**/\* 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. \*/**

# 1) Create and use transactions

## a. Adding a new OrderLine

**/\* Successful Transaction \*/**

START TRANSACTION;

SET autocommit = 0;

INSERT INTO OrderLine (OrderID, ProductID, Quantity) VALUES (1234, 1234, 1);

COMMIT;

**/\* Unsuccessful Transaction \*/**

START TRANSACTION;

SET autocommit = 0;

INSERT INTO OrderLine (OrderID, ProductID, Quantity) VALUES (1234, 1234, 1);

ROLLBACK;

## b. Deleting an OrderLine

**/\* Successful Transaction \*/**

START TRANSACTION;

SET autocommit = 0;

DELETE FROM OrderLine WHERE OrderID = 1234 AND ProductID = 1234;

COMMIT;

**/\* Unsuccessful Transaction \*/**

START TRANSACTION;

SET autocommit = 0;

DELETE FROM OrderLine WHERE OrderID = 1234 AND ProductID = 1234;

ROLLBACK;

## c. Deleting all OrderLines

**/\* Successful Transaction \*/**

START TRANSACTION;

SET autocommit = 0;

DELETE FROM OrderLine;

COMMIT;

**/\* Restoring all the records \*/**

Source views.sql;

**/\* Unsuccessful Transaction \*/**

START TRANSACTION;

SET autocommit = 0;

DELETE FROM OrderLine;

ROLLBACK;

## d. Reflecting on transactions

**/\* Deleting all OrderLine records - using TRUNCATE - COMMIT\*/**

START TRANSACTION;

SET autocommit = 0;

TRUNCATE TABLE OrderLine;

COMMIT;

**/\* Restoring all the records \*/**

Source views.sql;

**/\* Deleting all OrderLine records - using TRUNCATE - ROLLBACK \*/**

START TRANSACTION;

SET autocommit = 0;

TRUNCATE TABLE OrderLine;

ROLLBACK;

**/\* Restoring all the records \*/**

Source views.sql;

**1) What is the purpose of a transaction?**

The transaction is used to maintain ACID Compliance while performing certain operations in the database management system.

A - Atomic - The whole thing happens at once. Intermediate steps or changes will not be available to users and sessions that are using the database at the same time.

C - Consistent - The changes in the data are going to reflect across all required tables. Changes are going to be consistent in the database in different tables.

I - Isolated - Isolation is that the session does not impact the results that others see until the transaction is complete.

D - Durable - Changes that occurred in the database due to transactions are going to stick until another change overrides it.

**2) How do you start a transaction?**

We can use the 'START TRANSACTION' statement to start a transaction in the database management system.

**3) What are the two ways you end a transaction?**

There are 2 ways to end a transaction in a database management system and those are

1. COMMIT - This is used to commit the changes to the database and make the transaction successful.

2. ROLLBACK - This is used to rollback/undo the changes and cancel the transaction in the database.

**4) How do you know which way you want to end a particular transaction?**

If we want to commit/make changes in the database successfully then using the 'COMMIT' at the end of a transaction is the best way.

If we want to rollback/undo the changes during the transaction then we can use the 'ROLLBACK' Statement. It is used to roll back the changes when any kind of error occurs in the transaction while performing some operations in the database.

**5) Did the ROLLBACK on the TRUNCATE work as expected? Why or why not?**

ROLLBACK on the TRUNCATE did not work as expected. TRUNCATE is one of the SQL DDL statements that can cause implicit commit. Due to that changes committed to the database at that statement itself and, there is nothing to rollback after that implicit commit so ROLLBACK didn't work as expected.

# 2) Transaction Isolation

## a) Visibility of changes inside of transactions from the same session

START TRANSACTION;

SET autocommit = 0;

DELETE FROM OrderLine;

SELECT \* FROM OrderLine;

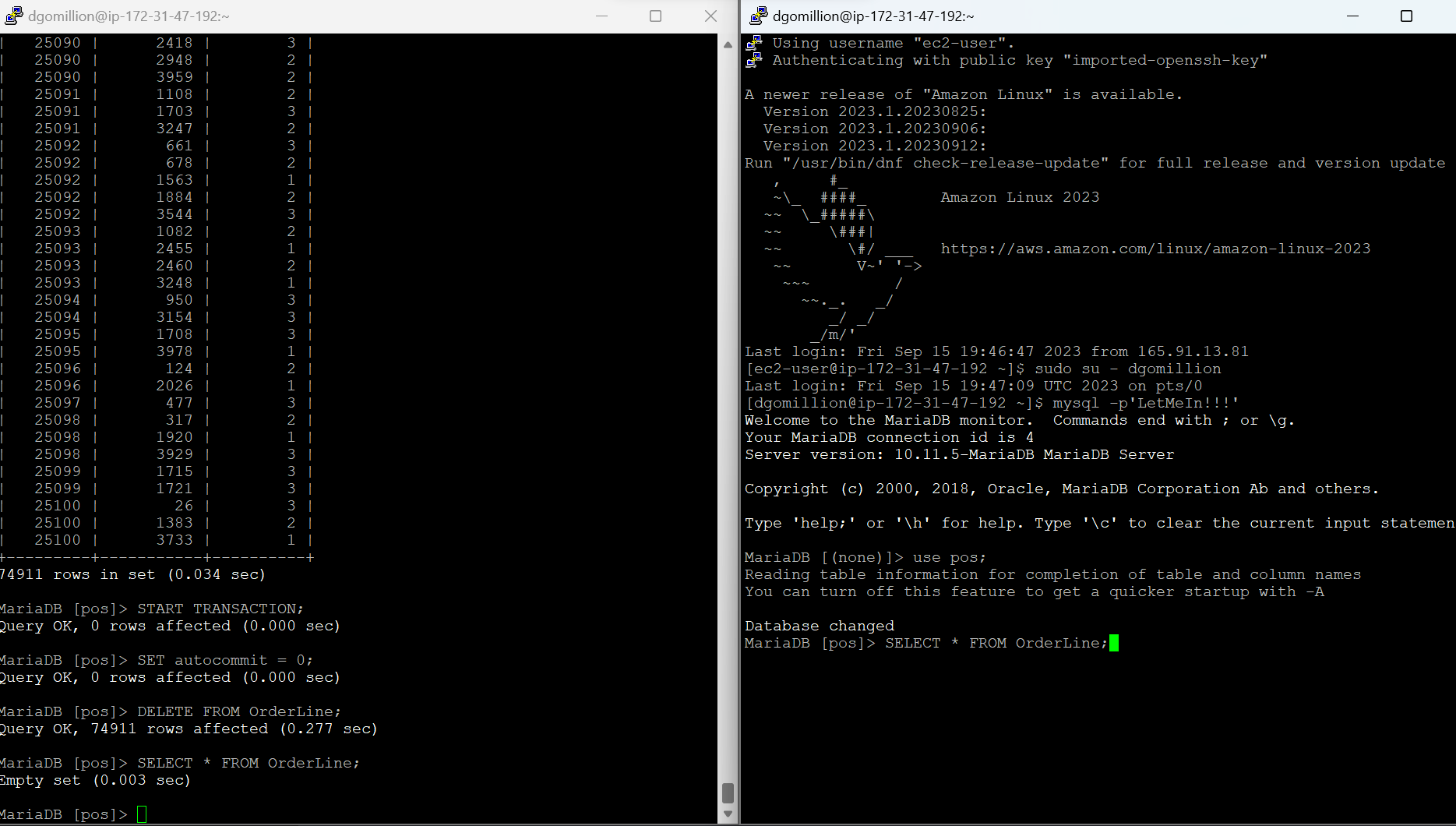
ROLLBACK;

SELECT \* FROM OrderLine;

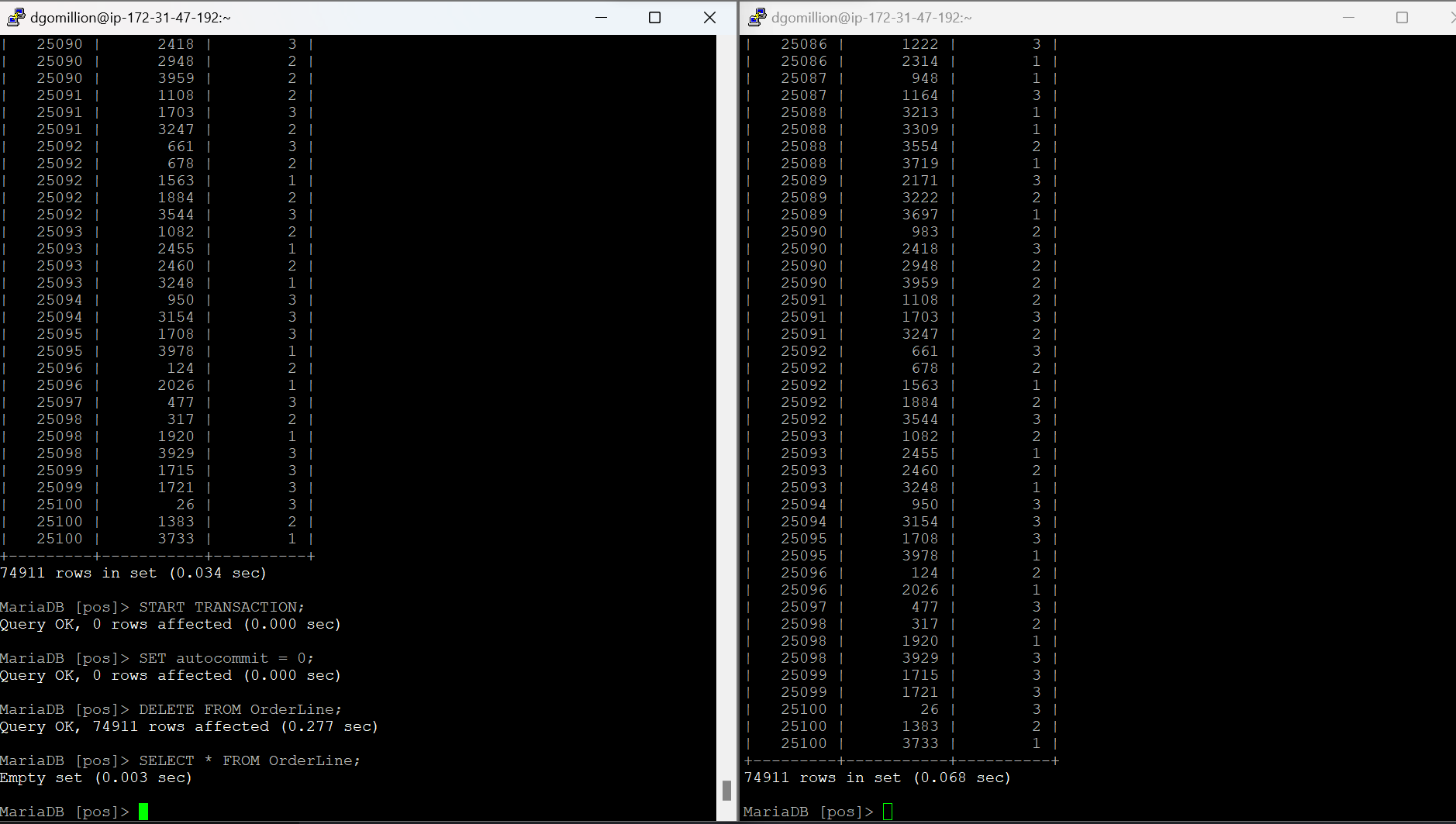
## b) Visibility of changes inside of transactions from other sessions

Relevant screenshots from both sessions.

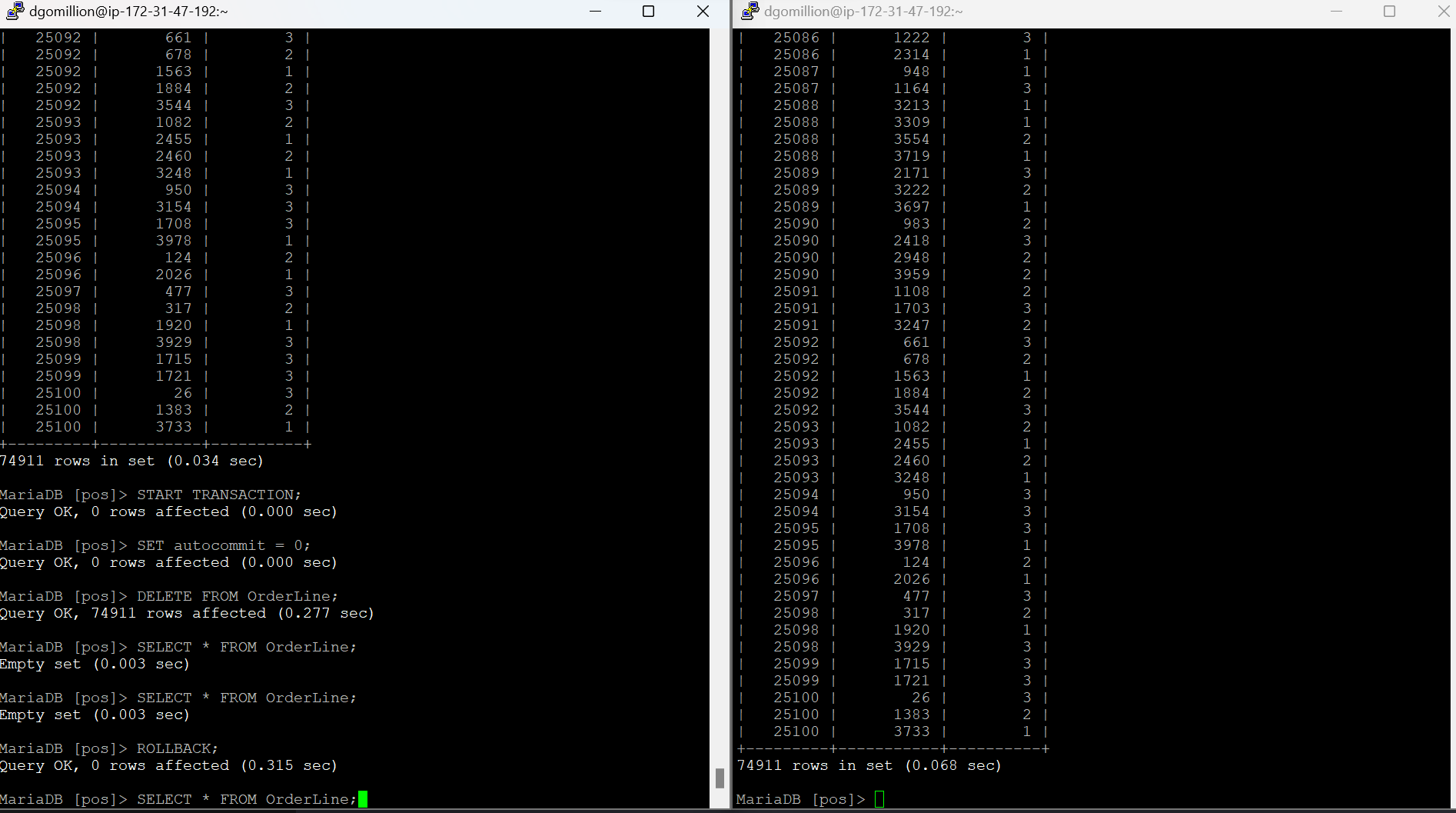
Before Executing SELECT query in session B



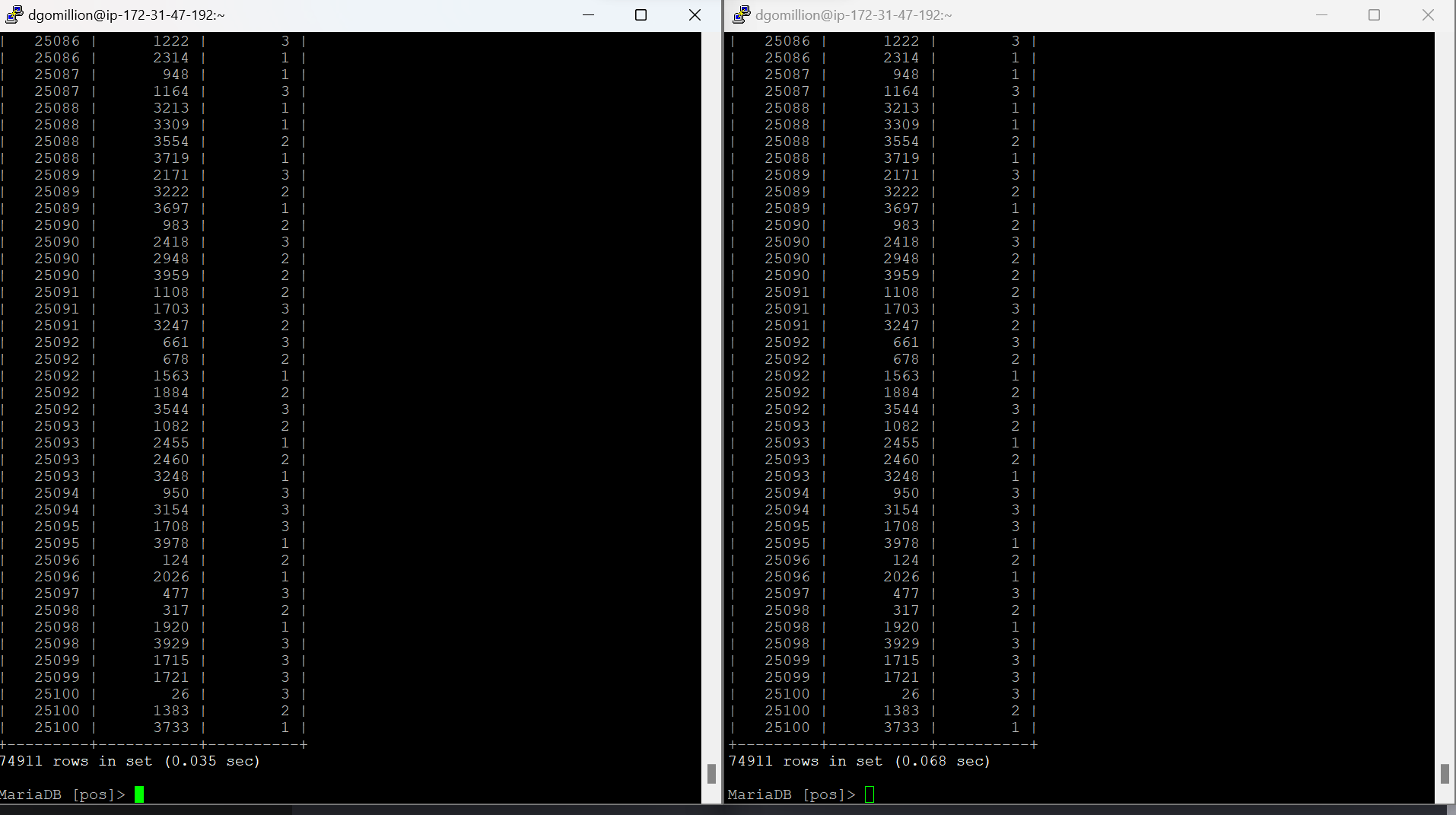
After Executing the SELECT in Session B



Before executing SELECT query in session A



After Executing the SELECT query in Session A.



## c. Reflecting on transaction isolation

**1) Can you see changes inside of the same session as they are happening but before they are**

**committed?**

Yes, I am able to see changes inside of the same session as they are happening before committing them to the database. For Instance, after starting the transaction executed a statement (DELETE FROM OrderLine;) - it resulted in the deletion of all rows, and after using SELECT \* FROM OrderLine; - It displayed the result as an empty set (0 rows). However, after doing the rollback - the transaction was cancelled and the changes were undone.

**2) Is the change guaranteed to be permanent? Why is this important to remember?**

The changes done during the transaction before actually committing to the database are not guaranteed to be permanent. For Instance, after starting the transaction executed a statement (DELETE FROM OrderLine;) - it resulted in the deletion of all rows and after using SELECT \* FROM OrderLine; - It displayed the result as empty set (0 rows). But later I executed a ROLLBACK statement which cancelled the transaction and restored all the orderline records. So the changes are not guaranteed to be permanent until committing the changes to the database.

**3) Can you see the changes outside of the same session as they are happening but before they are committed? Why is this important to remember?**

No, I am not able to see the changes outside of the session as they are happening but before they are committed. (Outside of session -> In session B – a different session)

It is important to remember that the changes that are being done in one transaction under a session will not disturb the actual database data until the transaction is committed.

**4) How do you think you could use this to your advantage in a production system?**

This is very useful in the production system in such a way that - we can perform all the operations we want in a transaction without actually committing to the database. So that we can observe how operations are going on without actually disturbing / changing the actual production data in the system. Later we can rollback if we don't want to commit the changes to the database.

If we really want to make changes we can use COMMIT instead of ROLLBACK to commit all those operations actions/results in the database.

# 3) Create and use prepared statements

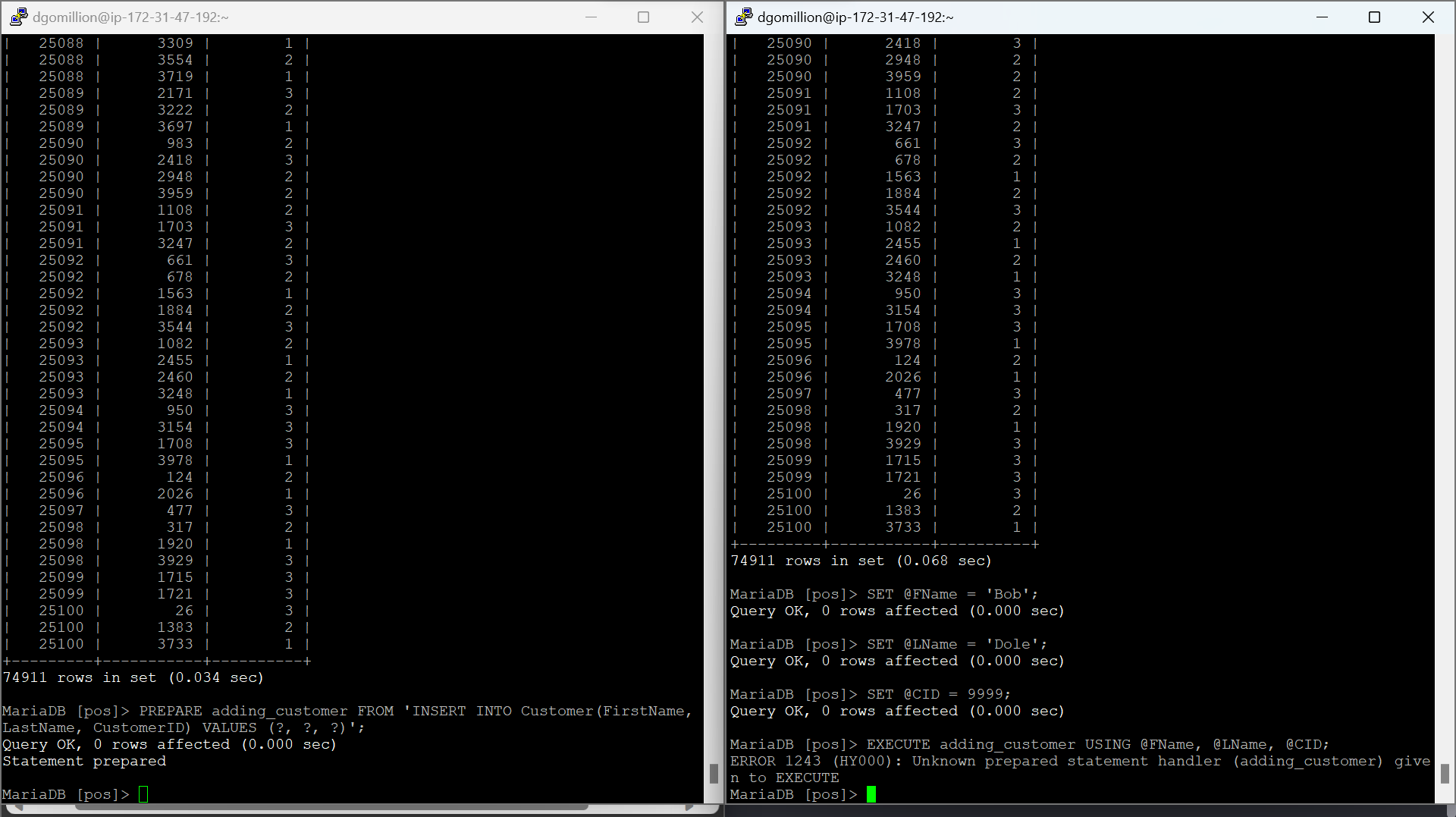
## a) Adding a new customer

PREPARE adding\_customer FROM 'INSERT INTO Customer (FirstName, LastName, CustomerID) VALUES (?, ?, ?)';

EXECUTE adding\_customer USING 'Bob', 'Dole', 9999;

DEALLOCATE PREPARE adding\_customer;

Declared the prepared statement in one session and executed the statement in the second session. Screenshots of the sessions side by side.



Faced an exception while using/executing the prepared statement in the second session.

## b) Adding a lot of new customers

**/\* Adding new 25 Customers to the Customer Table\*/**

PREPARE adding\_customer FROM 'INSERT INTO Customer (FirstName, LastName, CustomerID) VALUES (?, ?, ?)';

EXECUTE adding\_customer USING 'FirstNameRecord0', 'LastNameRecord0', 9999990;

EXECUTE adding\_customer USING 'FirstNameRecord1', 'LastNameRecord1', 9999991;

EXECUTE adding\_customer USING 'FirstNameRecord2', 'LastNameRecord2', 9999992;

EXECUTE adding\_customer USING 'FirstNameRecord3', 'LastNameRecord3', 9999993;

EXECUTE adding\_customer USING 'FirstNameRecord4', 'LastNameRecord4', 9999994;

EXECUTE adding\_customer USING 'FirstNameRecord5', 'LastNameRecord5', 9999995;

EXECUTE adding\_customer USING 'FirstNameRecord6', 'LastNameRecord6', 9999996;

EXECUTE adding\_customer USING 'FirstNameRecord7', 'LastNameRecord7', 9999997;

EXECUTE adding\_customer USING 'FirstNameRecord8', 'LastNameRecord8', 9999998;

EXECUTE adding\_customer USING 'FirstNameRecord9', 'LastNameRecord9', 9999999;

EXECUTE adding\_customer USING 'FirstNameRecord10', 'LastNameRecord10', 99999910;

EXECUTE adding\_customer USING 'FirstNameRecord11', 'LastNameRecord11', 99999911;

EXECUTE adding\_customer USING 'FirstNameRecord12', 'LastNameRecord12', 99999912;

EXECUTE adding\_customer USING 'FirstNameRecord13', 'LastNameRecord13', 99999913;

EXECUTE adding\_customer USING 'FirstNameRecord14', 'LastNameRecord14', 99999914;

EXECUTE adding\_customer USING 'FirstNameRecord15', 'LastNameRecord15', 99999915;

EXECUTE adding\_customer USING 'FirstNameRecord16', 'LastNameRecord16', 99999916;

EXECUTE adding\_customer USING 'FirstNameRecord17', 'LastNameRecord17', 99999917;

EXECUTE adding\_customer USING 'FirstNameRecord18', 'LastNameRecord18', 99999918;

EXECUTE adding\_customer USING 'FirstNameRecord19', 'LastNameRecord19', 99999919;

EXECUTE adding\_customer USING 'FirstNameRecord20', 'LastNameRecord20', 99999920;

EXECUTE adding\_customer USING 'FirstNameRecord21', 'LastNameRecord21', 99999921;

EXECUTE adding\_customer USING 'FirstNameRecord22', 'LastNameRecord22', 99999922;

EXECUTE adding\_customer USING 'FirstNameRecord23', 'LastNameRecord23', 99999923;

EXECUTE adding\_customer USING 'FirstNameRecord24', 'LastNameRecord24', 99999924;

EXECUTE adding\_customer USING 'FirstNameRecord25', 'LastNameRecord25', 99999925;

DEALLOCATE PREPARE adding\_customer;

## c) Data-sending requirements with prepared statements

**/\* Adding 10,000 new customers \*/**

INSERT INTO Customer (CustomerID, FirstName, LastName) VALUES (12345, 'FirstNameRecord', 'LastNameRecordK');

Count of characters to be send = 106 \* 10,000 = 1,060,000

**/\* Adding 10,000 new customers – using prepared statement \*/**

PREPARE adding\_customer FROM 'INSERT INTO Customer (FirstName, LastName, CustomerID) VALUES (?, ?, ?)';

EXECUTE adding\_customer USING 12345, 'FirstNameRecord', 'LastNameRecordK;

Prepared statement Characters = 102

For 10,000 customers -

Characters count = 102 + (57 \* 10,000) = 5,70,102 (Excluding adding\_customer in execute)

Characters count = 102 + (73 \* 10,000) = 7,30,102 (Including adding\_customer in execute)

The number of characters to be sent to the database are less in prepared statement methodology compared to the normal Insertion. This will reduce the compile time and increases efficiency.

## d) Reflecting on prepared statements

**1) How did using a prepared statement impact the creation of a single customer?**

Prepared statement creation really made the creation/insertion of customer records easier compared to the normal insert statement. In my opinion, for a single customer insertion the normal insert statement will be the best approach because here we have to run 2 statements.

**2) How did using a prepared statement impact the creation of many customers?**

In my opinion, using the prepared statement is the best approach when we insert/create multiple customers in the database. Because when we use a prepared statement, then we can just use the Exexute statement and the required columns information to create records whereas in the normal insertion - to insert multiple customers we have to type the (Insert into table\_name(column1, column2..) VALUES(value1, value2..)) statement multiple times Which really consumes a lot of effort and reduces efficiency of the database.

By using prepared statements, the database prepares an execution plan based on the statement and executes commands and inserts records. This works really fast and it's the safer approach as well. Moreover the number of characters to be used or sent is very less in prepared statement compared to the normal insert while inserting numerous records.

**3) How is this likely to impact performance long term?**

In the long term, it's going to improve the overall performance of the database. Prepared statements are faster in execution and maintain the integrity and efficiency of the database.

# 4. SQL Injection

## a. Programmatically-created SQL

**/\* Inserting Customer Record by programmatically – created SQL \*/**

SET @ID = 999999;

SET @FN = 'FirstNameRecordA';

SET @LN = 'LastNameRecordA';

INSERT INTO Customer (CustomerID, FirstName, LastName)

VALUES (@ID, @FN, @LN);

## b. Injecting harmful code inside of a data field

SET @ID = 989898;TRUNCATE TABLE OrderLine;SELECT \* FROM OrderLine;

SET @FN = 'FirstNameRecordB';

SET @LN = 'LastNameRecordB';

INSERT INTO Customer (CustomerID, FirstName, LastName)

VALUES (@ID, @FN, @LN);

It inserted the customer record but it deleted all OrderLine records from the OrderLine table successfully.

## c. Reflecting on SQL injection

If we are accepting the input from the web, then attackers might use some malicious code to gain access/ make some changes to the backend database system.

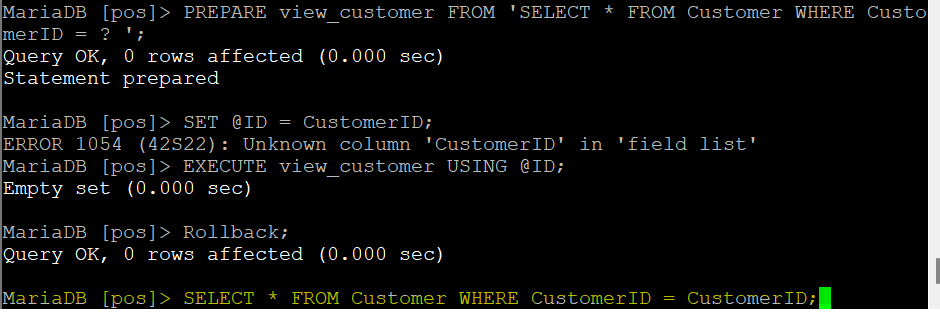
For instance:

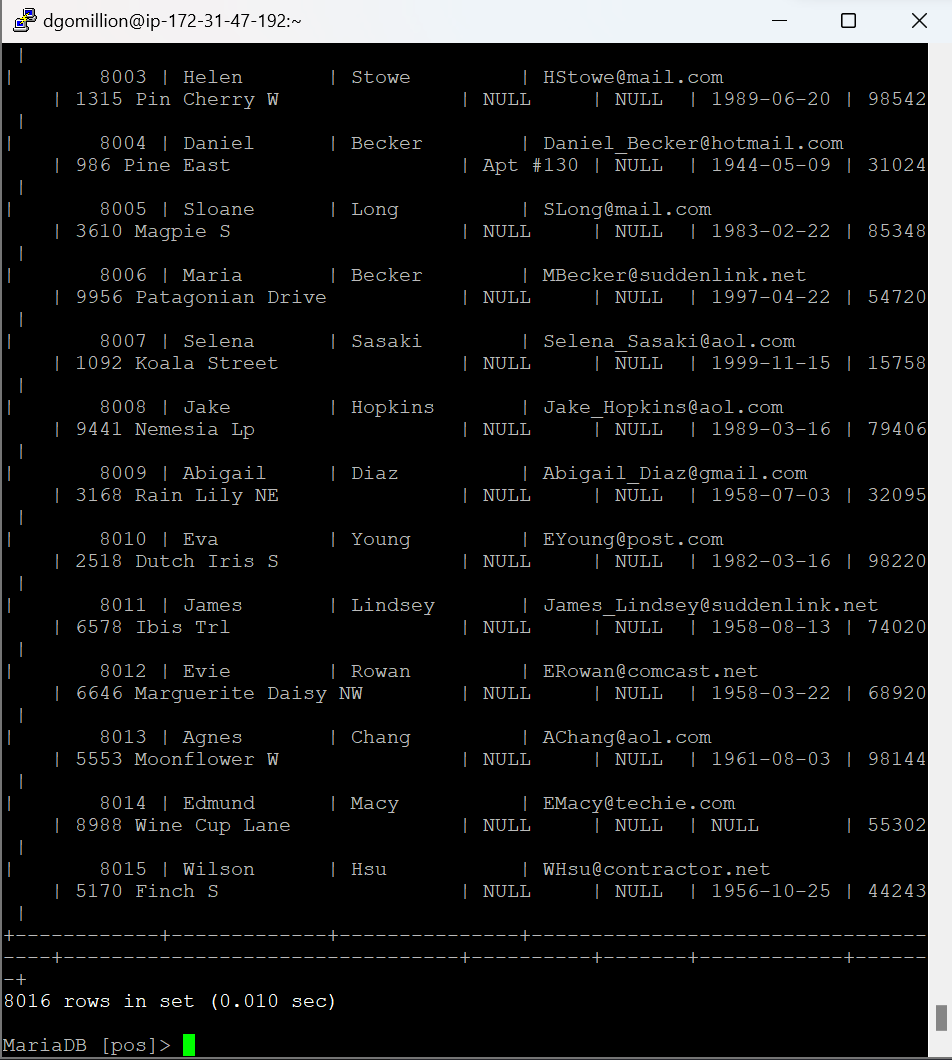
SET @ID = CustomerID;

We have a dynamic query as:

SELECT \* FROM Customers WHERE CustomerID = @ID;

In the above mentioned case, system is going to retrieve all the customer’s information. So the attacker can actually see all customer’s hidden information. This breaks the integrity of the database.





# 5. Preventing SQL Injection with prepared statements

## a. Protecting against SQL Injection using a prepared statement

**/\* Prepared statement from 3a \*/**

PREPARE adding\_customer FROM 'INSERT INTO Customer (FirstName, LastName, CustomerID) VALUES (?, ?, ?)';

SET @ID = 989898;TRUNCATE TABLE OrderLine;SELECT \* FROM OrderLine;

SET @FN = 'FirstNameRecordB';

SET @LN = 'LastNameRecordB';

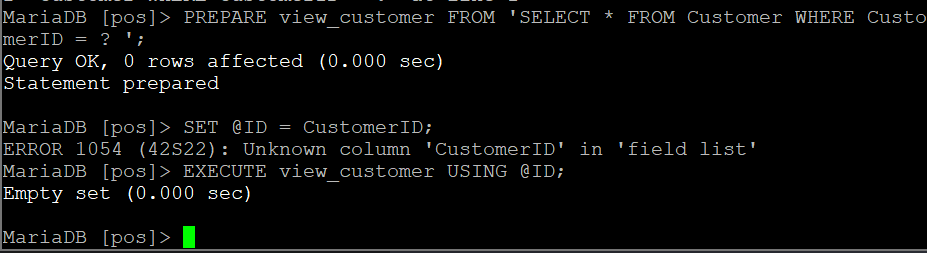
EXECUTE adding\_customer USING @FN, @LN, @ID;

Another example:

PREPARE view\_customer FROM 'SELECT \* FROM Customer WHERE CustomerID = ? ';

SET @ID = CustomerID;

EXECUTE view\_customer USING @ID;



Faced an exception while assigning value to ID as CustomerID.

## b) Reflection on using prepared statements to protect against SQL injection

Prepared statements are a power tool which can be used to prevent critical SQL injection attacks. Prepared statements prepares an execution plan and let the database know what it’s going to execute next. Prepared statement is like a template / method which can be used repeatedly to perform certain defined operation instead of calling the actual statement couple of times. As it accepts parameters to perform these operations, it will be very difficult for attackers to manipulate the prepared statement. Moreover these things are not directly added to the SQL query in that way it reduces the SQL Injection risks. Prepared statements are very easy to use and improves the performance of the database.