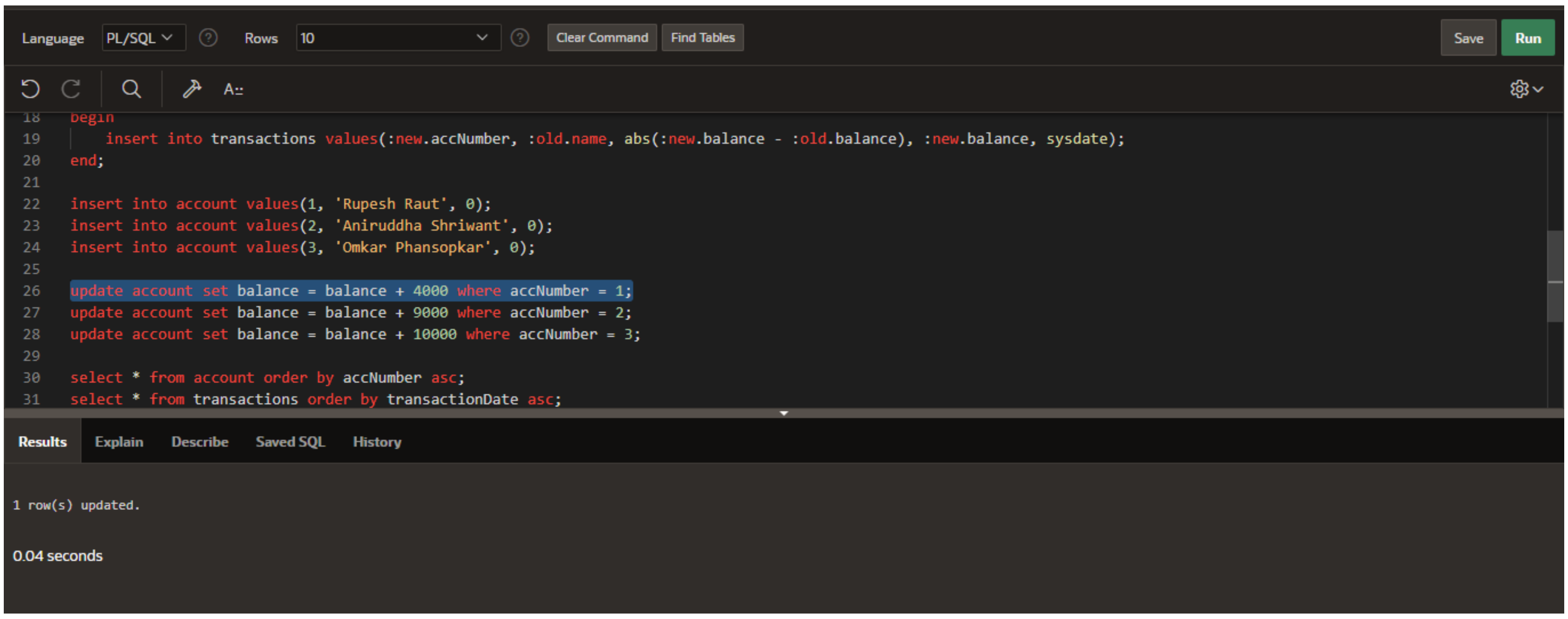
ACCNUMBER	NAME	BALANCE
1	Rupesh Raut	0
2	Aniruddha Shriwant	0
3	Omkar Phansopkar	0

3 rows returned in 0.02 seconds Download



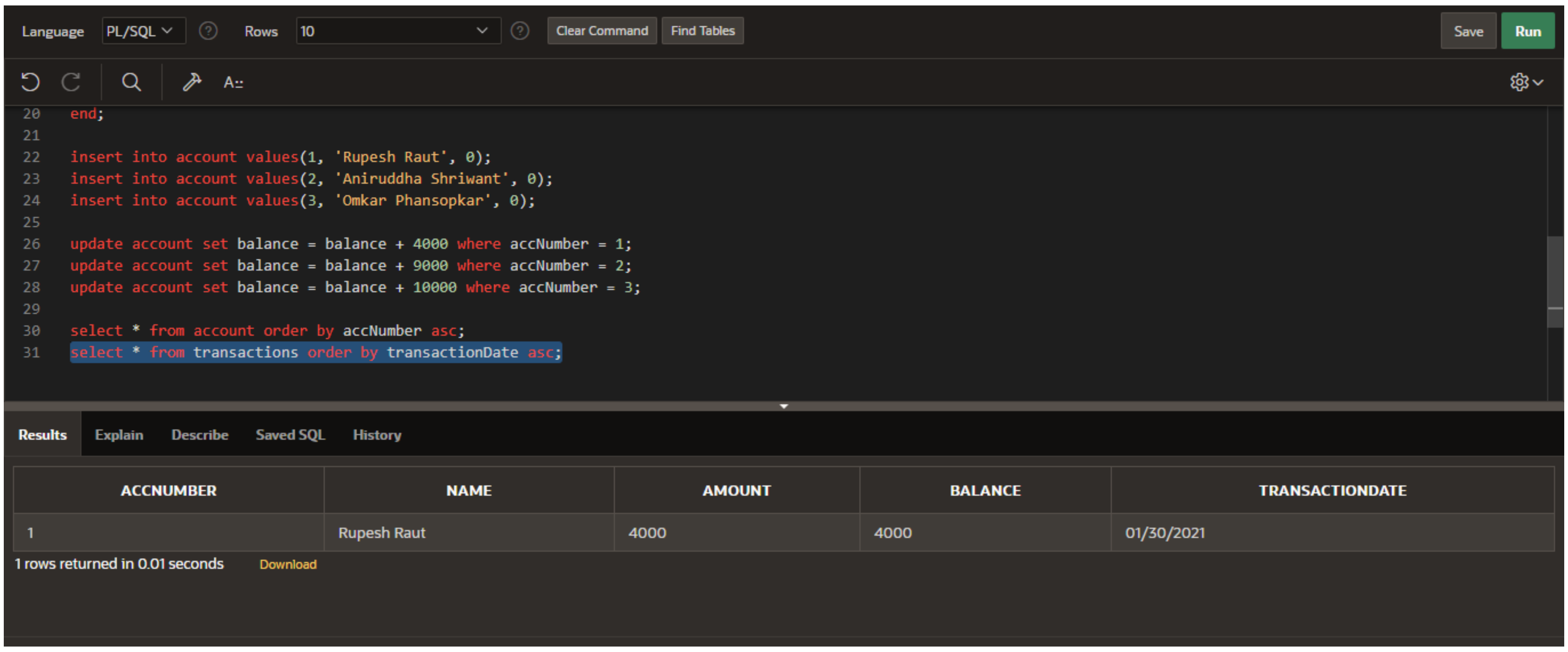
```sql

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update table set balance = balance + 4000 where accNumber = 1;
update table set balance = balance + 9000 where accNumber = 2;
update table set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

```
no data found
```



```sql
begin
    insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update account set balance = balance + 4000 where accNumber = 1;
update account set balance = balance + 9000 where accNumber = 2;
update account set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

```
1 row(s) updated.

0.04 seconds
```

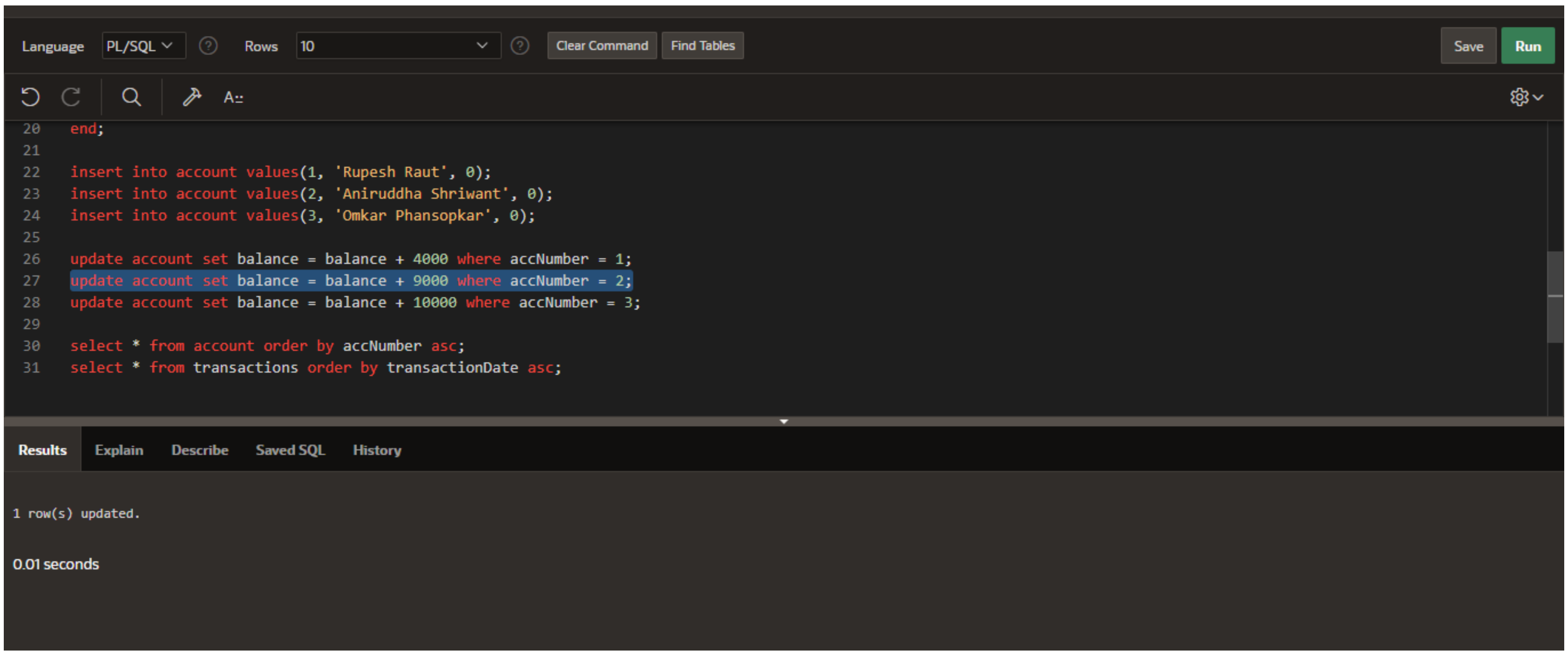


Language PL/SQL Rows 10

```
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update account set balance = balance + 4000 where accNumber = 1;
update account set balance = balance + 9000 where accNumber = 2;
update account set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

ACCNUMBER	NAME	AMOUNT	BALANCE	TRANSACTIONDATE
1	Rupesh Raut	4000	4000	01/30/2021

1 rows returned in 0.01 seconds

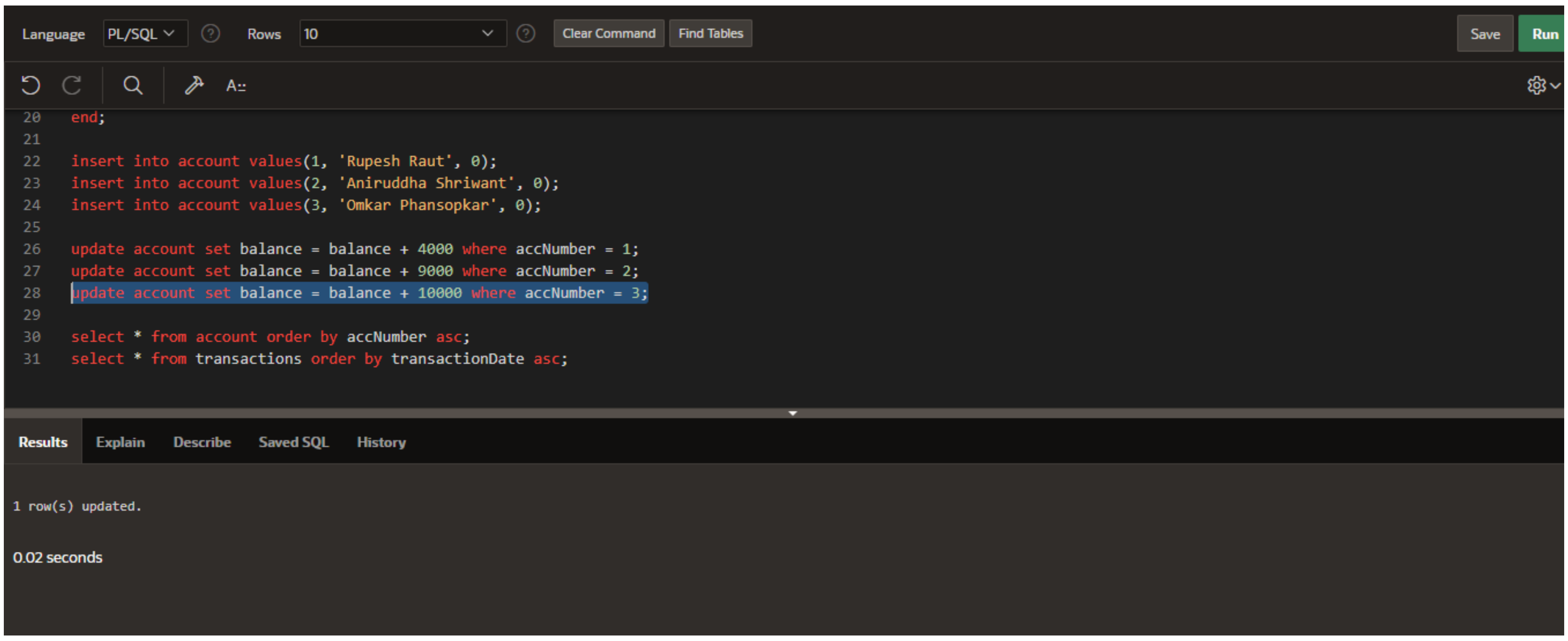


Language PL/SQL Rows 10

```
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update account set balance = balance + 4000 where accNumber = 1;
update account set balance = balance + 9000 where accNumber = 2;
update account set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

```
1 row(s) updated.
```

0.01 seconds

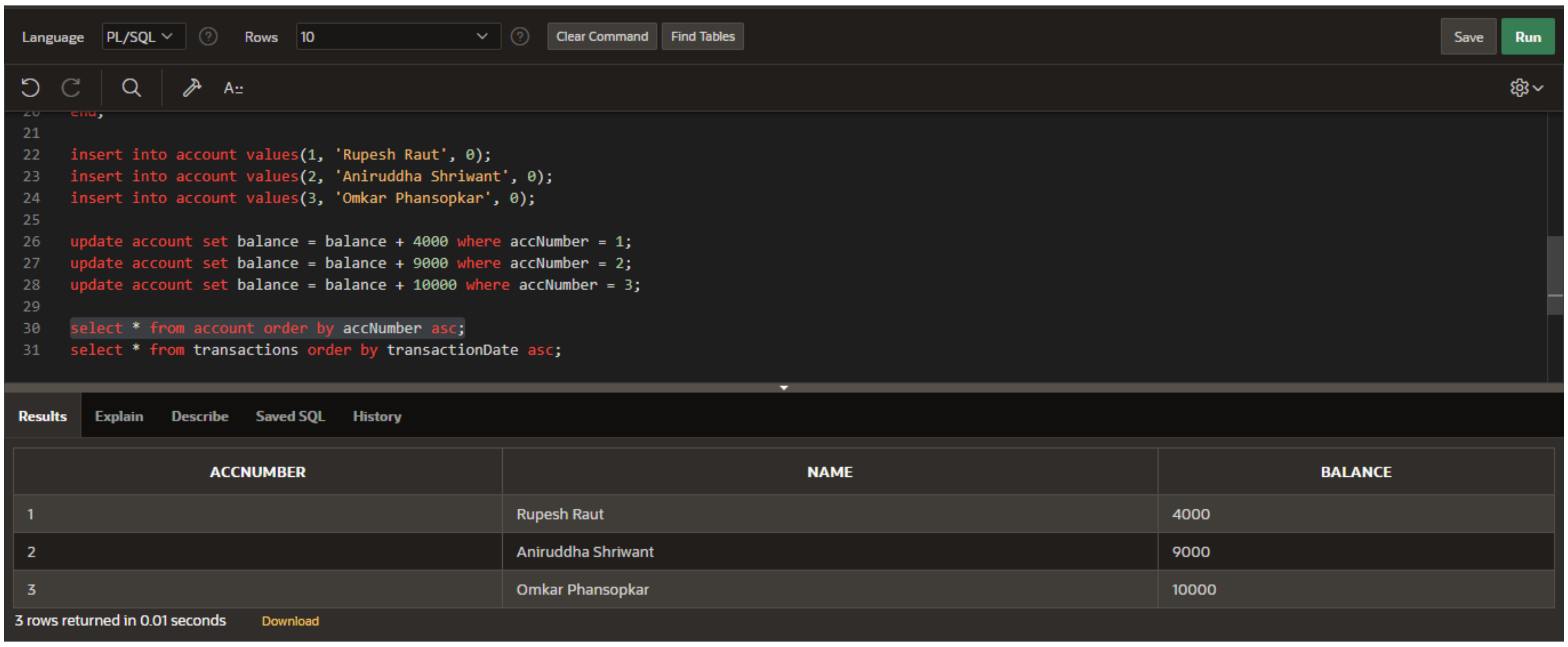


Language PL/SQL Rows 10

```
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update account set balance = balance + 4000 where accNumber = 1;
update account set balance = balance + 9000 where accNumber = 2;
update account set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

```
1 row(s) updated.
```

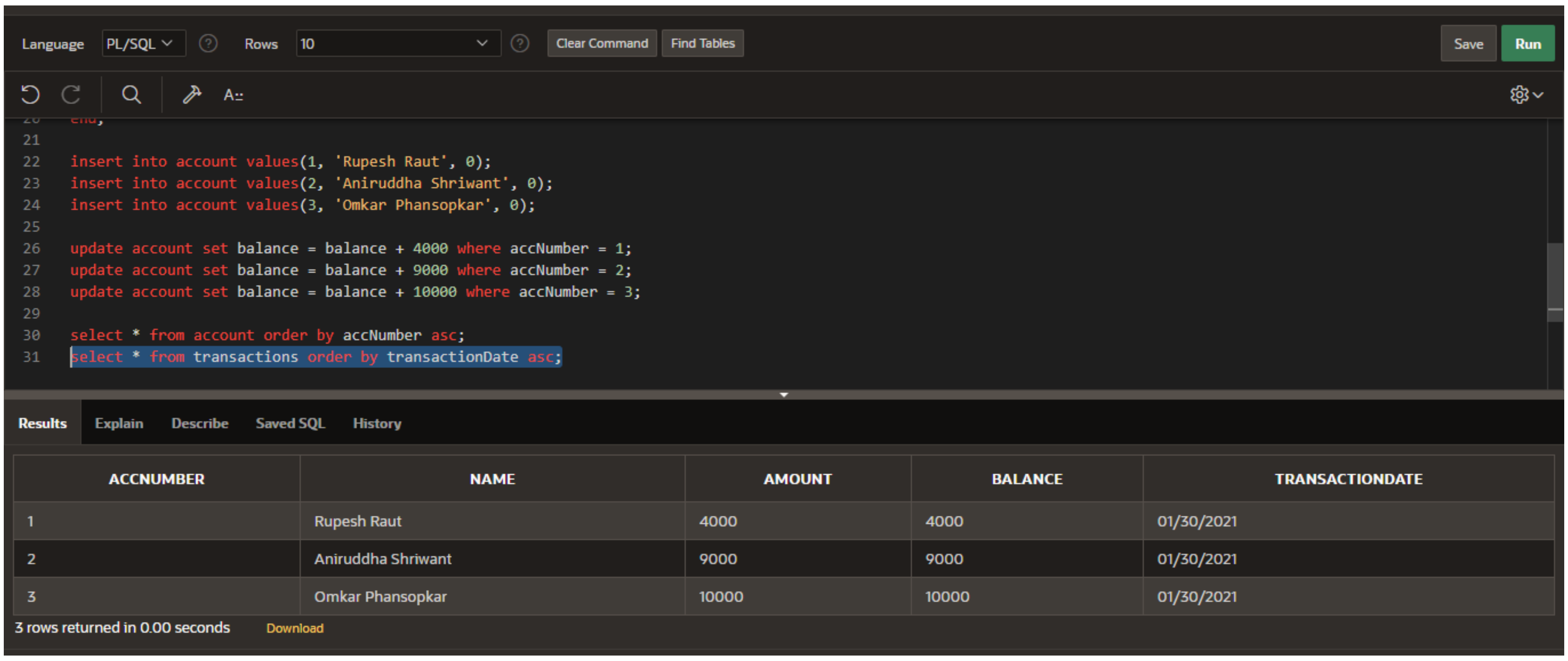
0.02 seconds



```
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update account set balance = balance + 4000 where accNumber = 1;
update account set balance = balance + 9000 where accNumber = 2;
update account set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

ACCNUMBER	NAME	BALANCE
1	Rupesh Raut	4000
2	Aniruddha Shriwant	9000
3	Omkar Phansopkar	10000

3 rows returned in 0.01 seconds



```
end;

insert into account values(1, 'Rupesh Raut', 0);
insert into account values(2, 'Aniruddha Shriwant', 0);
insert into account values(3, 'Omkar Phansopkar', 0);

update account set balance = balance + 4000 where accNumber = 1;
update account set balance = balance + 9000 where accNumber = 2;
update account set balance = balance + 10000 where accNumber = 3;

select * from account order by accNumber asc;
select * from transactions order by transactionDate asc;
```

ACCNUMBER	NAME	AMOUNT	BALANCE	TRANSACTIONDATE
1	Rupesh Raut	4000	4000	01/30/2021
2	Aniruddha Shriwant	9000	9000	01/30/2021
3	Omkar Phansopkar	10000	10000	01/30/2021

3 rows returned in 0.00 seconds

Conclusion: Thus, we created triggers on database.