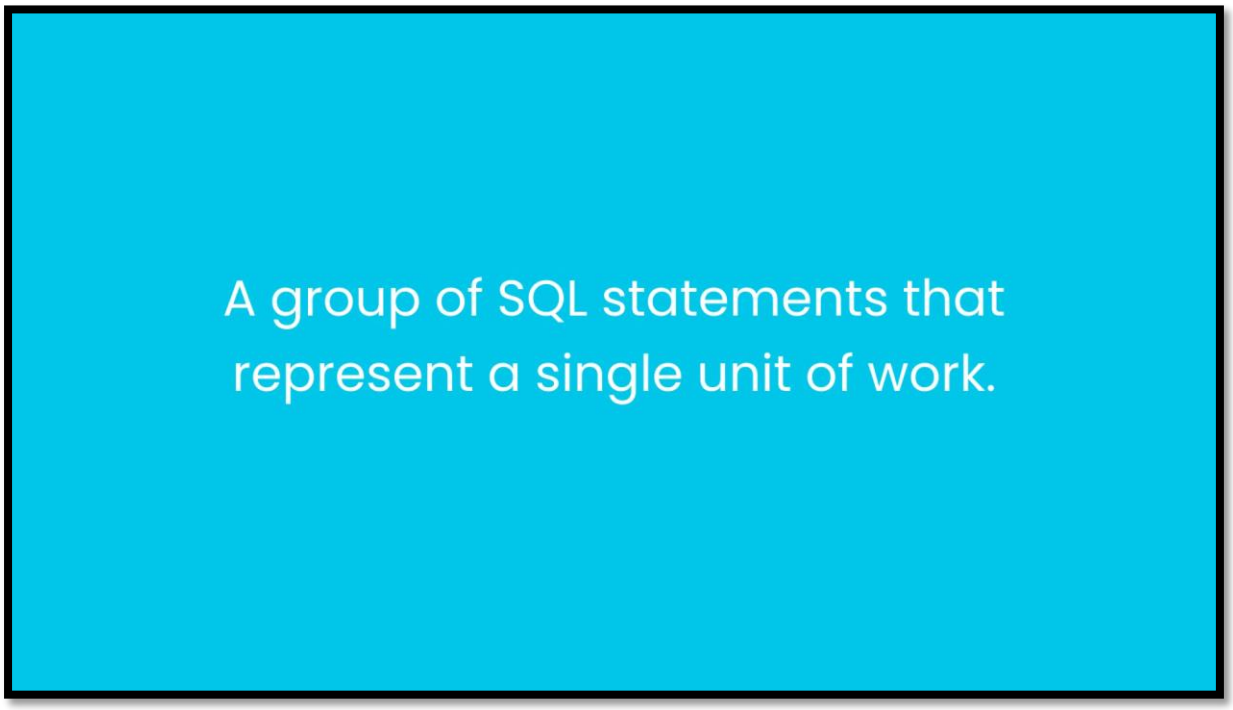
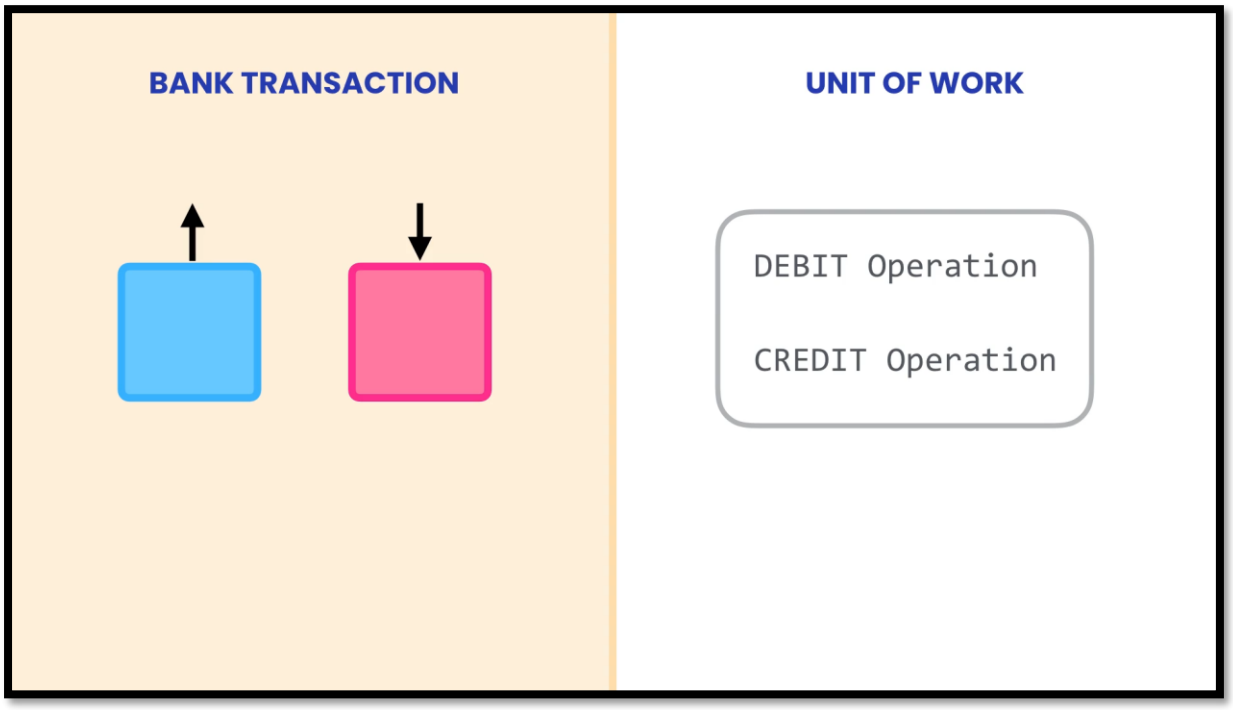
Transactions



All the statements should be completed successfully or the transaction should fail.

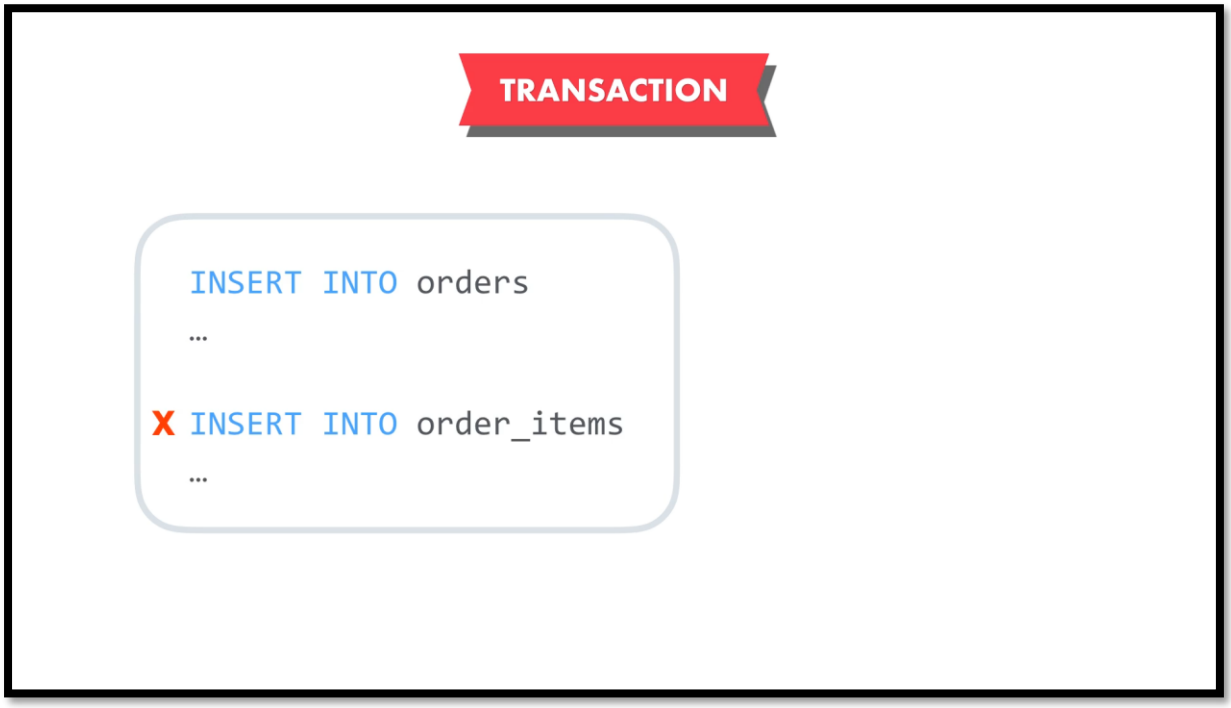


When we transfer RS10,000 from our account to friend's account. RS10,000 should be taken out of your account and deposited into your friend's account.

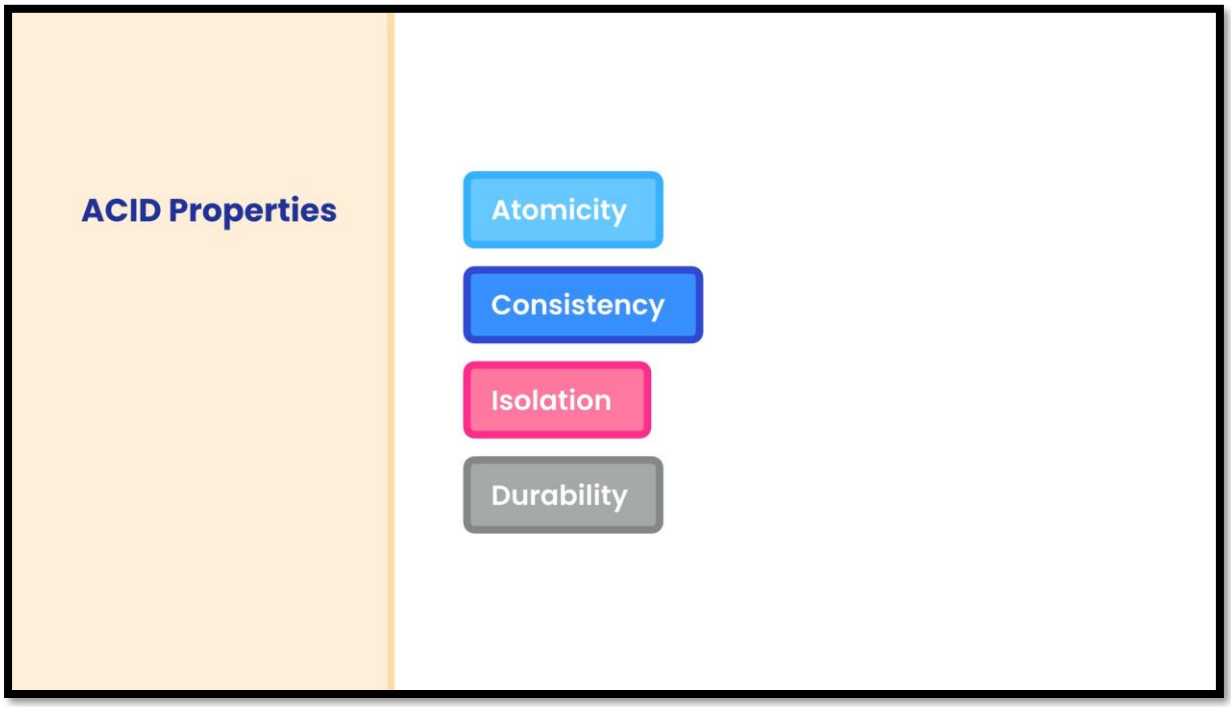
Here, we have two operations. These two operations represent single unit of work. Either both these operations complete successfully or if the

first operation succeeds successfully but the second operation fails, we need to rollback or revert the changes done by the first operation.

We use transaction when we want multiple changes to database and we want all these changes succeed or fail together as a single unit.



Properties of Transaction

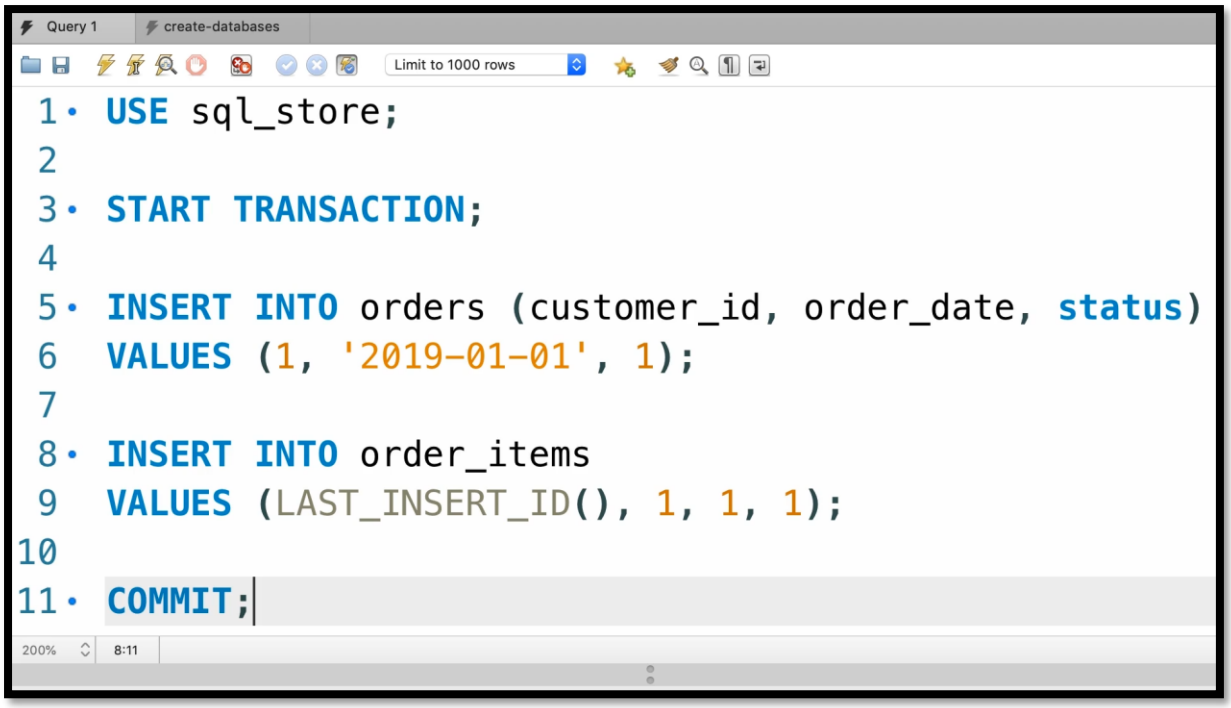


1. Atomicity
   Transaction is like an atom ie unbreakable. Each transaction is single unit of work no matter how many statements it contains. Either all the statements will be executed successfully and transaction is committed or the transaction is rolled back and all the changes are undone.

2. Consistency
   Our database will remain in a consistent state within these transactions. We can not have an order without an item.

3. Isolation
   These transactions are protected from each other if they try to modify the same data. They can not interfere with each other. If multiple transactions try to update the same data, only one transaction at time can update the data. Other transactions have to wait for that transaction to complete.

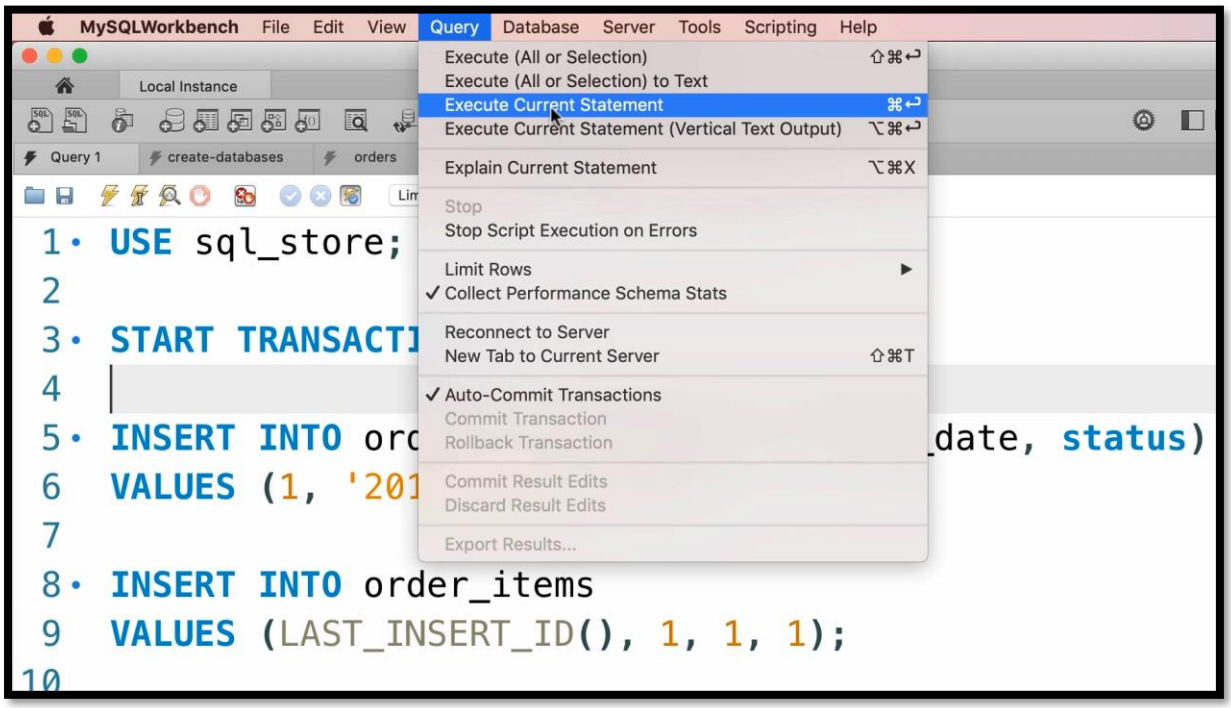
4. Durability
   Once a transaction is committed, changes made by a transaction are permanent. If there is a power failure or system crash, we are not going to lose the changes.



When MYSQL encounter this transaction, it will write all the changes to the database. If one of the changes fail, it will automatically undo the previous changes and the transaction will rollback.

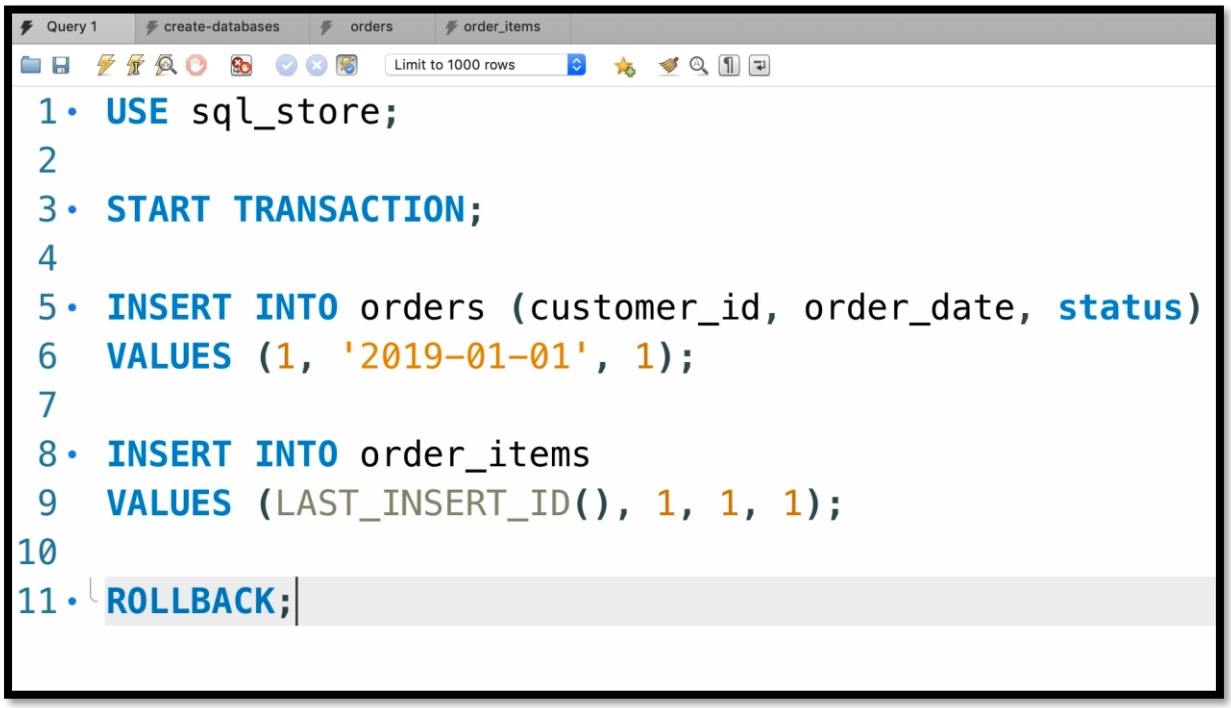
Let us simulate the scenario where second statement fails.

Let us execute our transaction line by line.



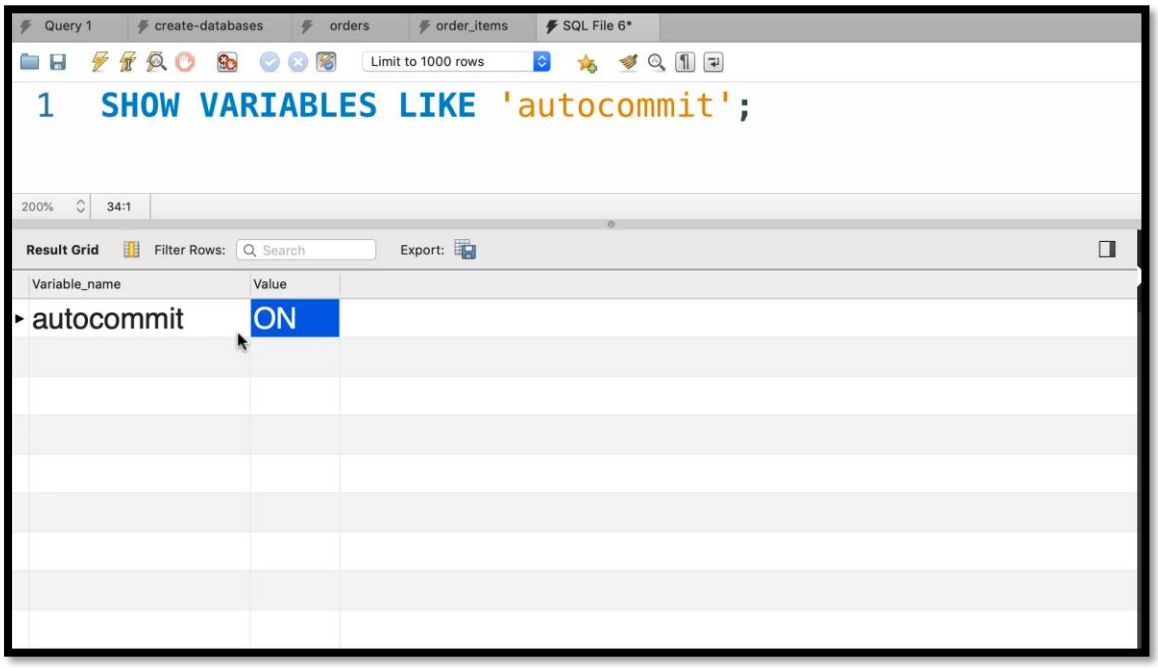
Let us disconnect the server after executing statement at line 5. In this scenario the transaction will rollback.

There are situations, we want some error checking and manually rollback the transaction. In that case, we use rollback instead of commit statement.



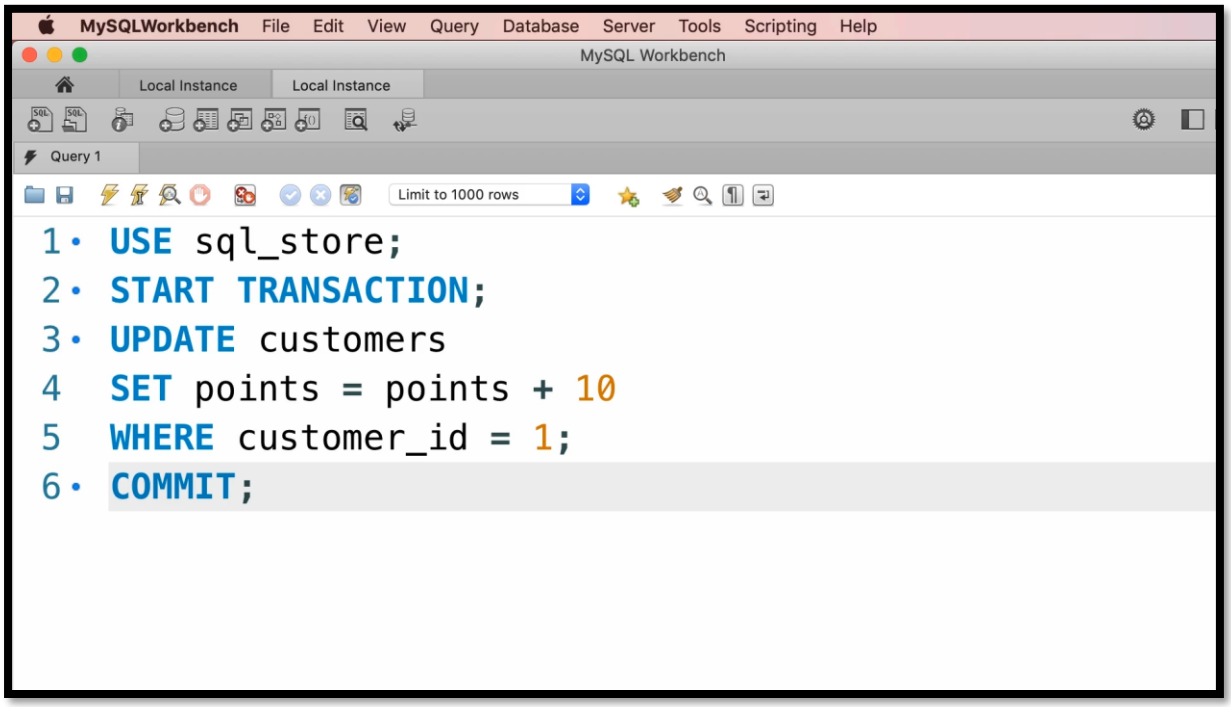
MYSQL wraps every single statement inside the transaction. If that transaction does not raise an error, it will do a commit automatically. This behaviour is controlled using a system variable i.e. AUTOCOMMIT.

SHOW VARIABLES LIKE "AUTOCOMMIT";



It is set to on by default.

Concurrency and Locking



Now don't execute it. Copy the code and paste it in a different session.

Let's execute the script line by line in first session. Let us run script from line 1 to line 5 but do not commit.

Let us move to second session. Because we have not committed in first session and if we run the update command in second session then it will go into eternal processing and ultimately timeout.

Let us run the commit command in first session and re run the script in second session.

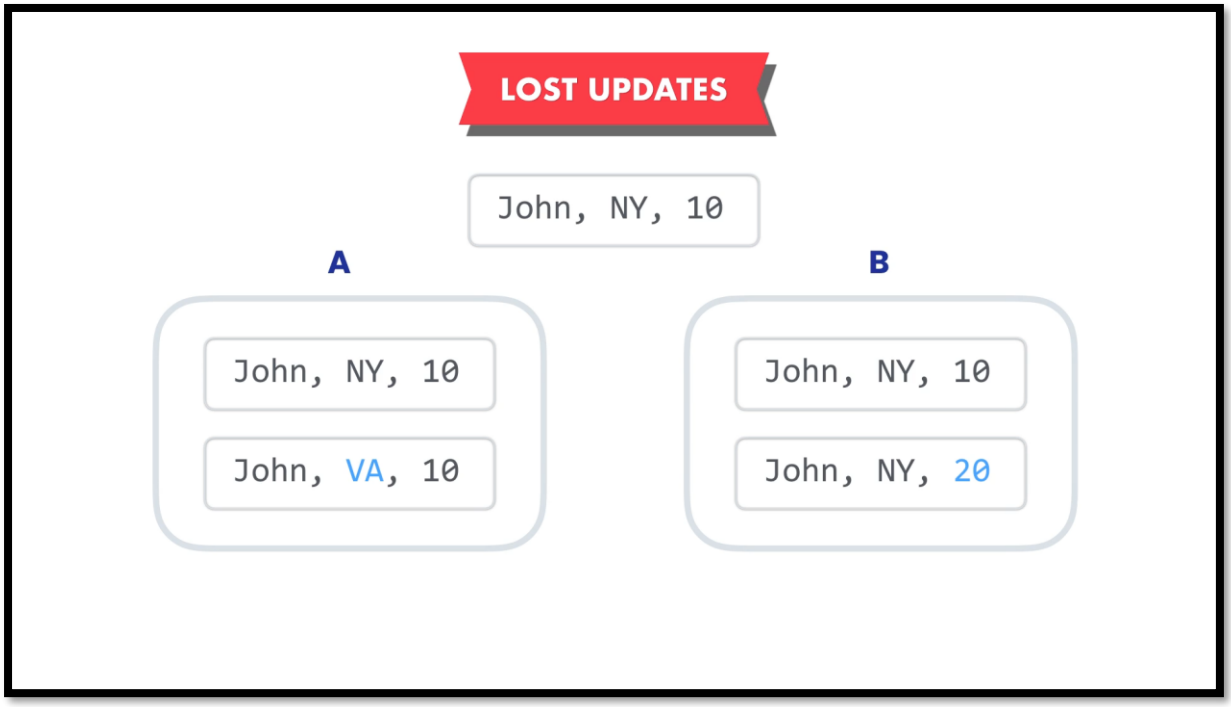
Originally the points of customer with id one were 2273 but now these are 2293. Because each transaction increased the points by ten.

So, a transaction puts a lock and this lock prevents other transactions to modify the data until its execution. Either it is committed or rollback.

So, with default concurrent behaviour MYSQL, we don't need to worry about concurrency problems.

Concurrency Problems

This happens when two transactions try to update the same data and we don’t use lock. In this situation the transaction that commits later will overwrite the changes made by previous transaction e.g. let us say we have to transactions that try to update the same customer. One tries to increase the points and other updates the state of the customer.

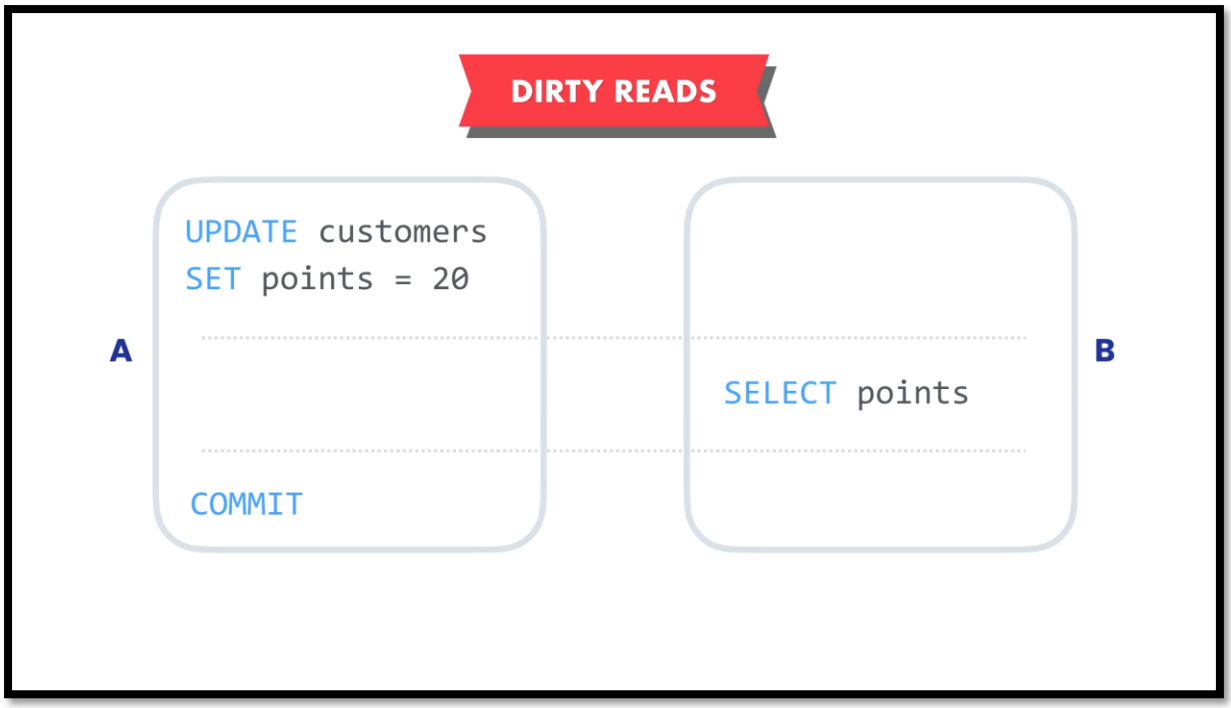


Transaction A tries to update the state and transaction B tries to update the points. These two transactions occur at the same time.

Let us say transaction A has update the state but it has not been committed yet. At the same time transaction B updates the points. Transaction that commits last over write the changes made earlier.

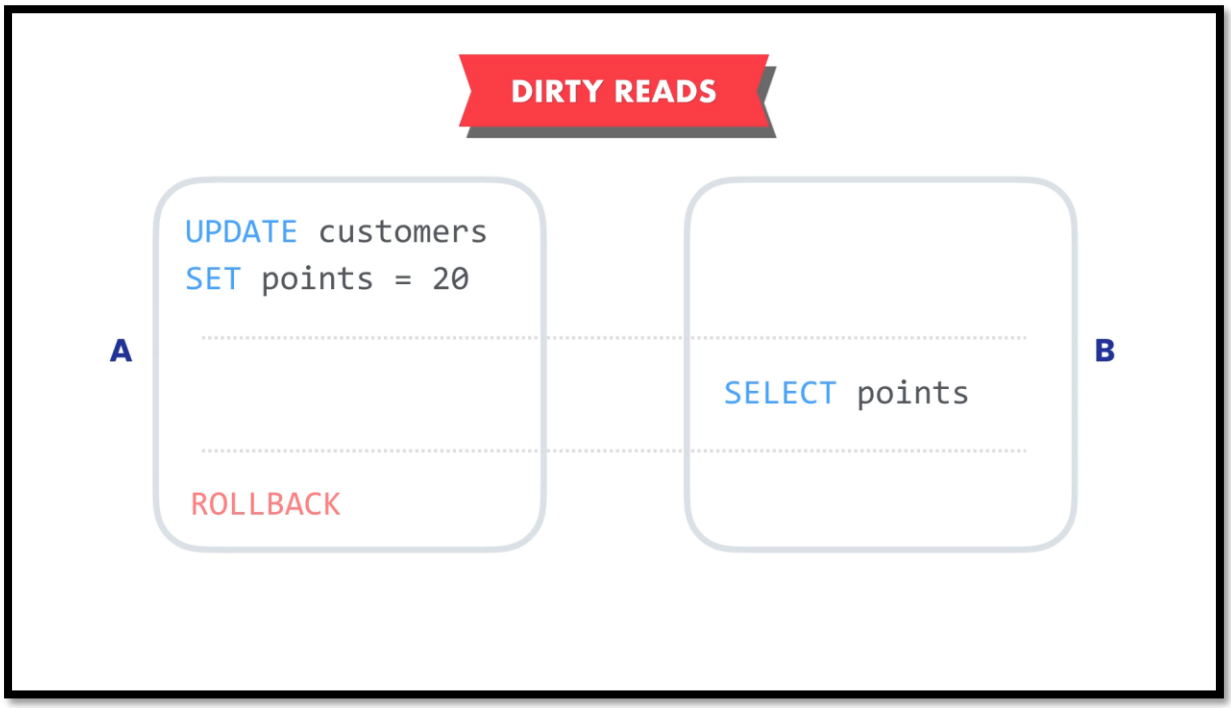
How can we prevent this?
Use locks.

Dirty reads are a situation when a transaction reads the data of another transaction that has not been committed yet.



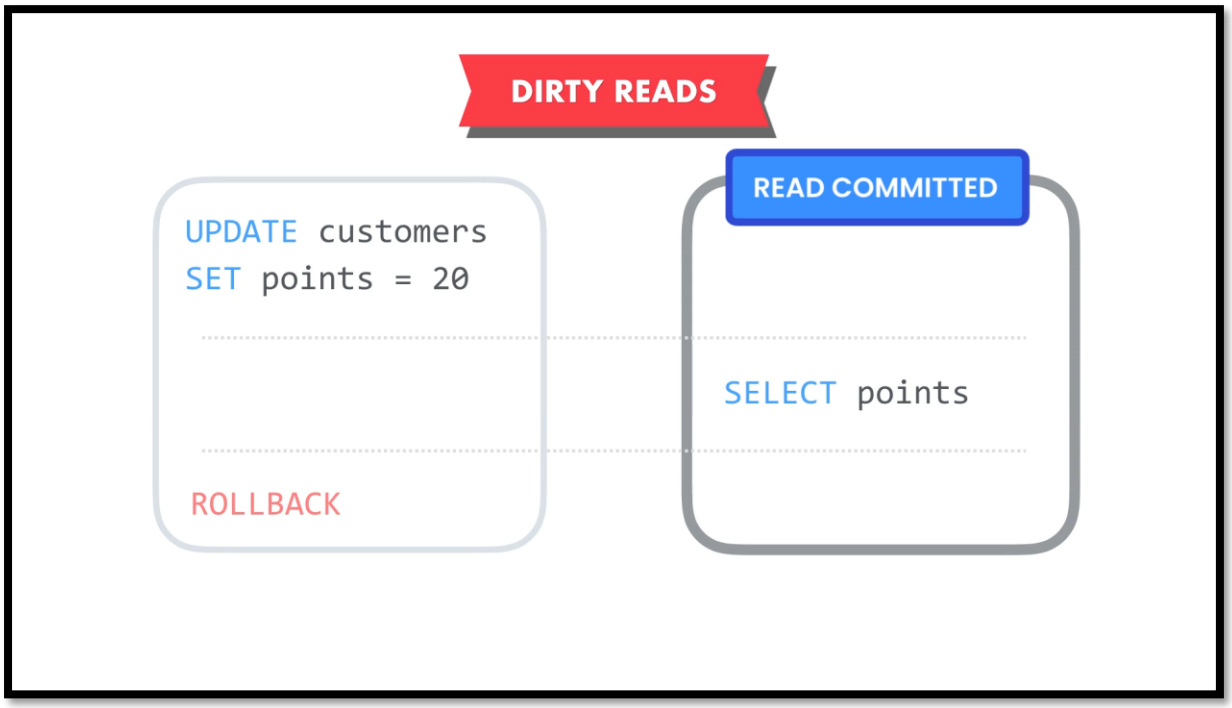
Transaction A changes the points for customer from 10 to 20. But before it commits the change, transaction B reads this customer and on these points make a decision.

Let us say for every one point, it gives a one rupee discount to a customer. So, it is going to give a customer Rs20/- discount.

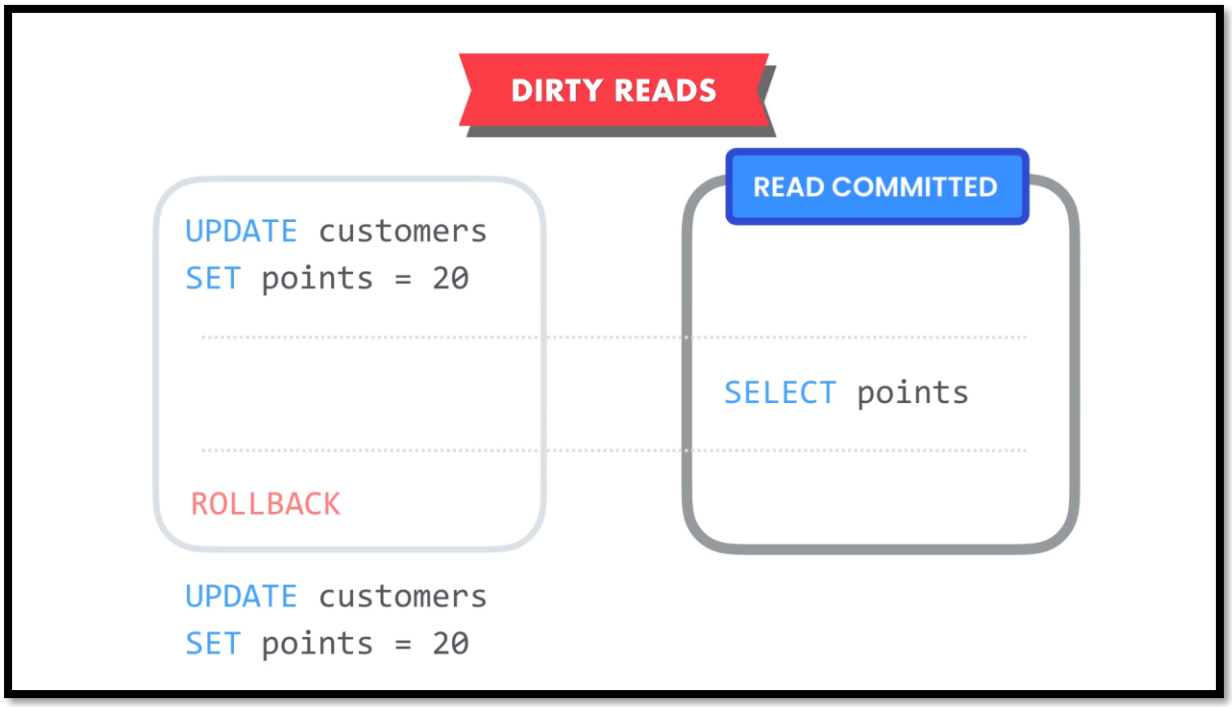


What if transaction A rollbacks before transaction B completes. Transaction B would have data that never has existed. So, in this case we

have read the uncommitted data in transaction B. Our data was dirty. To solve this problem we need to provide different level of isolations.

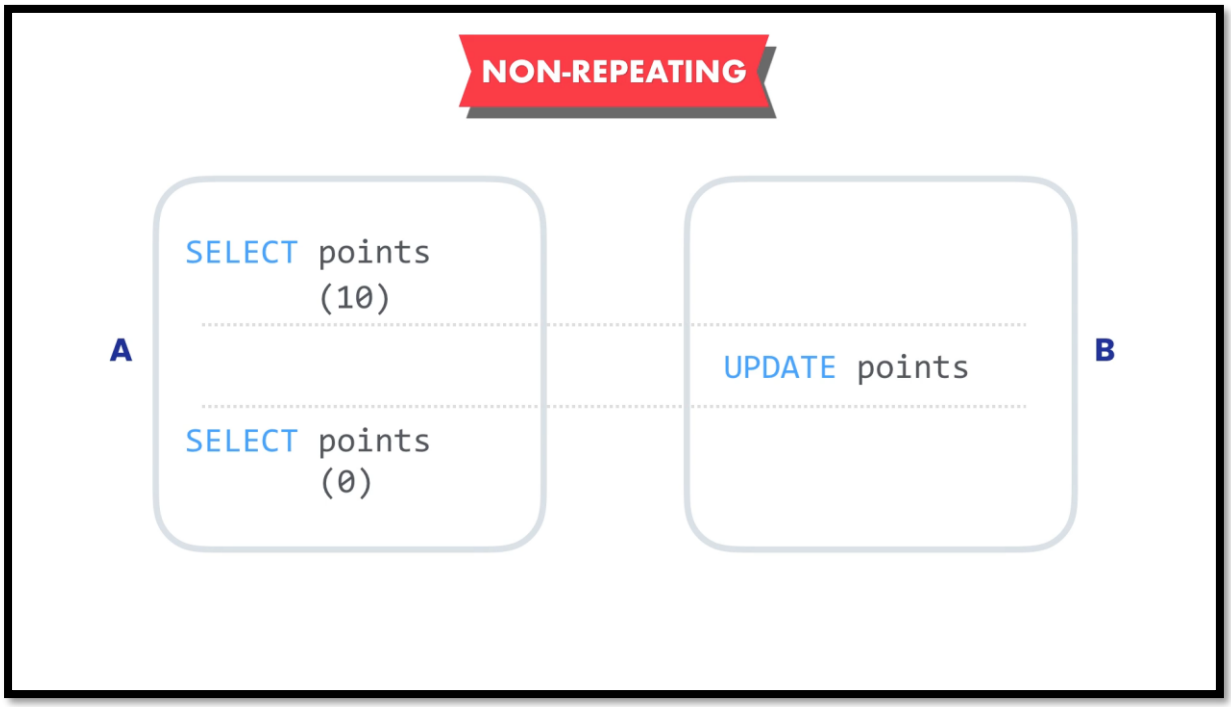


One of the isolation level is READ COMMITTED. So that transaction B could not read uncommitted data i.e. dirty data.



What if data gets change after the transaction A completes?
It does not matter.

READ COMMITTED ensures that transaction B could not read un committed data. In other words, transaction B can read only committed data. What if during course of transactions, you read something twice and get different results.



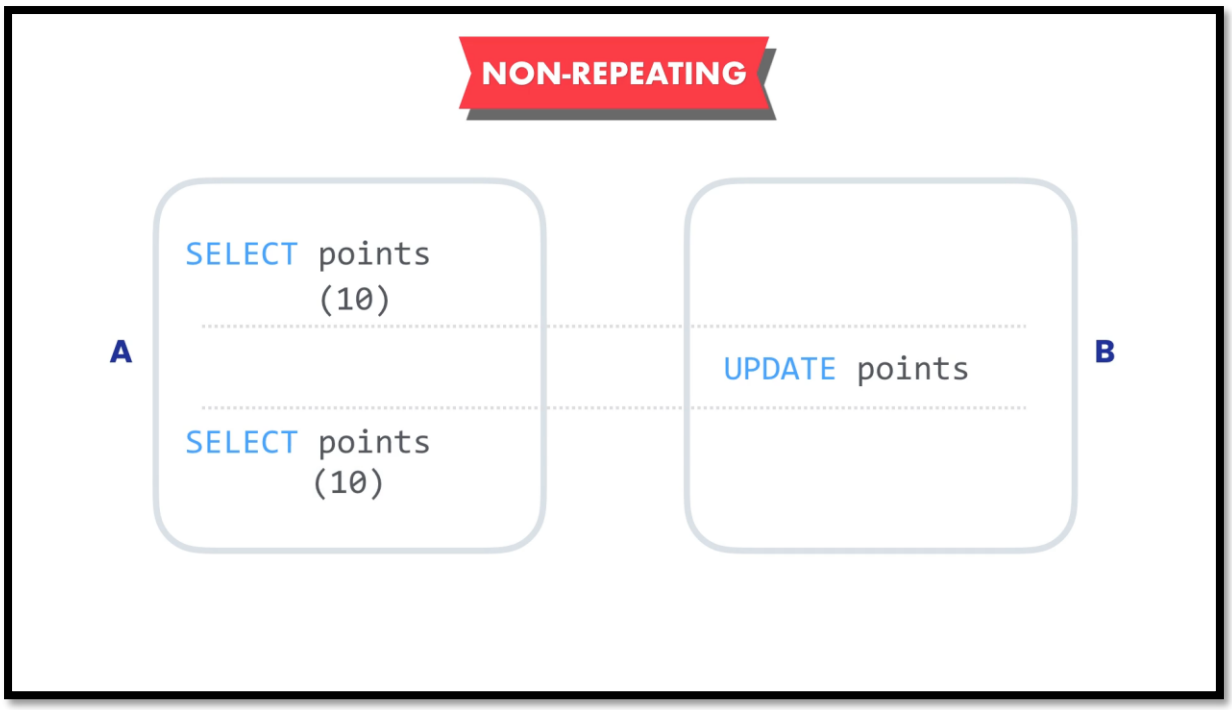
Transaction A reads the customer points. It says customer has 10 points. So, it makes business decision based on this value. Before transaction A completes, transaction B updates the points and set it zero. Back to transaction A, we read the points zero.

In this transaction A we read the points twice and each time we have seen a different value. This is called as non-repeatable or inconsistent read.

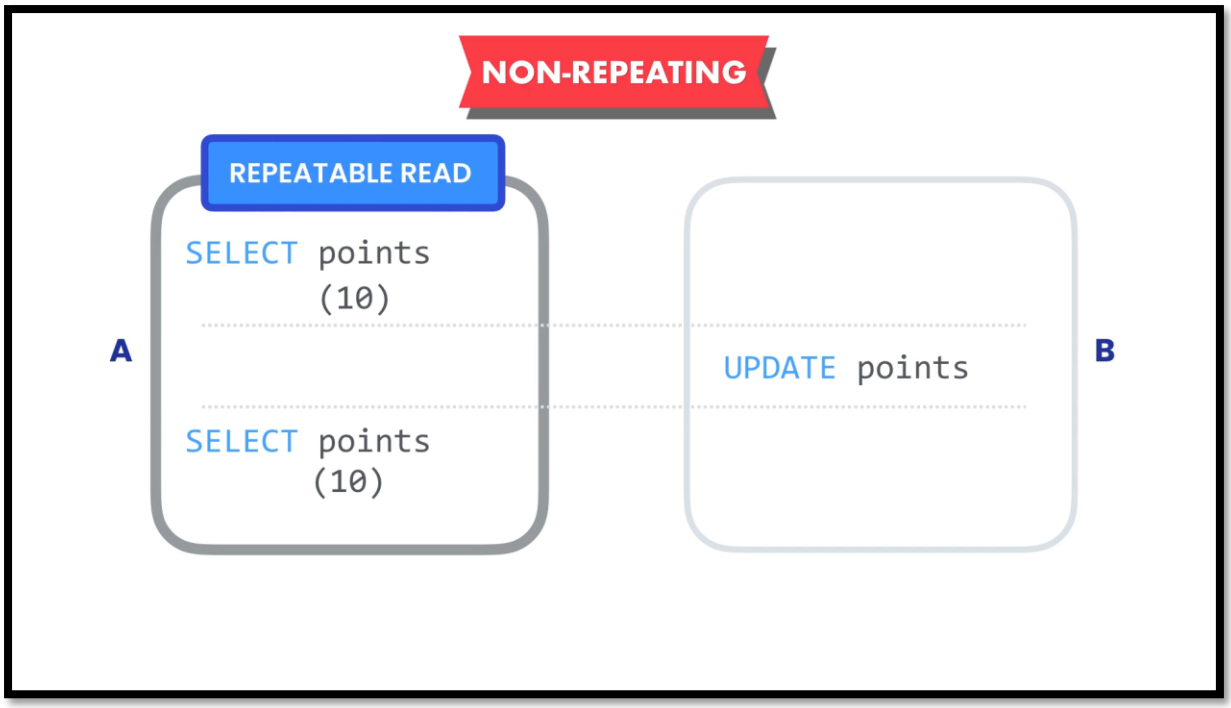
There can be two possibilities.

We can argue that any point in time, we should make decision based on what is most up to date. If that is the case, we do not need to worry here.

We can also argue that at the point when transaction started, the customer had ten points. So, we should give him Rs10/- discount.



If the point changes during this transaction, we should not see the changes. We should see the initial snapshot. If that is the case, we need to increase the isolation level. We need to isolate the transaction from other transactions such that changes to data are not visible to our transaction.



The SQL standards define another isolation level called REPETABLE READS. With this read, our reads are repeatable and consistent even if data gets change by other transaction.

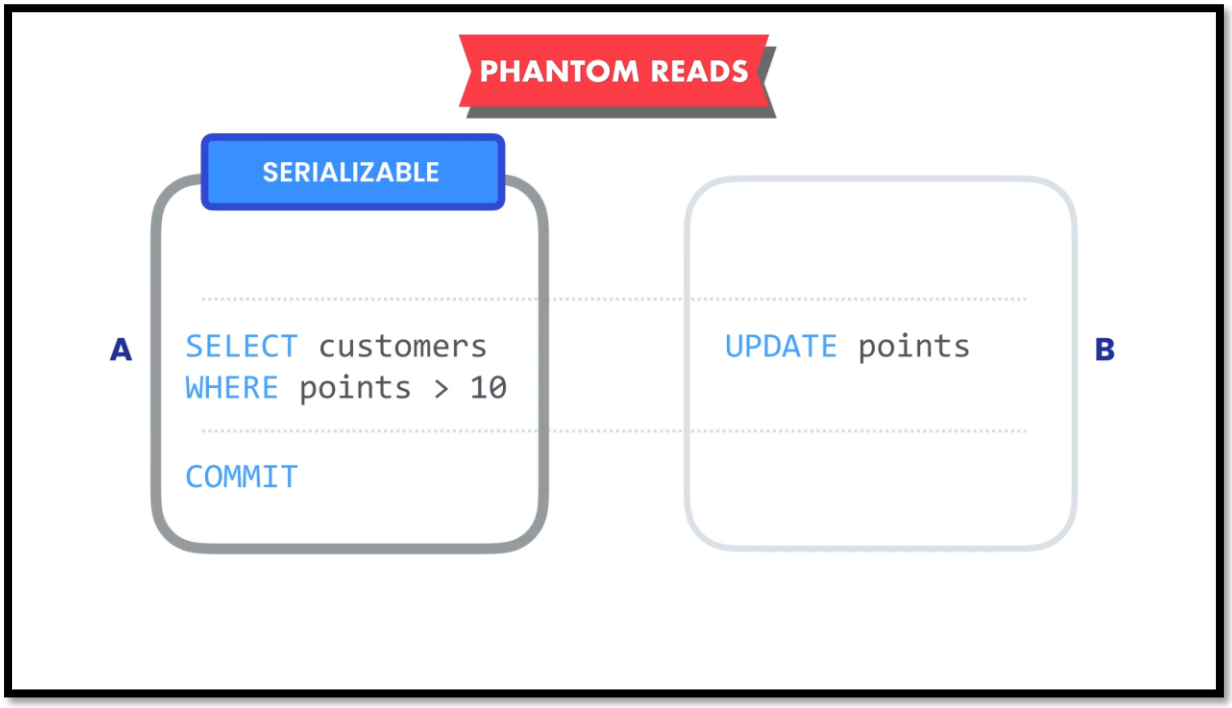
Imagine transaction A where we are querying customers with more than ten points. We want to send them special discount code.

At the same time transaction B updates the points for another customer that was not returned by query in transaction A. So, this customer too is now eligible for this discount code.

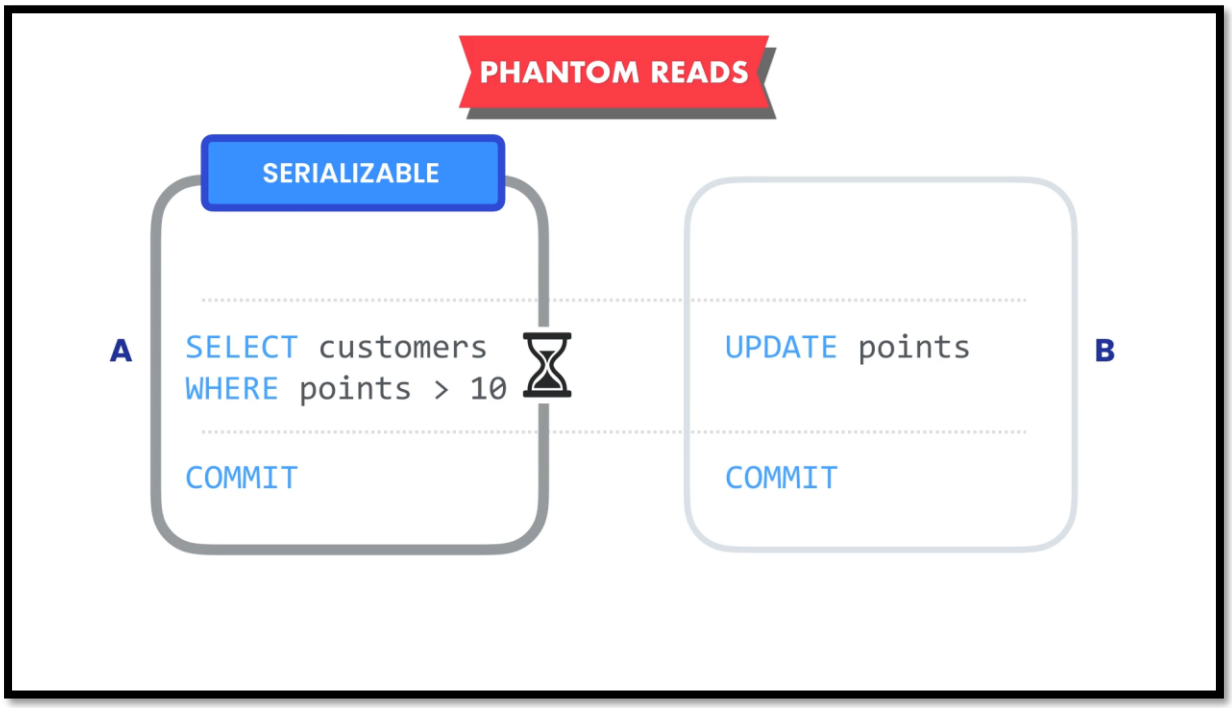
The time we were querying the eligible customers, we did not encounter this customer. After transaction A completes, there is a still one

eligible customer who did not receive the discount code. This is called as phantom read.

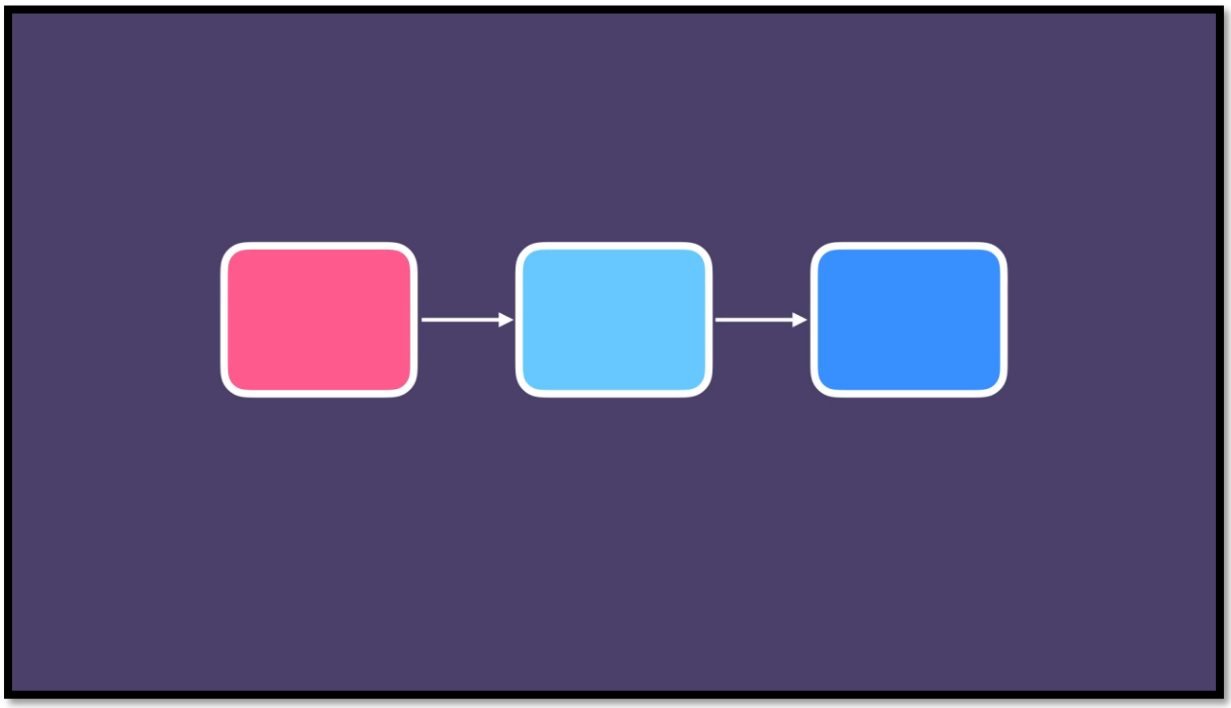
Phantom means ghost. We have data that suddenly appears like a ghost and we missed that data in the query.



We have another isolation level called as SERIALIZABLE. It ensures that no other transaction is running that can impact our query to find the eligible customers.



If there is a transaction running that can impact our query to find the eligible customers, then our query in transaction A has to wait till completion of transaction B.



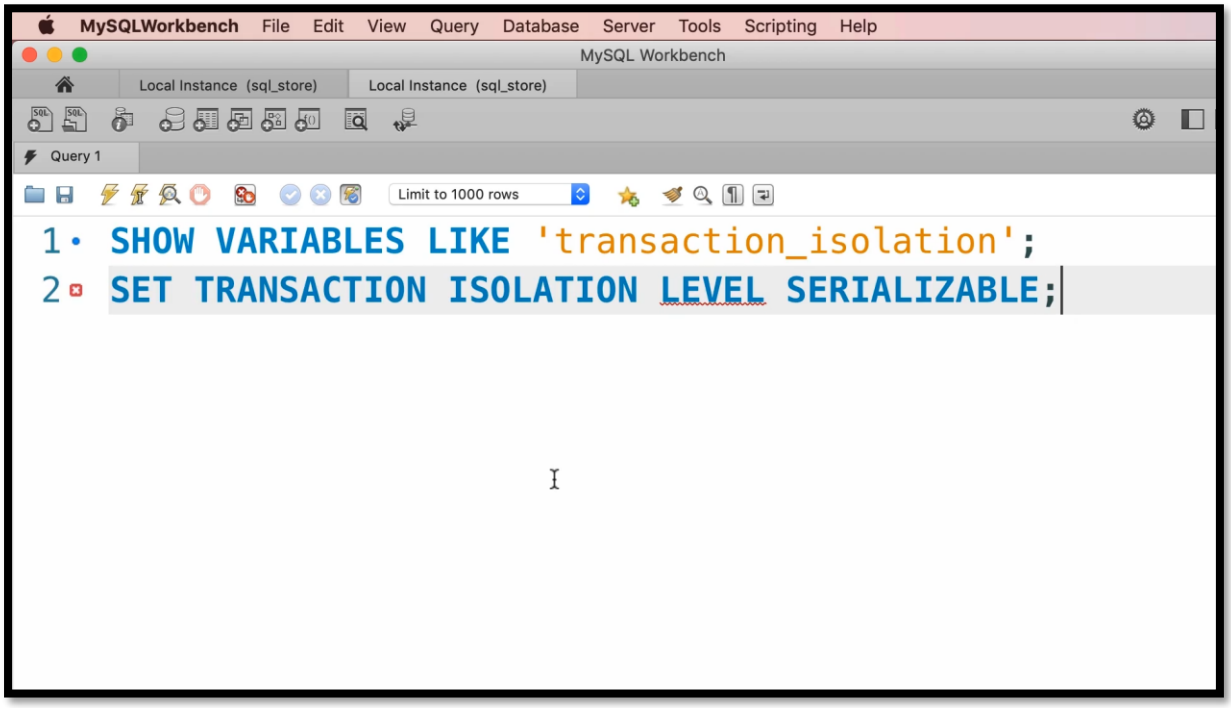
Transactions should be executed sequentially. This is the highest level of isolation that we can apply to a transaction.

READ UNCOMMITTED does not protect us from any concurrency problems because our transactions are not isolated.

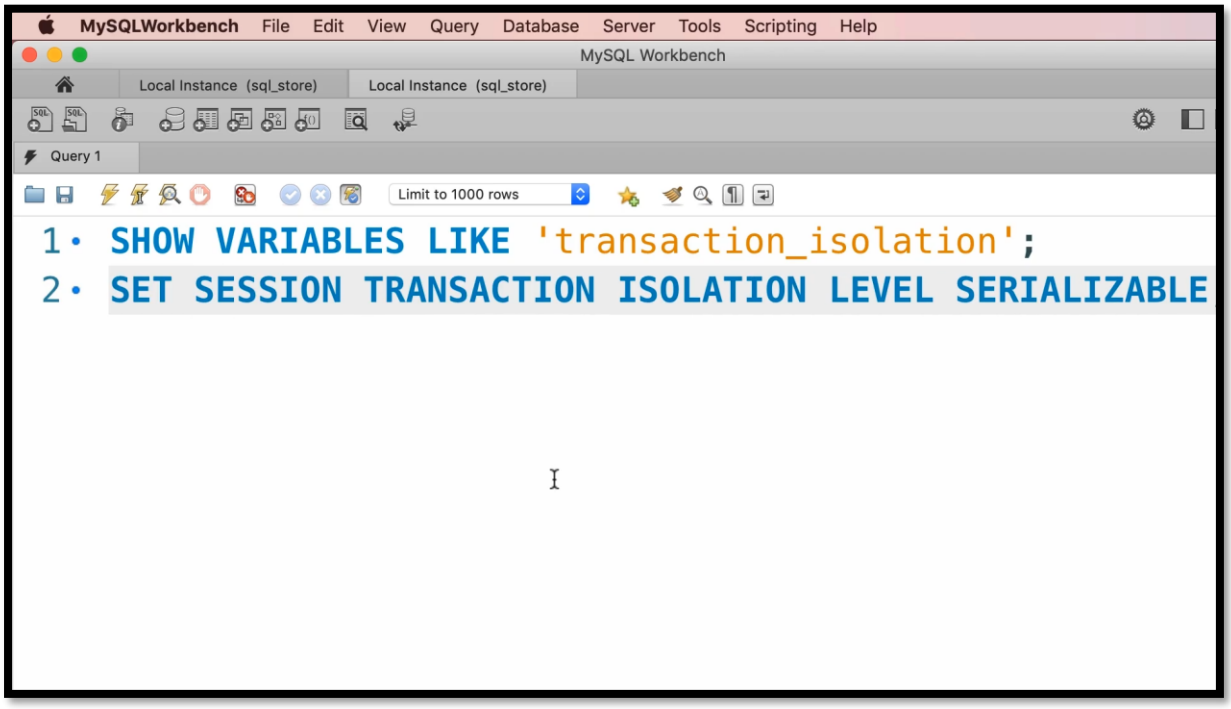
READ COMMITTED can read only committed data and this prevents dirty reads only.

REPETABLE READ prevents lost updates, dirty reads and non-repeatable reads.

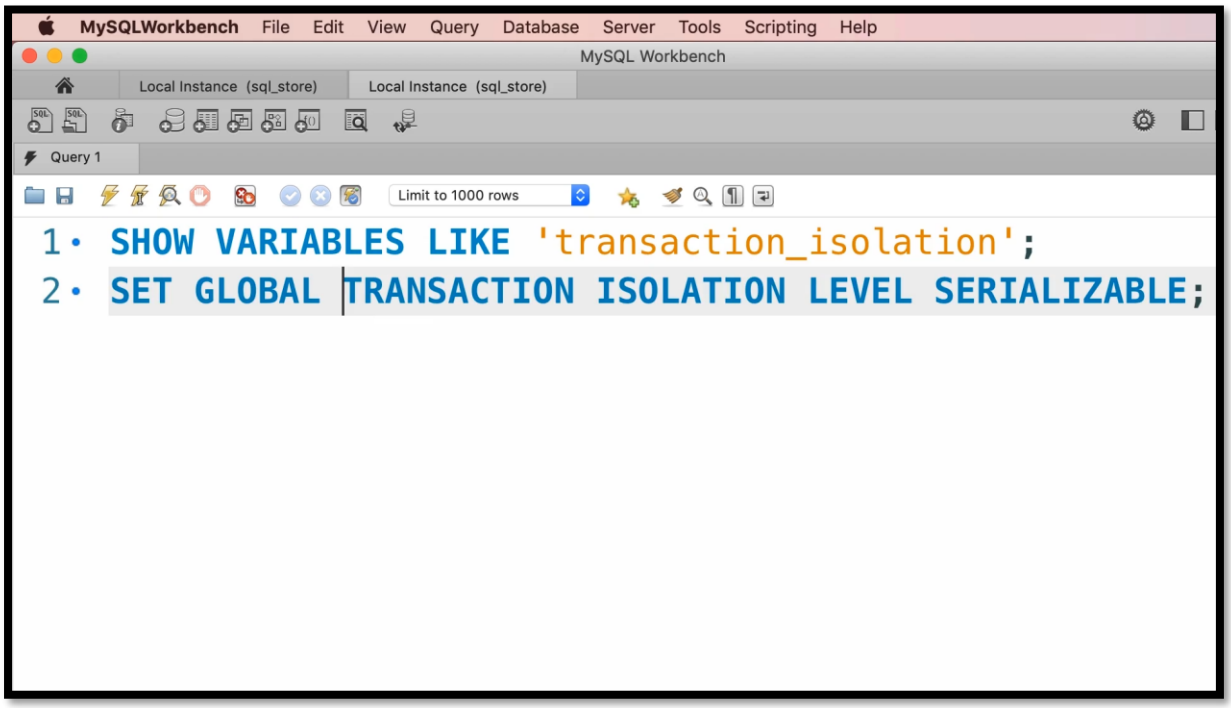
SERIALIZABLE READ prevents all the concurrency problems.



This script will set the isolation level for next transaction.

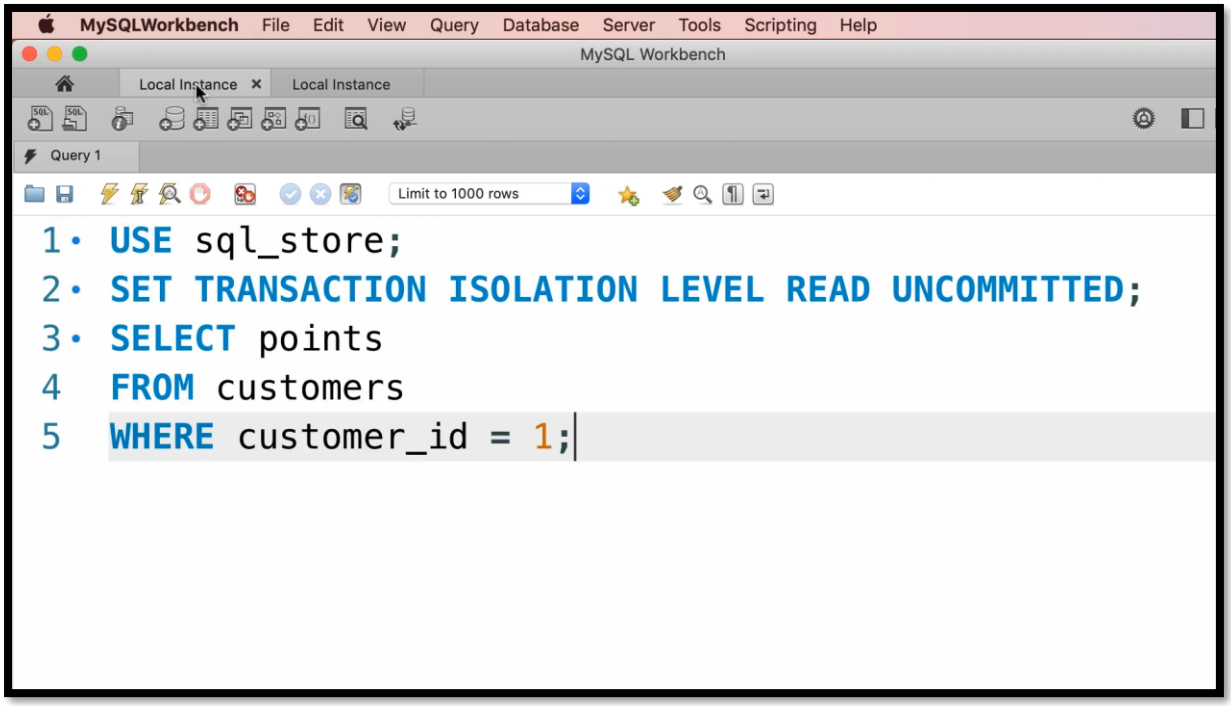


We can also set the isolation level for all future transactions in the current connection or session.

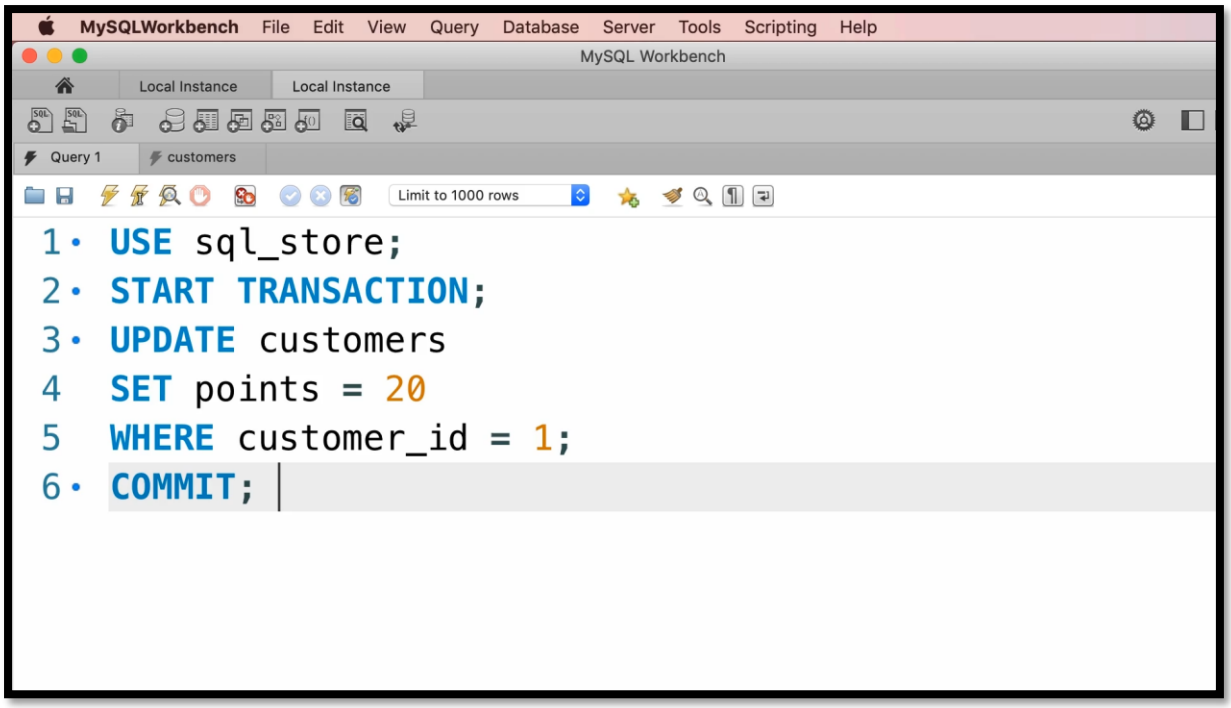


We can also set the isolation level globally for all new transactions in all sessions.

READ UNCOMMITTED



SESSION 1



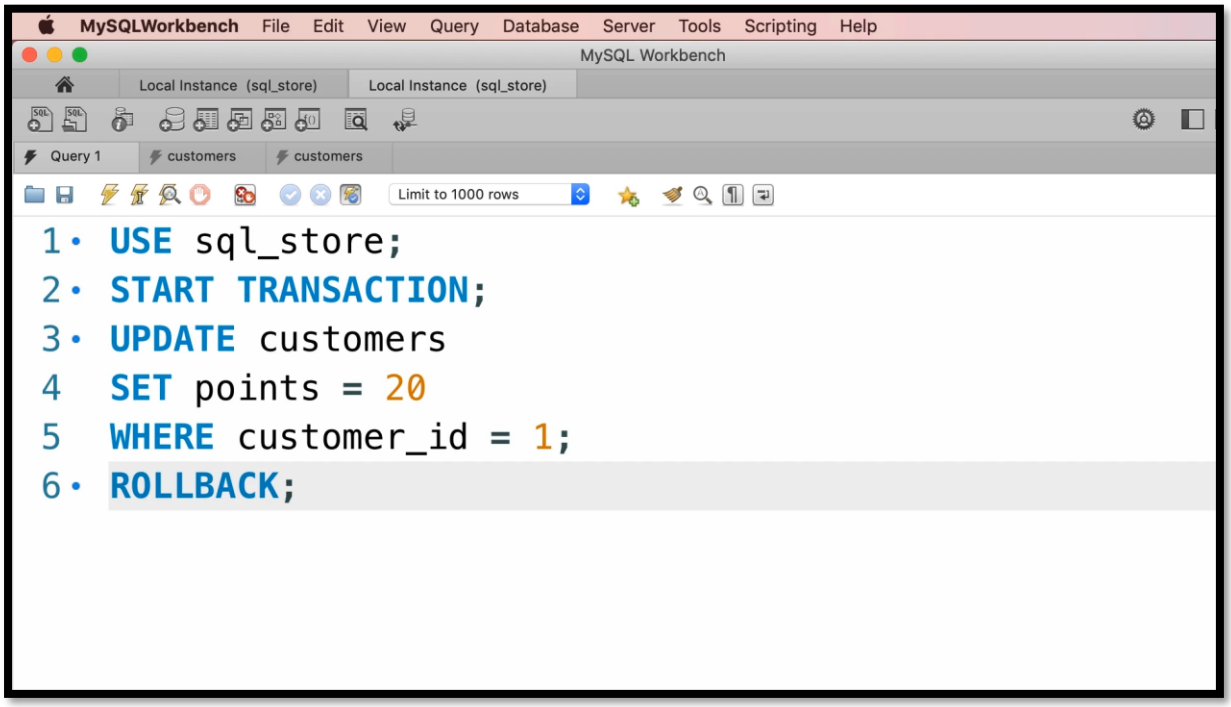
SESSION 2

Let us go to session 1 and run the script line by line. Let us run only line 1 and line 2 and go to session 2.

In session 2 run the whole transaction.

Now, back to session 1 and run the remaining script.

We will get 20 points but initially this customer has 2293 points. We might take business decision based on this value.

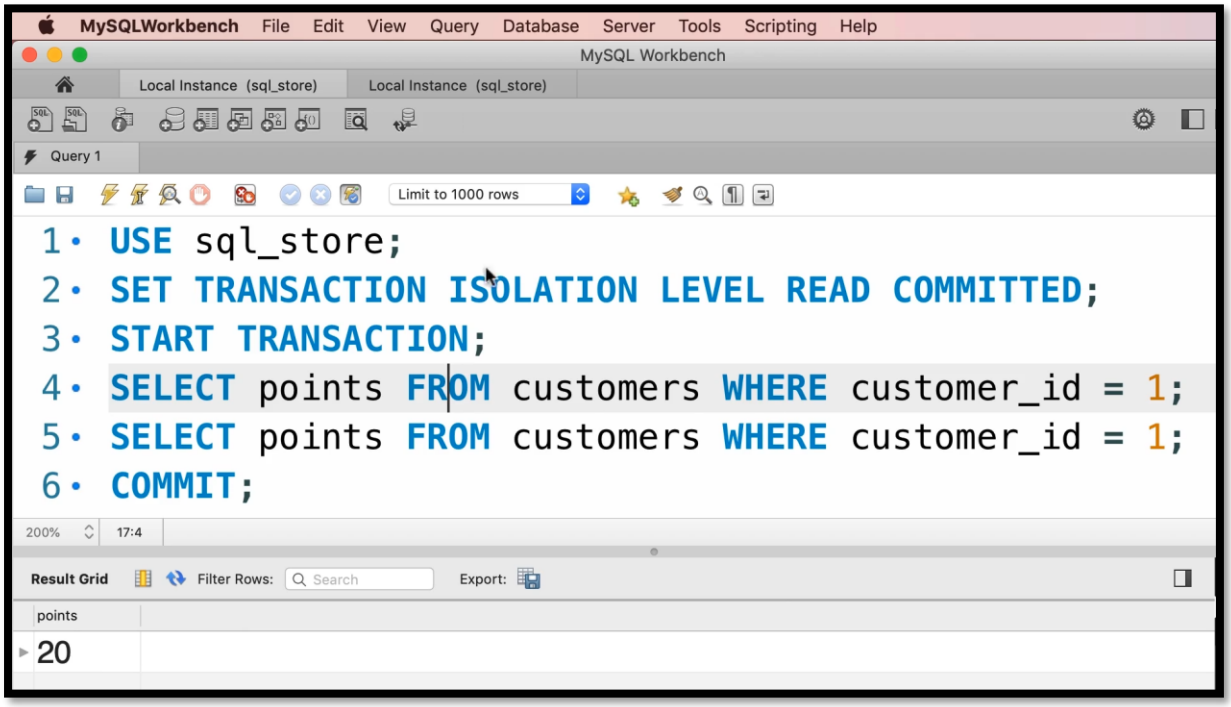


What if in session 2, the commit never happens, maybe the server crashes or transaction explicitly rollback.

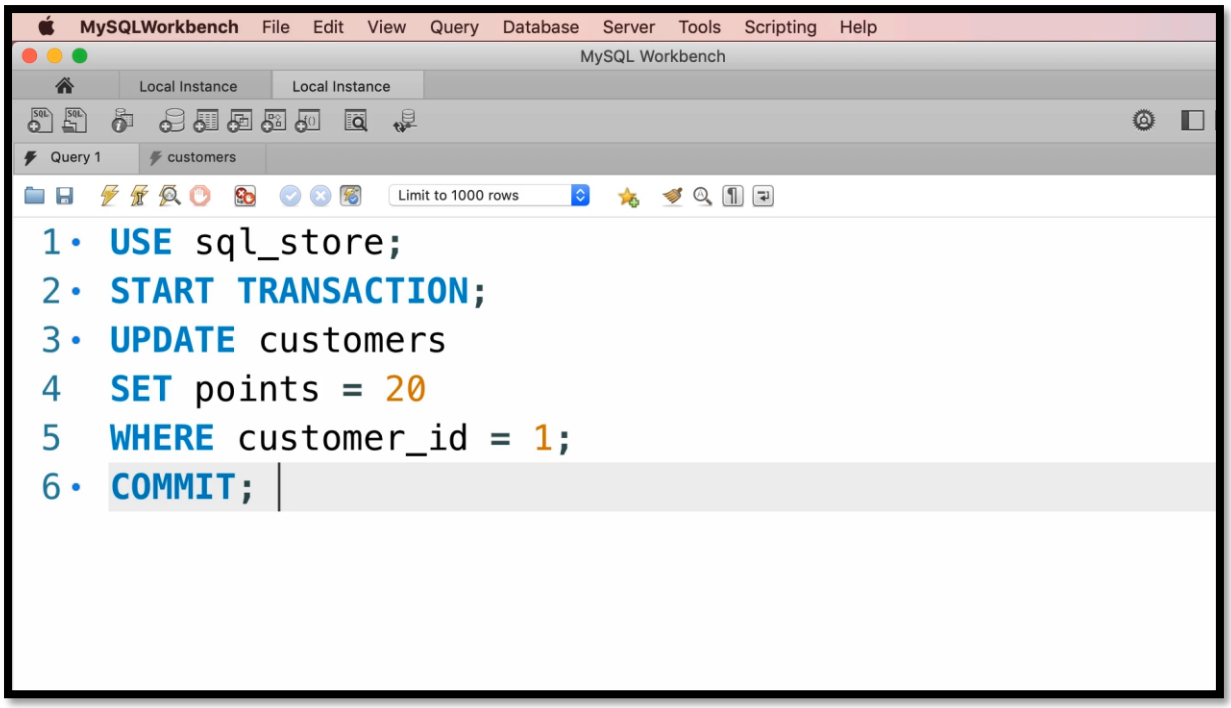
Now in session 1 we are dealing with the data that never existed in a database. This is called as dirty reads.

READ UNCOMMITTED is the lowest isolation level. With this isolation we may experience all concurrency problems.

READ COMMITTED



SESSION 1



SESSION 2

Let us go to session 1 and run the script line by line. Let us run only line 1, line 2, line 3 and line 4.

We will get 2293 points as initial points.

Let us go to session 2.

Let us run only line 1, line 2 and line 3.

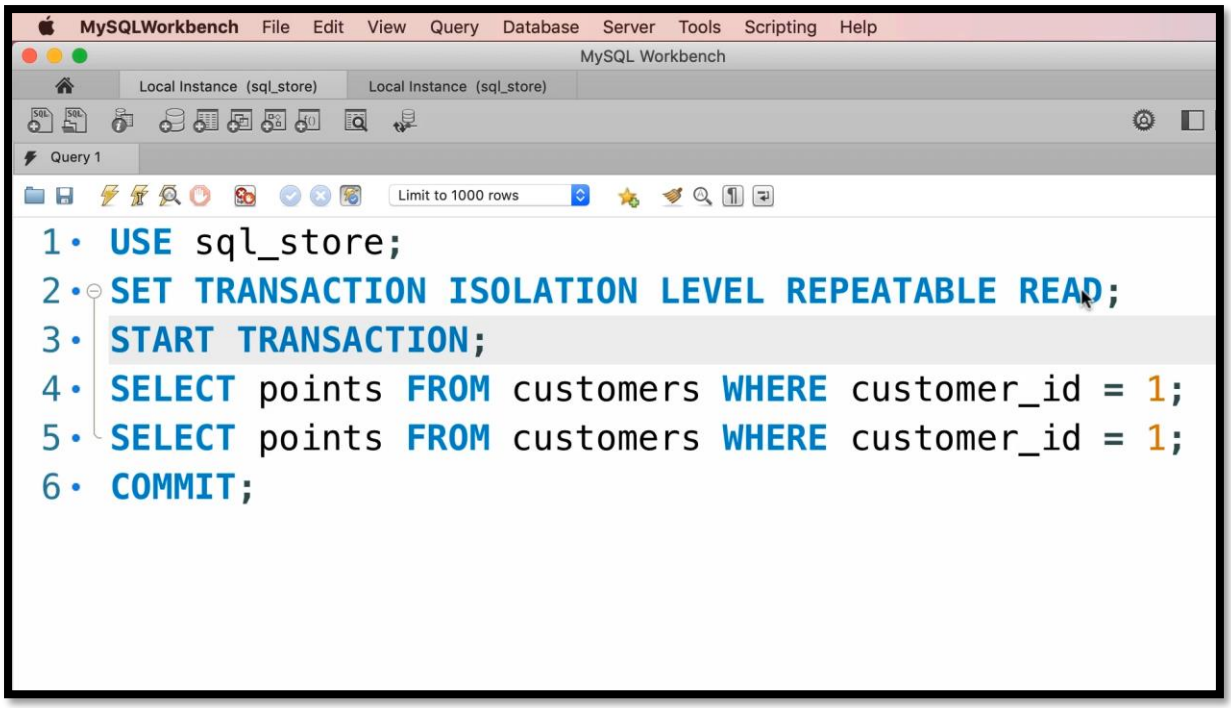
Let us go to session 1 and run the line 5. We will get 2293 points because uncommitted read is not allowed here.

Let us go to session 2 and run line 6 i.e. do a commit.

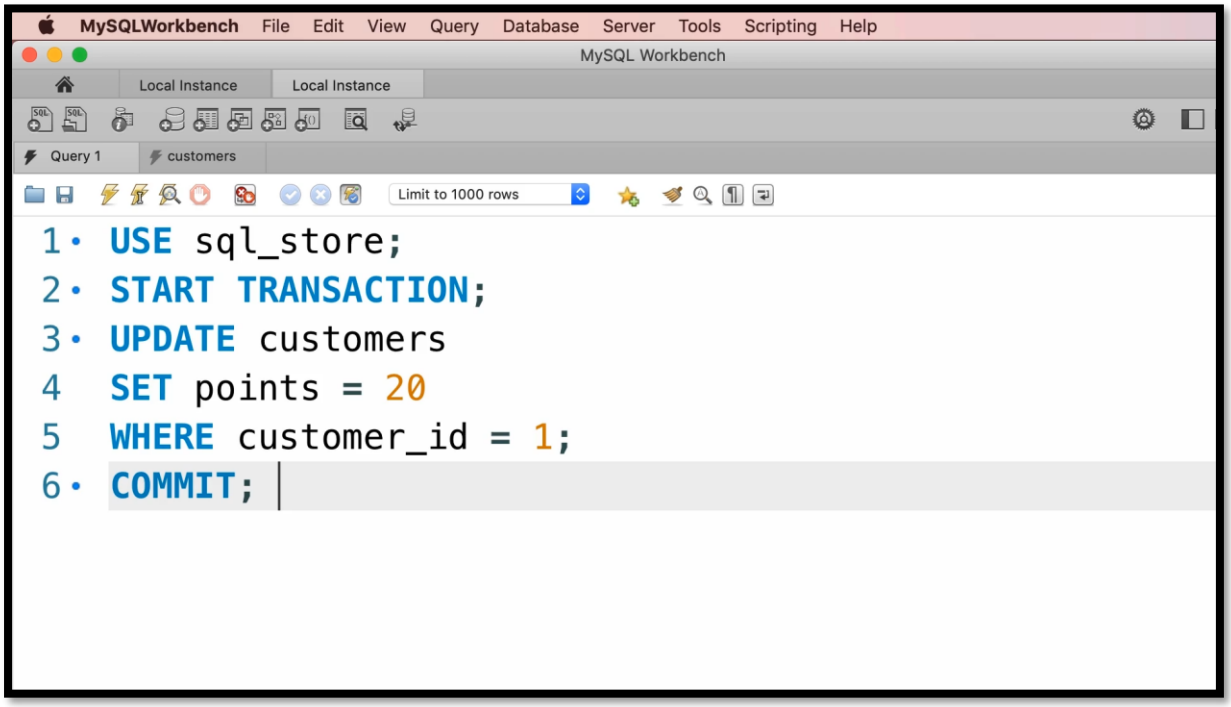
Let us go to session 1 and again run the line 5. We will get 20 points this time.

At this isolation level, we have non repeatable or inconsistent reads.

REPEATABLE READ



SESSION 1



SESSION 2

Let us go to session 1 and run the script line by line. Let us run only line 1, line 2, line 3 and line 4.

We will get 2293 points as initial points.

Let us go to session 2.

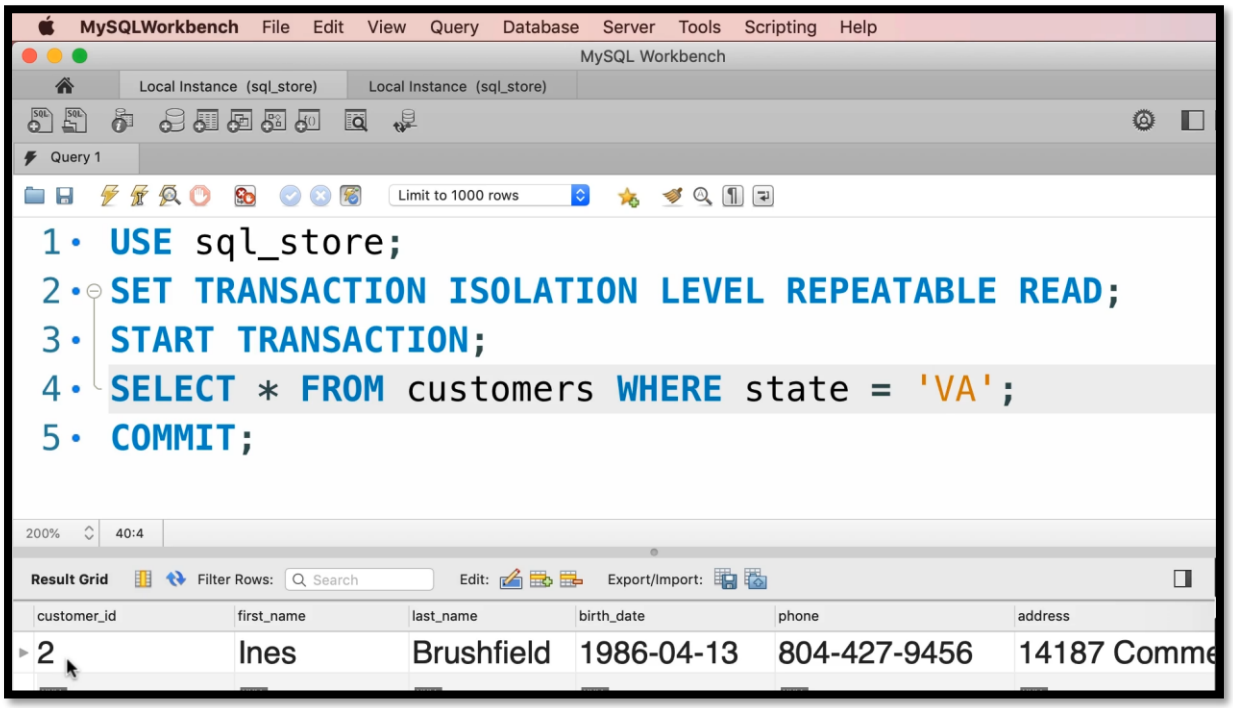
Let us run only line 1, line 2 and line 3.

Let us go to session 1 and run the line 5. We will get 2293 points because uncommitted read is not allowed here.

Let us go to session 2 and run line 6 i.e. do a commit.

Let us go to session 1 and again run the line 5. We will get again 2293 points because committed read is also not allowed here.

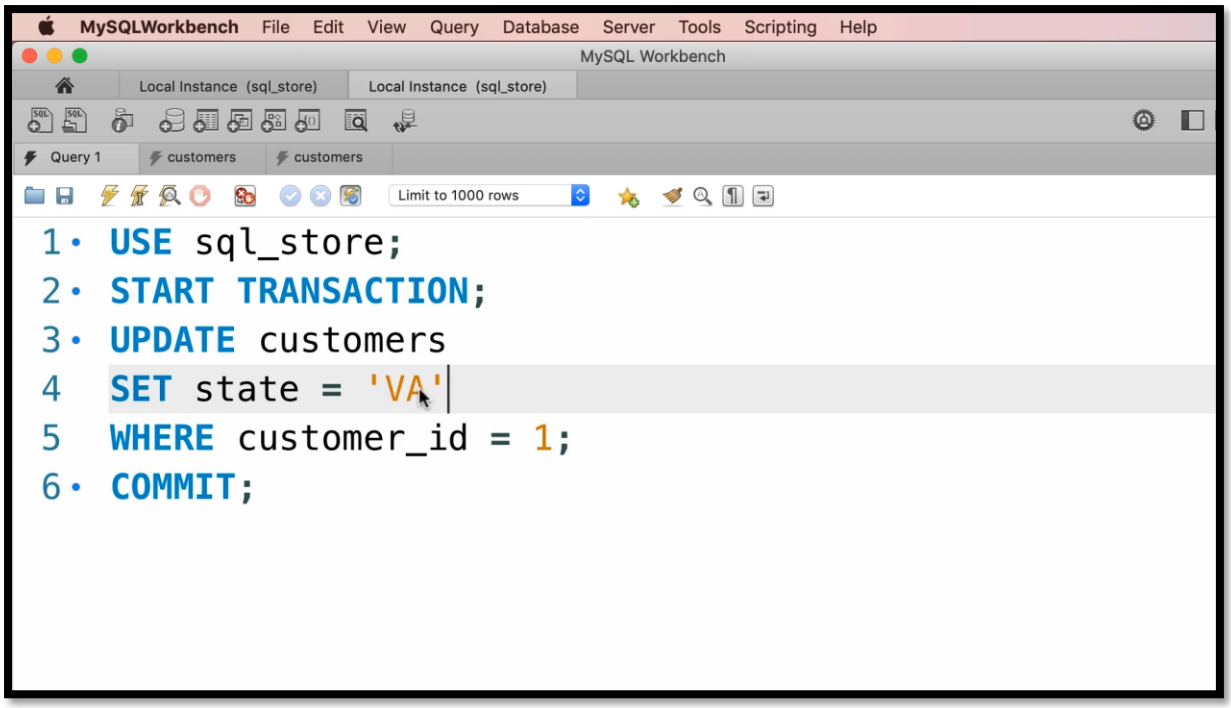
At this isolation level, the problem of non repeatable or inconsistent reads gets resolved but we have a problem of phantom reads.



SESSION 1

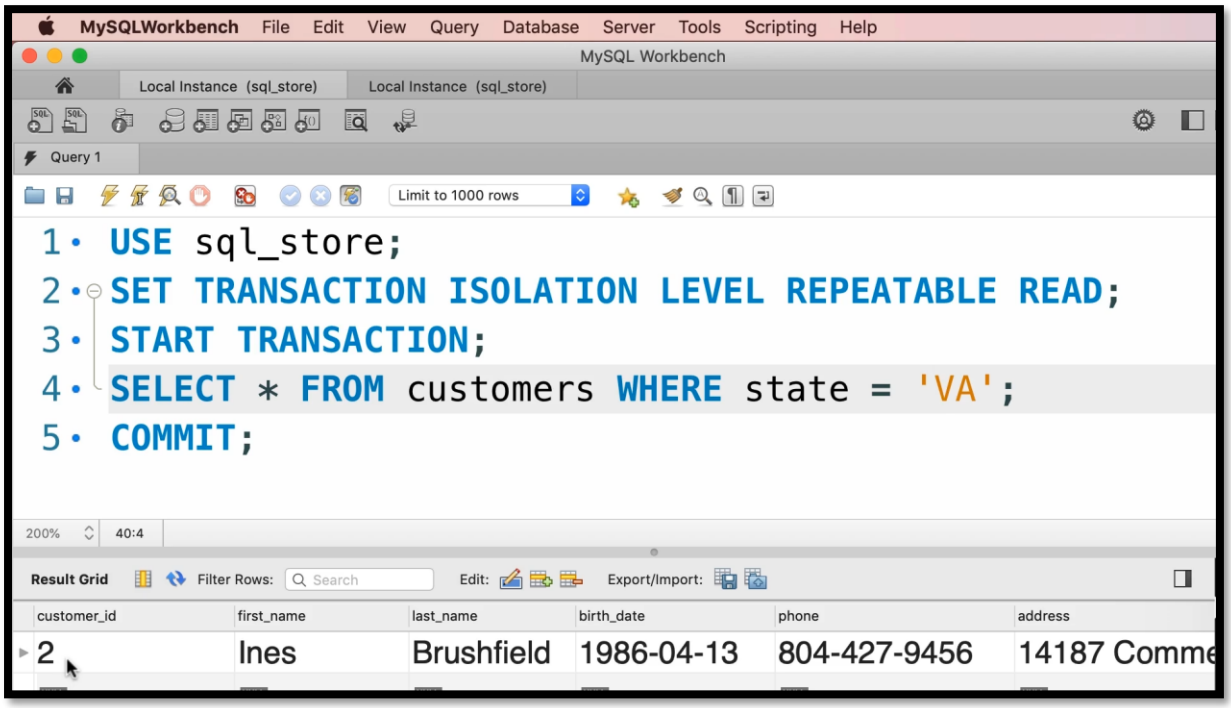
Let us run the script line by line. Let us run only line 1, line 2, line 3 and line 4.

We will get only customer with id 2 who lives in Virginia.

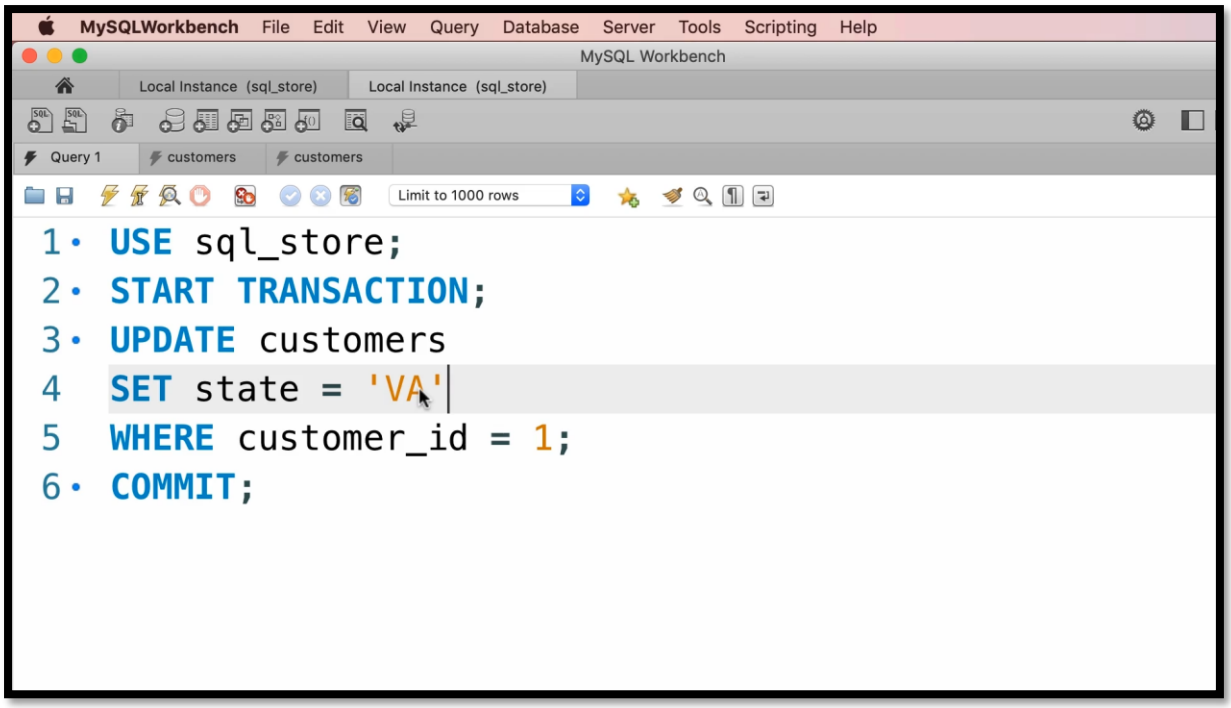


SESSION 2

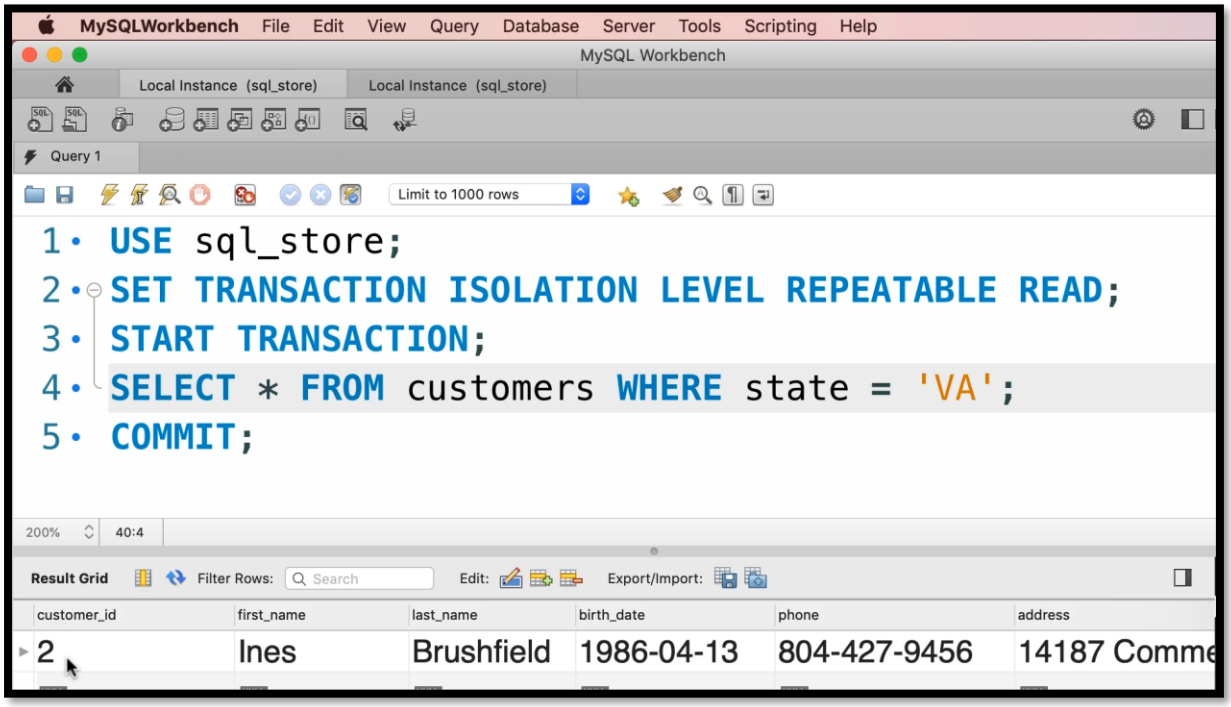
Let us run only line 1, line 2 and line 3.



Let us again run the line 4. We will get the same results.

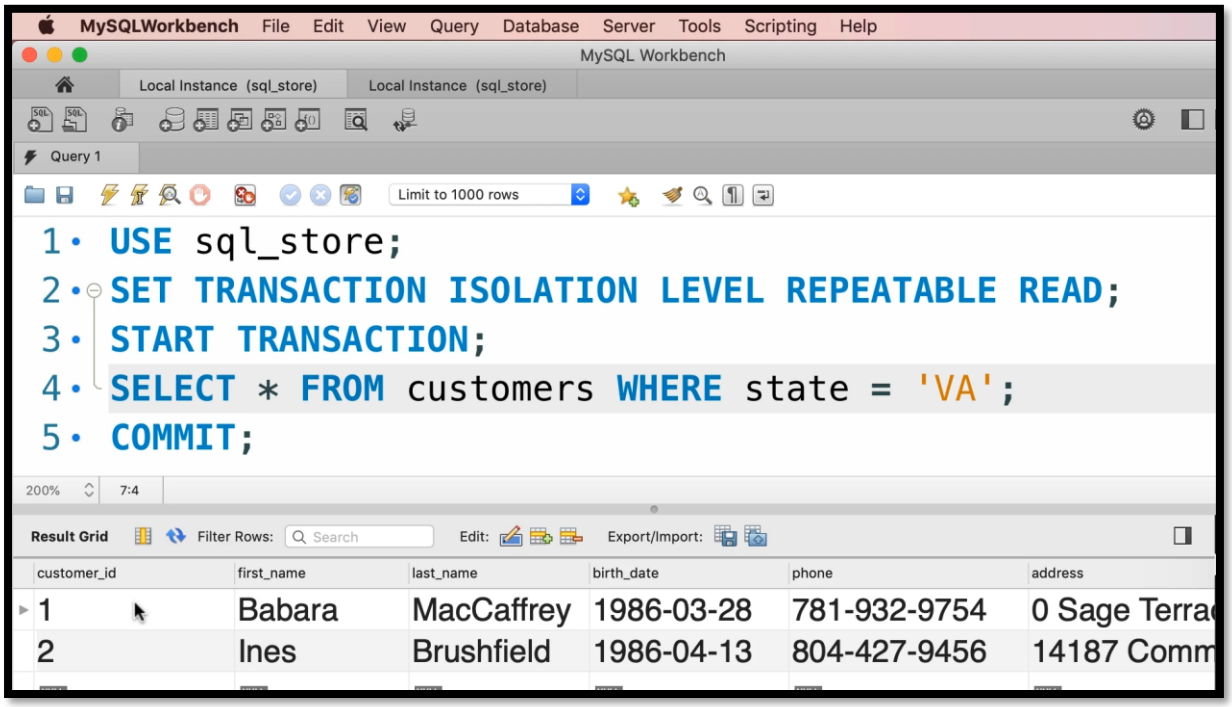


Let us do a commit.

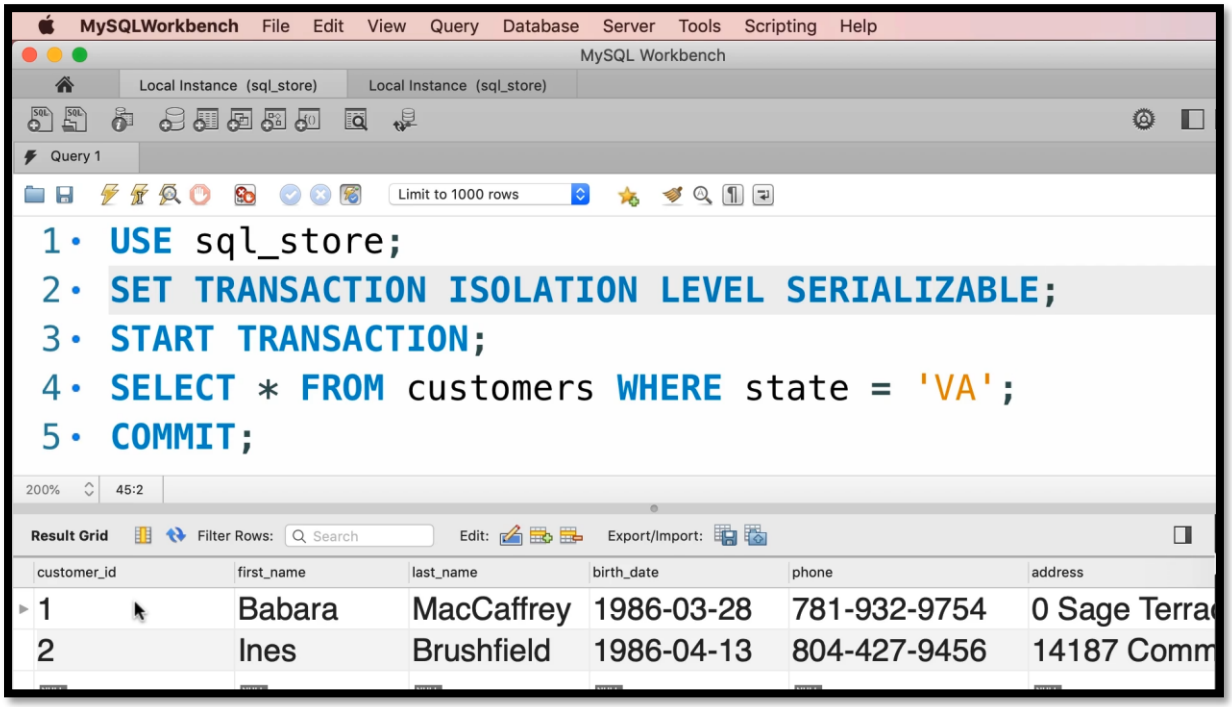


Let us again run the line 4. We will get the same results. We have missed one customer in this case. This is called PHANTOM READ.

SERIALIZABLE READ

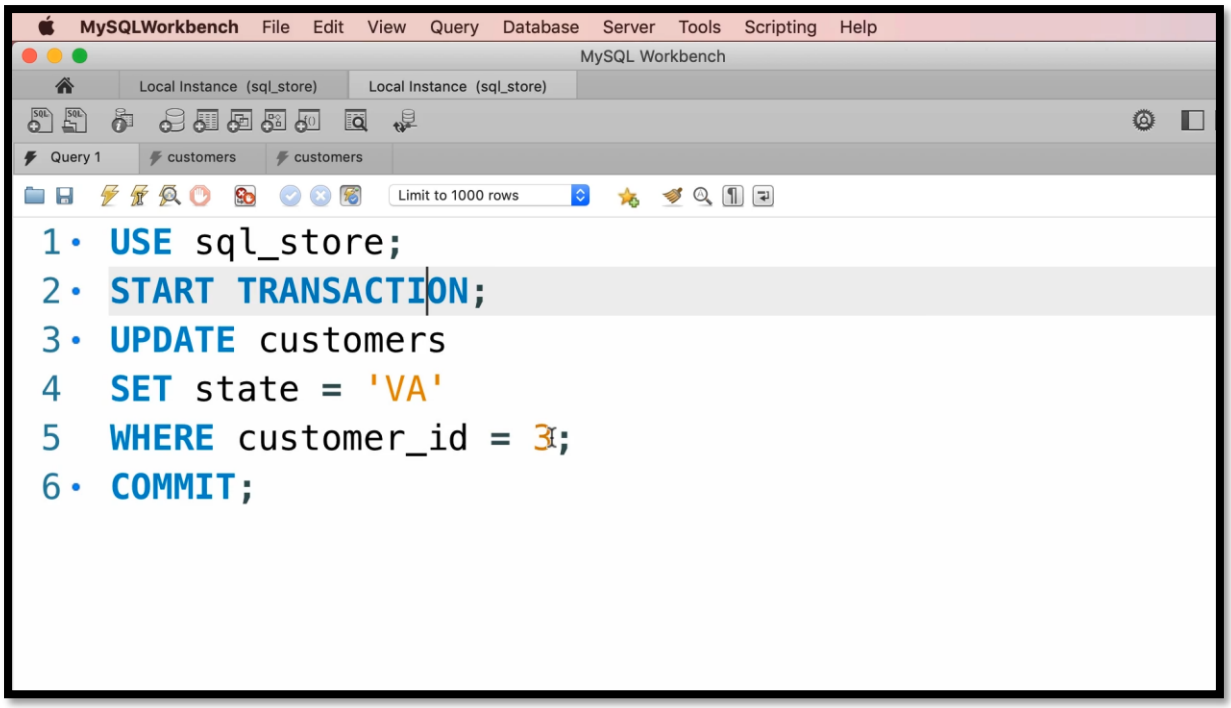


Currently we have only two customers who belong to state Virginia.



SESSION 1

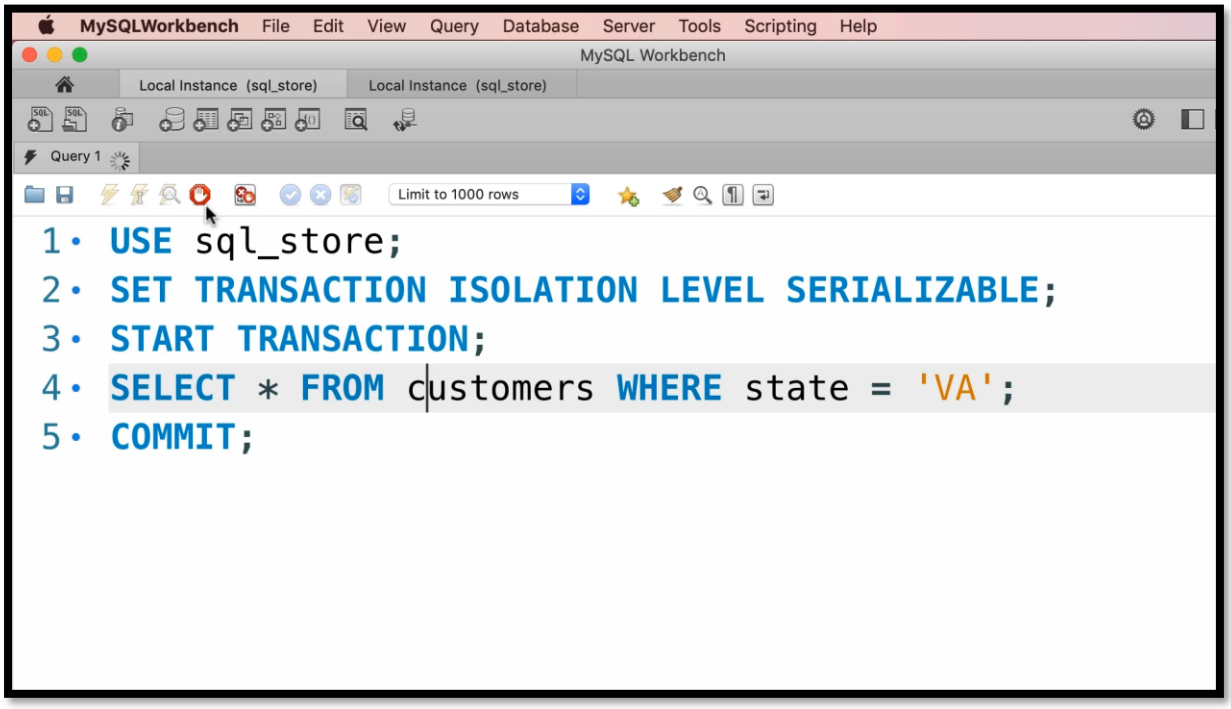
Let us run line 1, line 2 and line 3.



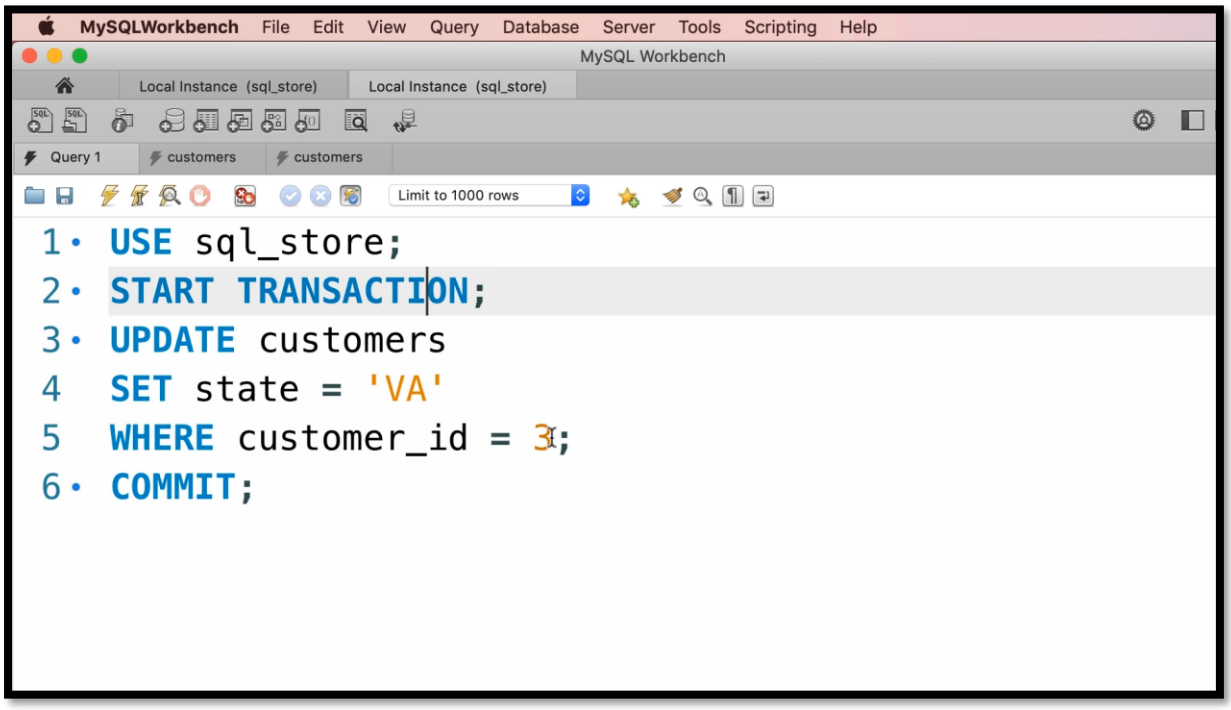
SESSION 2

Let us run line 1, line 2 and line 3.

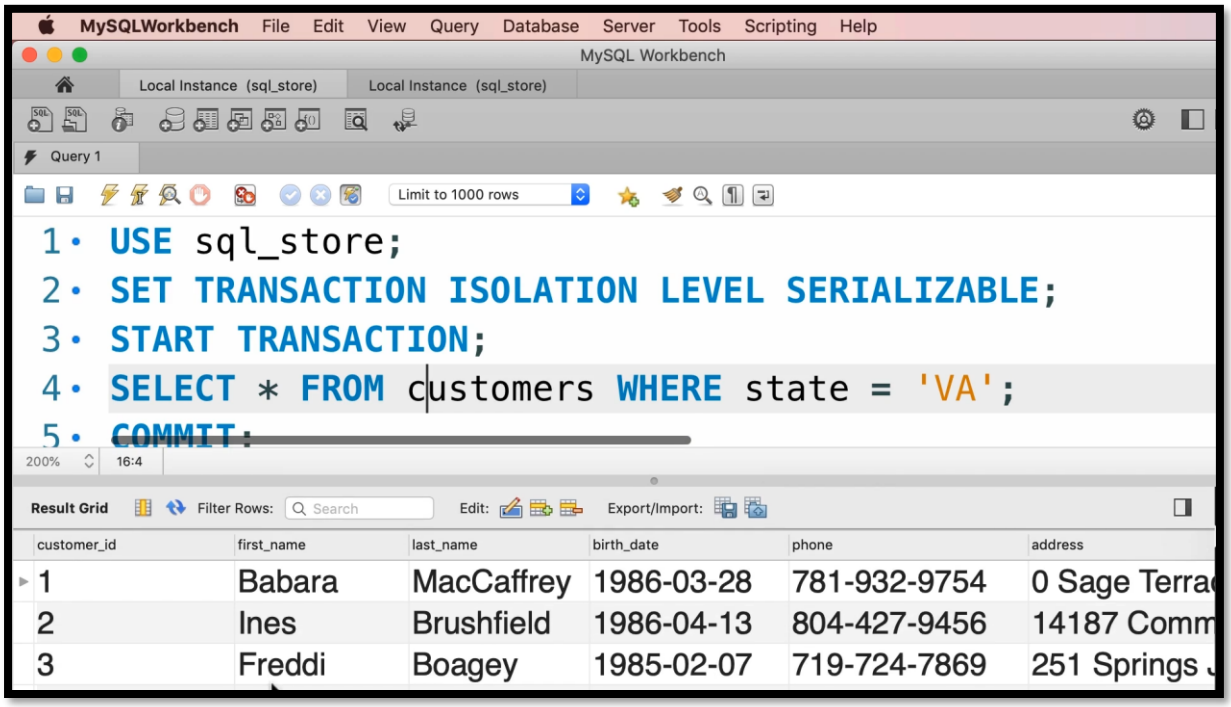
Another client is updating customer 3 while the first client is trying to read the customers who live in Viginia. So, customer 3 must be included in our query otherwise we will have phantom reads.



Let us run line 4 and look at the spinner. This transaction is waiting for transaction in session 2 to finish.



Let us do a commit.



Now we will have three customers who live in Virginia.

In this isolation level, all concurrency problems get resolved.