# /\* On my honor, as an Aggie, I have neither given nor received unauthorized assistance on this assignment. I further affirm that I have not and will not provide this code to any person, platform, or repository, without the express written permission of Dr. Gomillion. I understand that any violation of these standards will have serious repercussions. \*/

# **CREATE AND USE TRANSACTIONS**

START TRANSACTION;

Insert into OrderLine (OrderID, ProductID) values ('1234','1234');

ROLLBACK;

START TRANSACTION;

Insert into OrderLine (OrderID, ProductID) values ('1234','1234');

COMMIT;

(b)

START TRANSACTION;

DELETE from OrderLine where OrderID='1234' AND ProductID='1234';

ROLLBACK;

START TRANSACTION;

DELETE from OrderLine where OrderID='1234' AND ProductID='1234';

COMMIT;

## (c)

START TRANSACTION;

DELETE from OrderLine;

ROLLBACK;

START TRANSACTION;

DELETE from OrderLine;

COMMIT;

START TRANSACTION;

TRUNCATE OrderLine;

ROLLBACK;

START TRANSACTION;

TRUNCATE OrderLine;

COMMIT;

(d)

### **Q1.**

Ans: Transactions are a block of code which can help in updating multiple tables. This block of code gets executed as a whole which means the command succeeds or fails as a whole.

**Q2.**

Ans: One can start a Transaction using “START TRANSACTION” command. We can also use “BEGIN TRANSACTION” command.

**Q3.**

Ans. The two ways we can end a transaction are- “COMMIT” which means that the entire transaction gets stored to the disc and “ROLLBACK” which is like an undo option.

**Q4.**

Ans. “COMMIT” is when we store the data to the disc permanently. If we are sure about our data (No changes would be made in the future) and want to store it, We use the “COMMIT” Command. On the Other Hand, if are not sure about the data, i.e if the data would change in the future, we use the “ROLLBACK” Command.

**Q5.**

Ans. No, The ROLLBACK on the TRUNCATE did not work as expected. This is because Since Truncate is a DDL Commad, it is an implicit commit. Implicit Commands cant be rolled back.

# **TRANSACTION ISOLATION**

START TRANSACTION;

DELETE from OrderLine;

SELECT \* FROM OrderLine;

ROLLBACK;

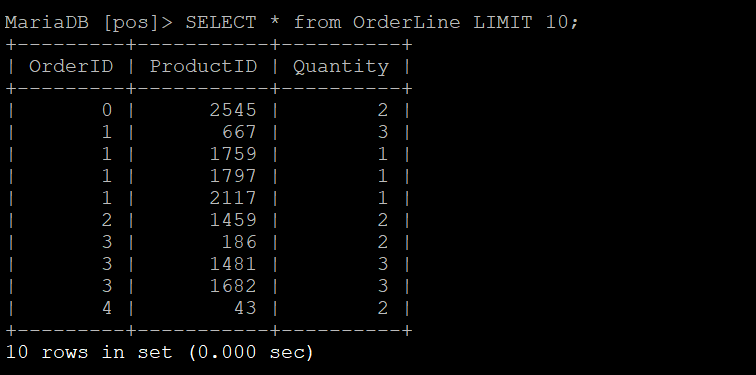
SELECT \* FROM OrderLine;

**(Session A)**

A screen shot of a computer

Description automatically generated

**(Session B)**



(c)

### **Q1.**

Ans. Yes, We can see changes inside of the same session.

**Q2.**

Ans. No, the change is not guaranteed. We can use the “ROLLBACK” command.

**Q3.**

Ans. No, we cannot see the changes outside of the session, before it is committed. It means that that the transactions are limited to the same session.

**Q4.**

Ans. Changes that cannot be seen outside of the same session unless committed results in data consistency. This means that multiple users can use the production system at the same time and similar data would be seen for all the users unless a user commits on the data. Then, the new data can be seen by all the users.

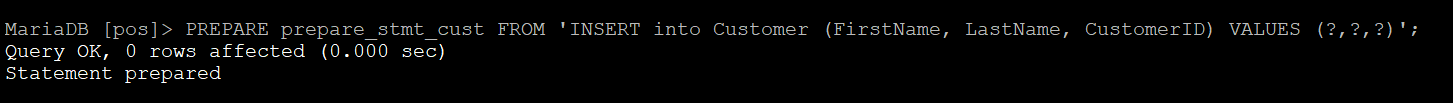
1. **Create and use Prepared statements.**

PREPARE prepare\_stmt\_cust FROM 'INSERT into Customer (FirstName, LastName, CustomerID) VALUES (?,?,?)';

EXECUTE prepare\_stmt\_cust using ‘Bob’,’Dole’,9999;

DROP PREPARE prepare\_stmt\_cust;

**Session A:**



**Session B:**

A black background with white text

Description automatically generated

PREPARE mult\_cust FROM 'INSERT into Customer (FirstName, LastName, CustomerID) VALUES (?,?,?)';

EXECUTE mult\_cust using 'Khyati','Heda',102456;

EXECUTE mult\_cust using 'Bob','Vance',102457;

EXECUTE mult\_cust using 'Micheal','Scott',102458;

EXECUTE mult\_cust using 'Pam','Beesly',102459;

EXECUTE mult\_cust using 'Jim','Halpert',102460;

EXECUTE mult\_cust using 'Stanley','Nun',102461;

EXECUTE mult\_cust using 'Kh','He',102462;

EXECUTE mult\_cust using 'Lee','Chong',102463;

EXECUTE mult\_cust using 'Ross','Dee',102464;

EXECUTE mult\_cust using 'Evan','Lii',102465;

EXECUTE mult\_cust using 'Peter','Evans',102466;

EXECUTE mult\_cust using 'Sean','Peter',102467;

EXECUTE mult\_cust using 'Yuv','Das',102468;

EXECUTE mult\_cust using 'Raahil','Armaan',102469;

EXECUTE mult\_cust using 'Adi','Kochu',102470;

EXECUTE mult\_cust using 'Lily','Scott',102471;

EXECUTE mult\_cust using 'Luke','Dunphy',102472;

EXECUTE mult\_cust using 'Phil','Dunphy',102473;

EXECUTE mult\_cust using 'Alex','Russo',102474;

EXECUTE mult\_cust using 'jahs','Reel',102475;

EXECUTE mult\_cust using 'Anjali','Cutie',102476;

EXECUTE mult\_cust using 'Ro','Ro',102477;

EXECUTE mult\_cust using 'Sam','Club',102478;

EXECUTE mult\_cust using 'North','West',102479;

EXECUTE mult\_cust using 'Haley','Dunphy',102480;

EXECUTE mult\_cust using 'Kriti','Heda',102481;

Using the following command:

**Insert into Customer(FirstName, LastName,CustomerID) VALUES ('Khyati123456','Heda1982763990','15678');**

The number of characters in the above command is: 102

For 10,000 such commands; the Number of Characters would be: 1020000

Now For the Prepared Statement:

**EXECUTE mult\_cust using 'Khyati123456','Heda1982763990',15678;**

The number of characters in the above command is: 62

For 10,000 such commands: Number of characters would be: 620000

Hence to not use a PrepareStatement, we would have to execute 1020000-620000=400000 characters more. That’s why PrepareStatement helps is fast query execution.

**Q1.**

Ans: Using a Prepared Statement does not create much significant impact on a single customer. It is probably just 0.01 seconds faster than the Insert Command.

**Q2**.

Ans: Prepared Statements created a lot of impact for creation of many customers. Not only was it easier and faster to write; It executed much quicker than an Insert Statement and took less space (As seen in Q3-C, where the number of characters used are very less).

**Q3.**

Ans: It is likely to have a positive impact on the performance for long-term since Prepare Statements already “Prepare” the system for the queries that would be executed, this means that the statement is already pre-compiled, and we can keep using the same query by just changing the parameter values. This reduces the need for resources and storage to recompile the statement and improves the performance overall.

1. **SQL Injection**

(a)

SET @FN='Khyati'; SET @LN='Heda'; SET @ID=205678;

INSERT into Customer (FirstName, LastName, CustomerID) VALUES (@FN,@LN,@ID);

(b)

SET @ID='69087); Delete from OrderLine;' ;

INSERT into Customer (CustomerID) VALUES (@ID);

# In this command, Since we have kept the datatype for ID as int, with 11 characters, it shows that the Customer ID’s data is truncated.

**(c)**

Based on the above example, The SQL Query can be easily manipulated by adding extra code (Like for eg: Deleting an entire table), which can change the entire structure of database, and this can harm the entire system. In (b), we passed on the “Delete from OrderLine” command directly with the data. So, technically, The @ID will first take the value (69087) and then it would run the next command, altering the OrderLine table.

1. **Preventing SQL Injection with prepared statements**

SET @ID='69090); Delete from OrderLine;' ;

SET @SAMPLE = CONCAT('INSERT INTO Customer (CustomerID) VALUES ("', @ID);

PREPARE mult\_cust FROM @SAMPLE;

EXECUTE mult\_cust;

Prepared statements help protect against the SQL Injection attack. In SQL Injection Attack the attacker hides a malicious code inside the data type (Like we did before). In 4(b), we saw how without Prepared Statements, the attacker could easily insert malicious code inside data. With Prepared Statements, The Query is first executed; and then the data is added to the query. So, whenever an attacker tries to insert another query, The prepared statement would discard it; thus saving the program to be altered.