COMP90050: Advanced Database Systems
Semester 2, 2024
Lab 2a: Locking

Databases contain data that is interacted with by multiple processes/systems/users at the same time. There could be multiple queries executing on the same data row even. When these scenarios come up, they require mechanisms to ensure a safe and consistent way of allowing access and performing updates on the shared data. One of the ways to ensure this safe access/update is by using "Locking".

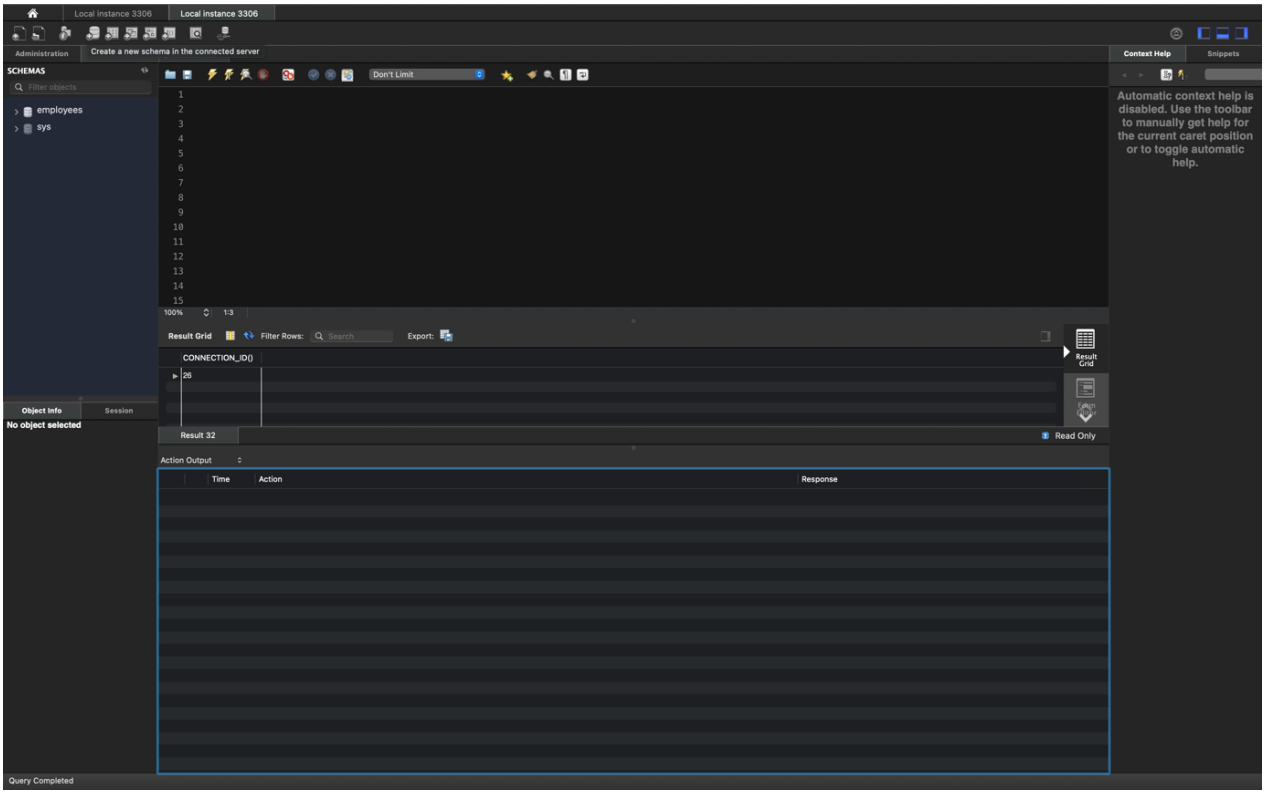
MySQL supports different types of Locking mechanisms such as S (Shared Lock), X (Exclusive Lock), I (Intention Locks), etc. We will cover S and X locks as part of this document with I locks being covered in a separate lab.

Note: As part of this lab, we will be setting Locks explicitly, which is not the norm in databases. Due to the fast nature of query execution, it is hard to understand what locking does and guarantee, whether two transactions just ran at different times or whether locking was in effect. So, to catch events as they unfold, we will use locks explicitly to showcase the side effects of different types of locks more easily.

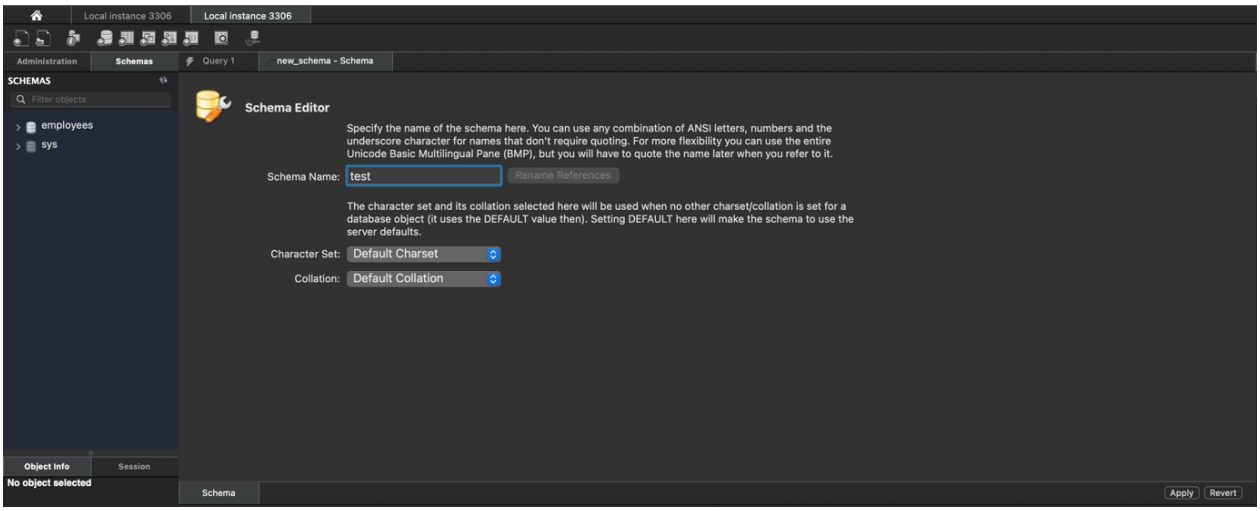
we will first create a sample database of our own with a few rows of data as follows:

Create the Schema:

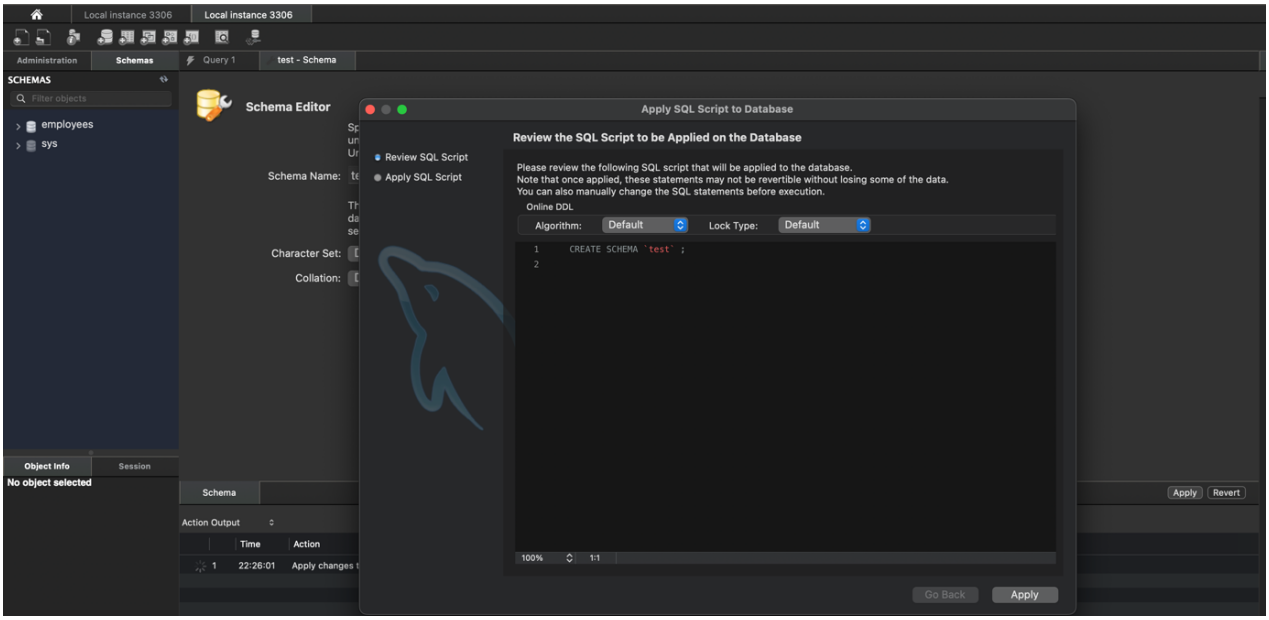
We first create a new schema to host our data objects using the built-in functionality in MySQL Workbench as seen in the image below. Schemas can also be created using a SQL command (CREATE SCHEMA <schema-name>). For the purposes of this document, we use the in-built functionality in Workbench. Click on the icon with the DB symbol, which presents a description of "Create a new schema…" when you hover your mouse over it.



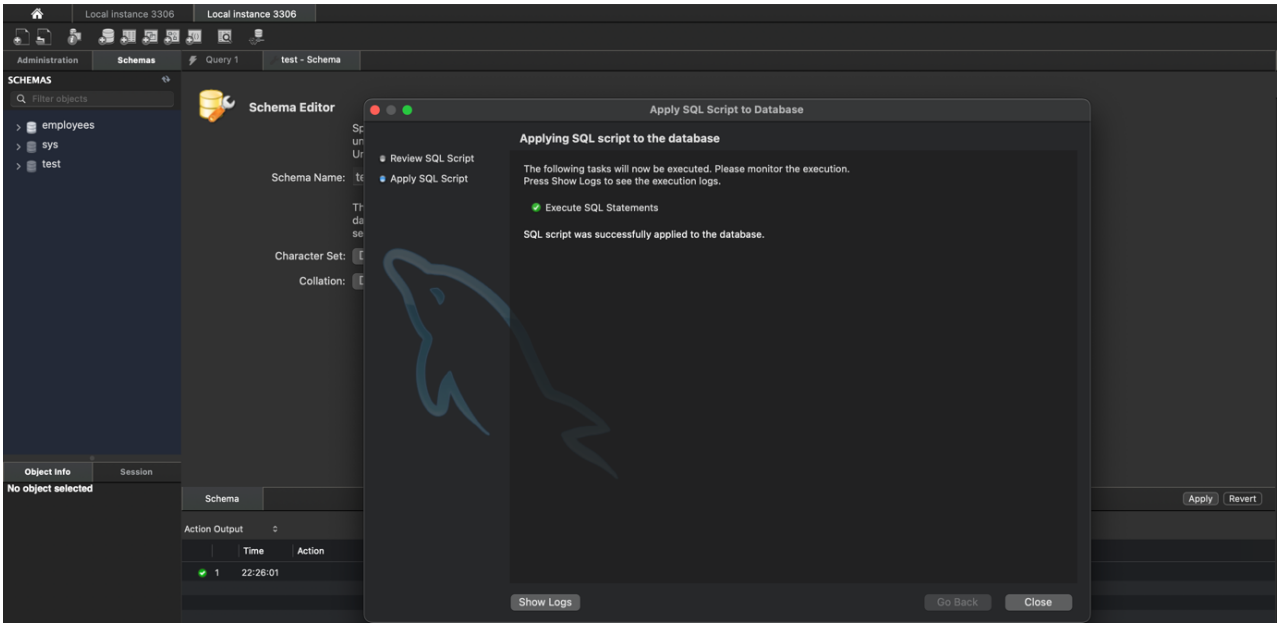
The following tab will pop-up, asking the name of the schema as well as any properties that require setting. In our case, we only name the schema and let the other settings remain as their default value. In our case, we name our schema as “test” and then click on the “apply” button at the bottom of the pop-up.



By clicking “apply”, workbench will generate the SQL query for creating the schema, which we can alternatively run ourselves instead of going through the self-guided prompt in workbench. Review the command and then click on “apply”.



After clicking on “apply”, we get a confirmation pop-up showcasing that the schema has been successfully created.

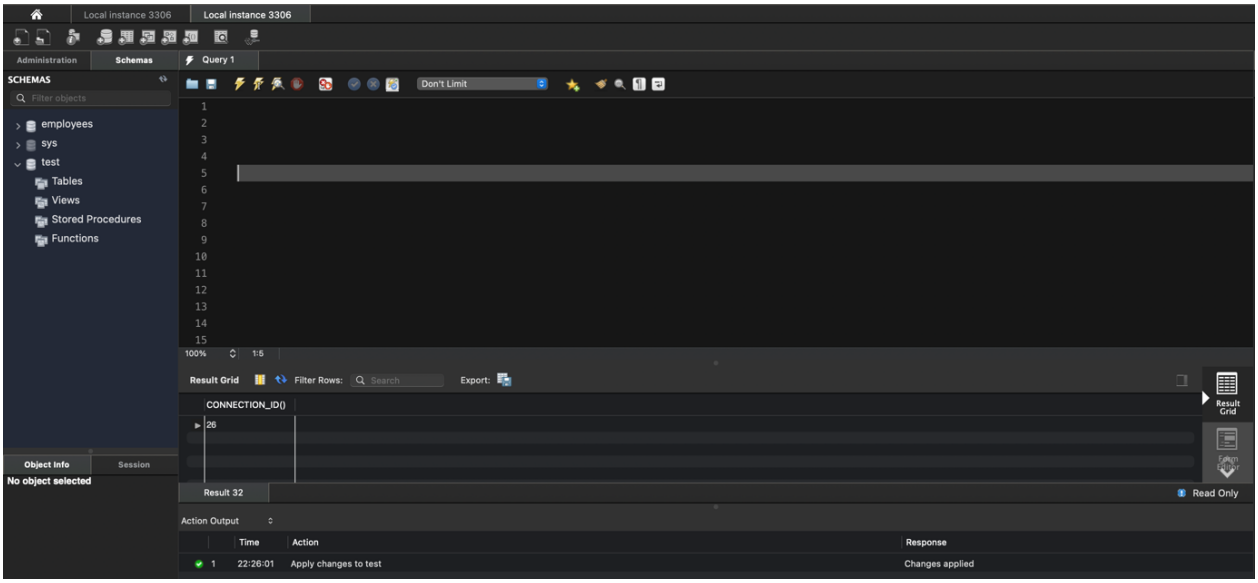


You can click on “close” and observe that a schema called “test” is present on the left side bar under the “Schemas” tab window in workbench. Close the “test – Schema” tab in workbench by clicking the “X” button on the side of the tab window.

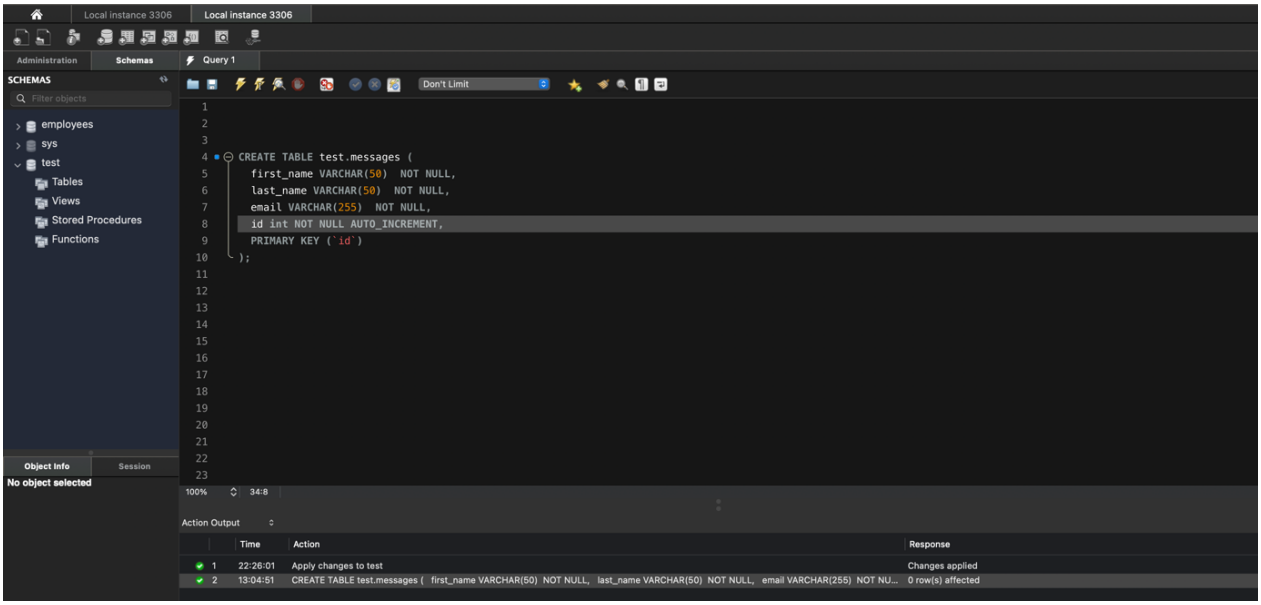
When we expand the “test” schema on the left side of the pane, we can observe that currently it has no tables nor data attributes. As part of this exercise, we will create a table next and populate it with some data.

Create a Table:

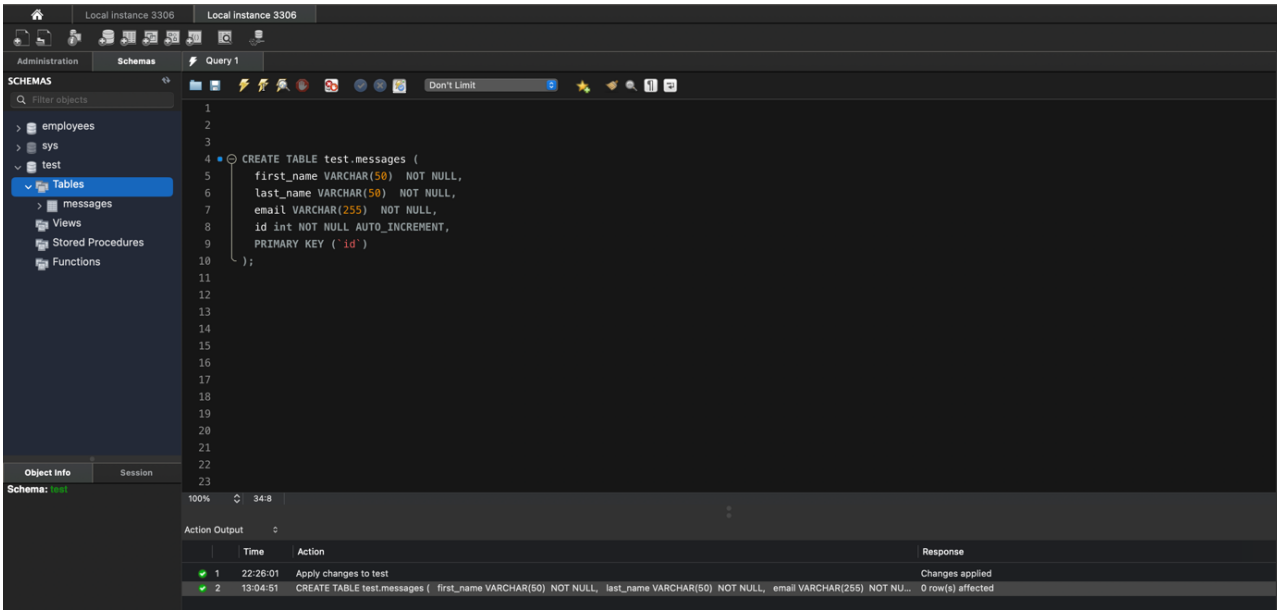
If we expand the “test” schema, we can note that there are currently no tables associated with it as seen in the image below.



We are going to create a table called messages as part of this exercise. You can create a table by running the CREATE TABLE SQL command as follows:



If you cannot view the table on the left panel of the Schemas tab then you can right click on the “Tables” entry and click on “Refresh All”. This would refresh your workbench view and you can observe the new table called “messages” there.



Insert data into the Table:

We can insert data into the table by using the INSERT SQL command as follows:

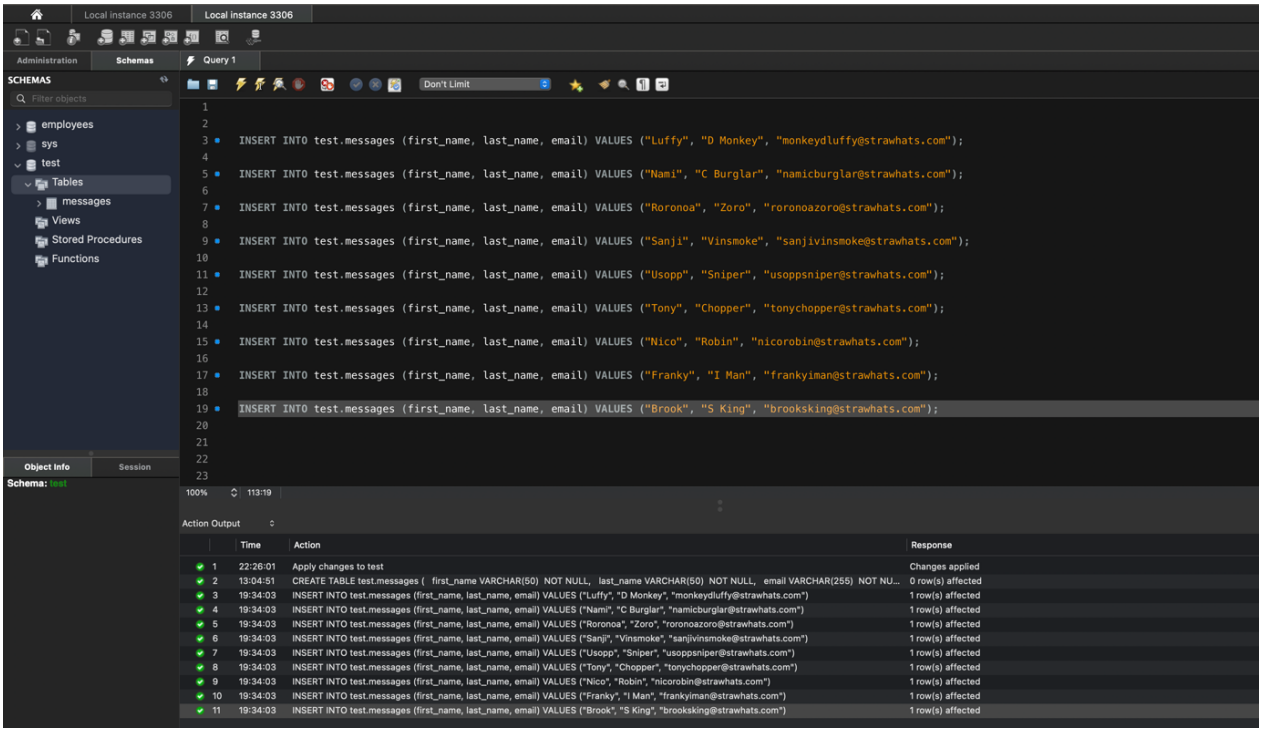
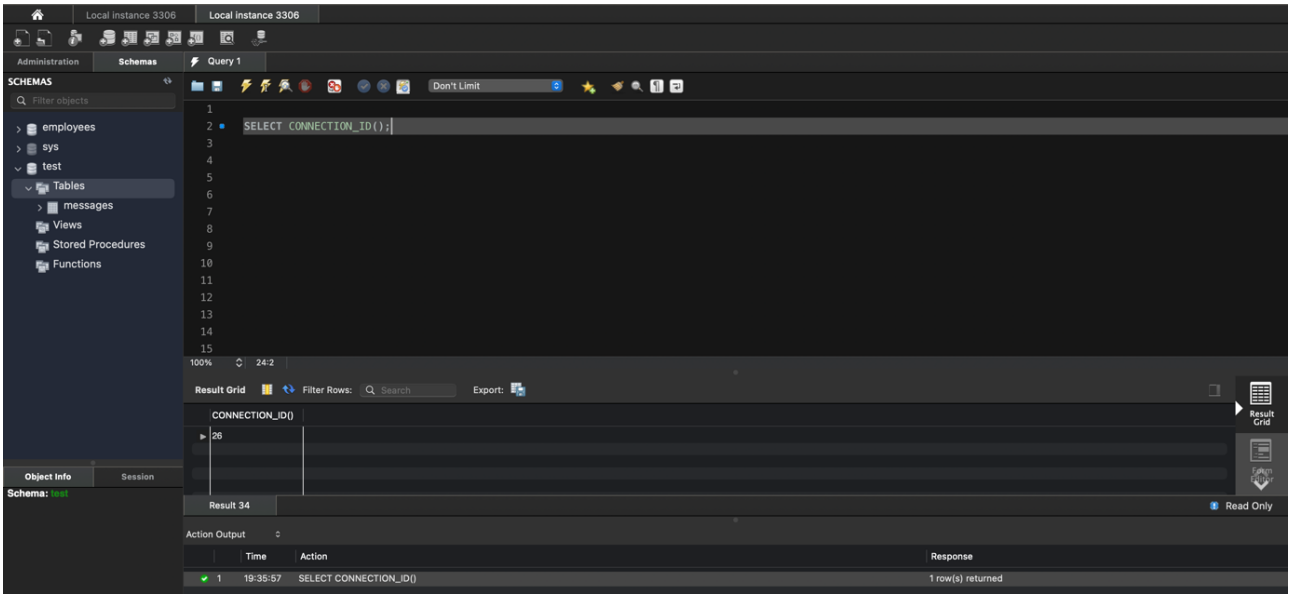


Table Level READ Lock:

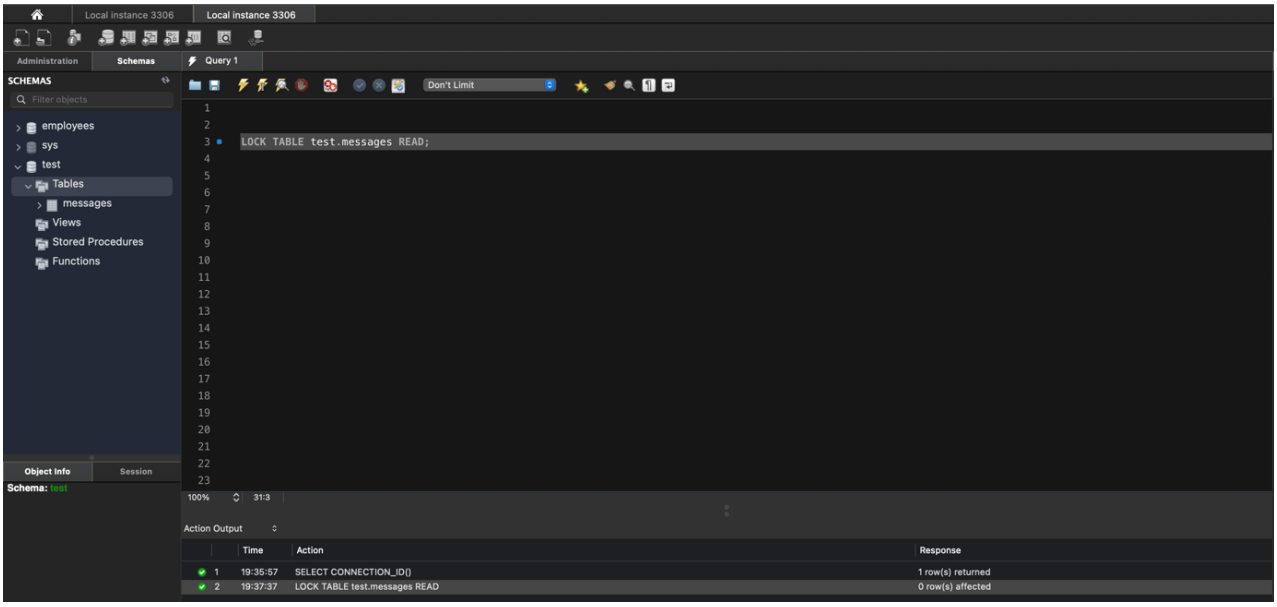
This mode allows queries to read data only without allowing write operations.

We will first view our current unique connection Id for the session to the MySQL database by using the following command:

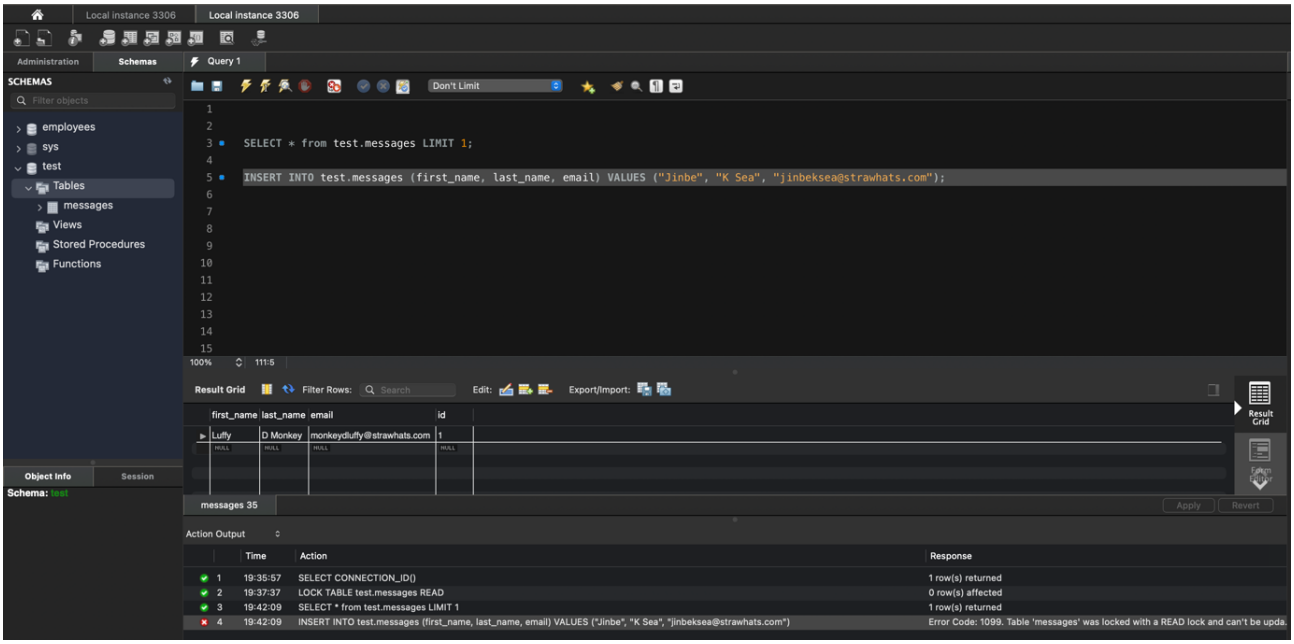


We can view that the current connection Id for our session is: 26

Next, we are going to lock the table in READ mode using the LOCK TABLE SQL command as follows:

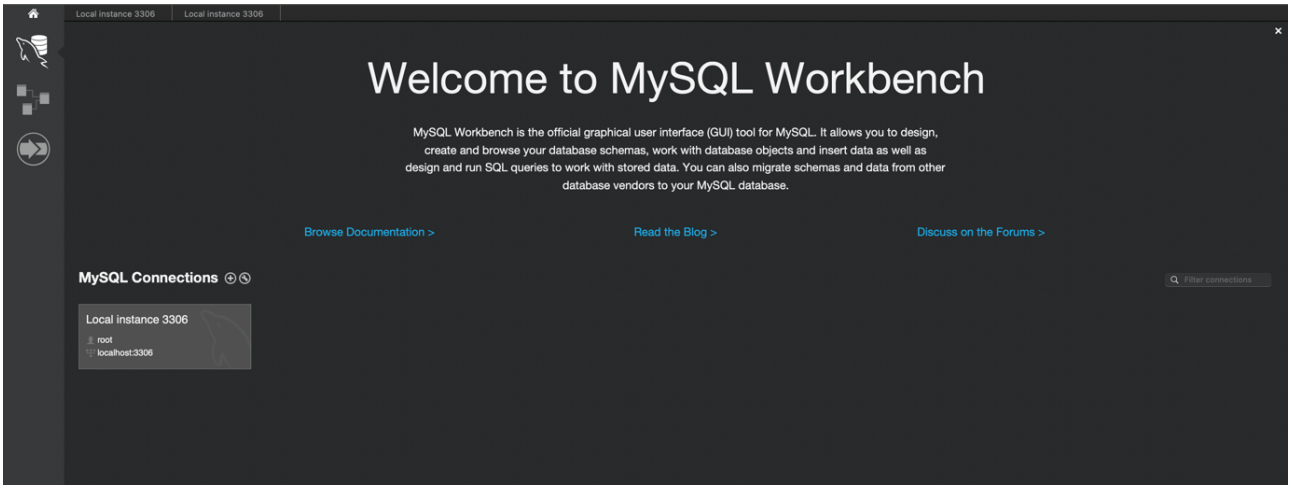


This locks the “messages” table in READ mode, enabling us to only read data from this table while stopping all write commands that we try to execute. We can verify this by running a SELECT command to read 1 row from the “messages” table and try to write another row into this table.

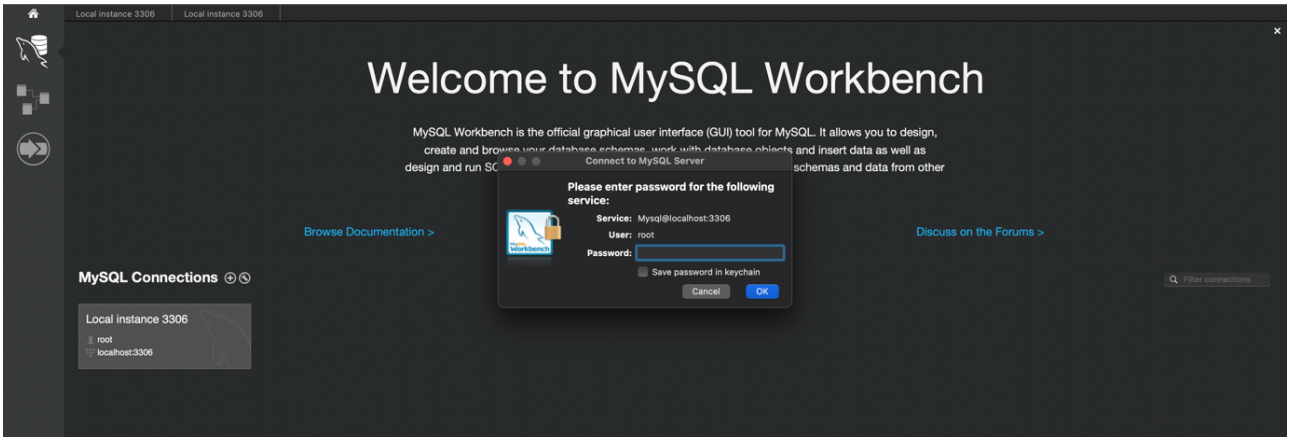


We can observe that the SELECT command returned the 1st row in the "messages" table. However, the INSERT command failed to run as the table was locked with a READ lock. Since the READ lock was acquired in this session, MySQL will prevent us from writing to this table. Since this table has a read lock on it, multiple users can read data from it but will be prevented to write data to it. This can be verified by opening a new session.

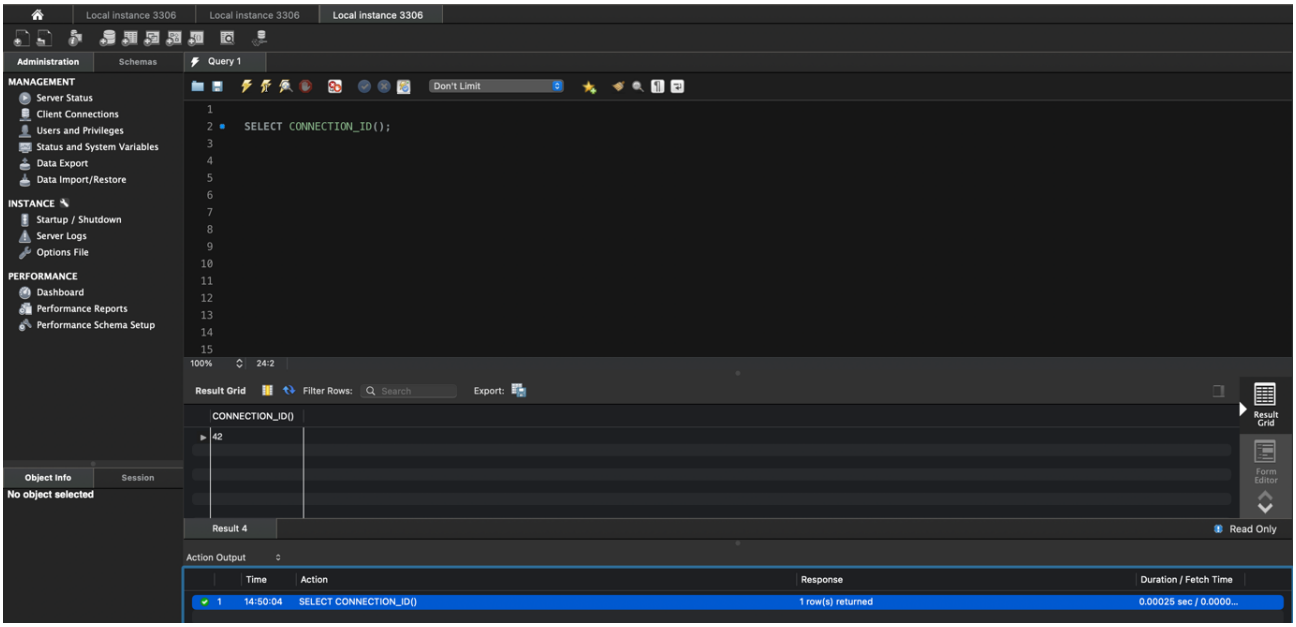
To open a new session, click on the home button on the top left corner of the UI. You will be led to the Workbench home page with the database connection listed as follows:



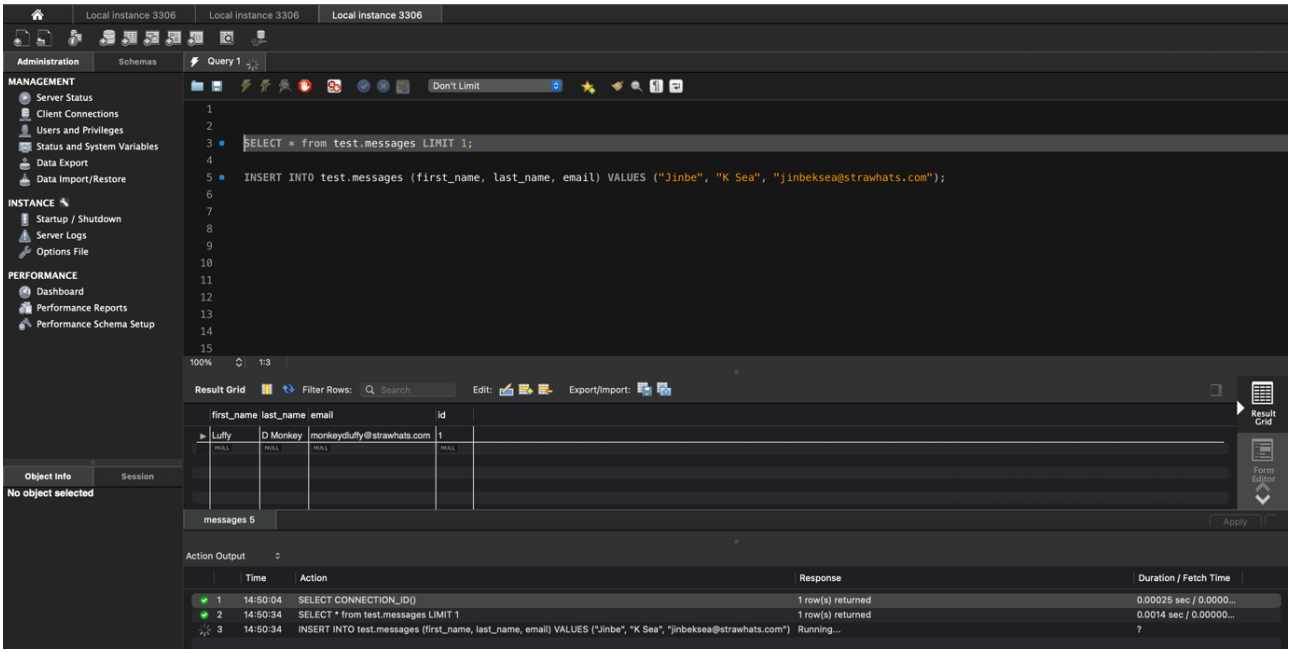
Click on the MySQL Connection and enter your password for accessing the database.



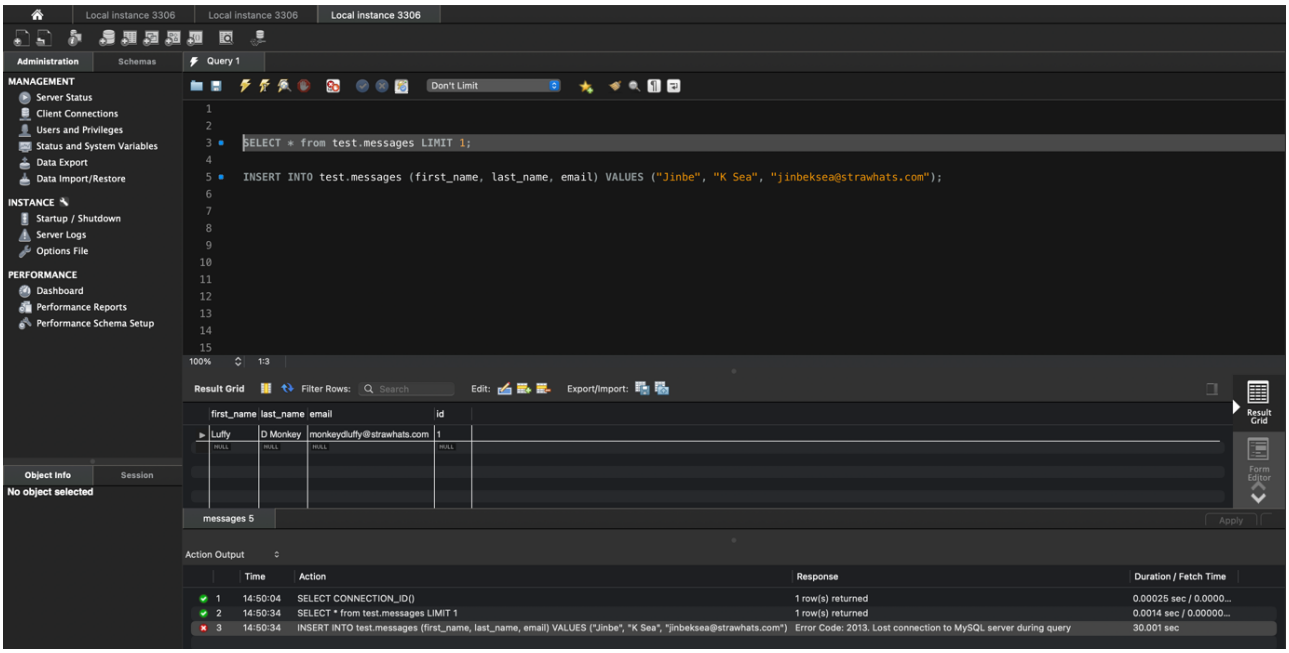
A new tab will open. You can verify that this is a new session by running the SELECT CONNECTION_ID(); SQL query as follows:



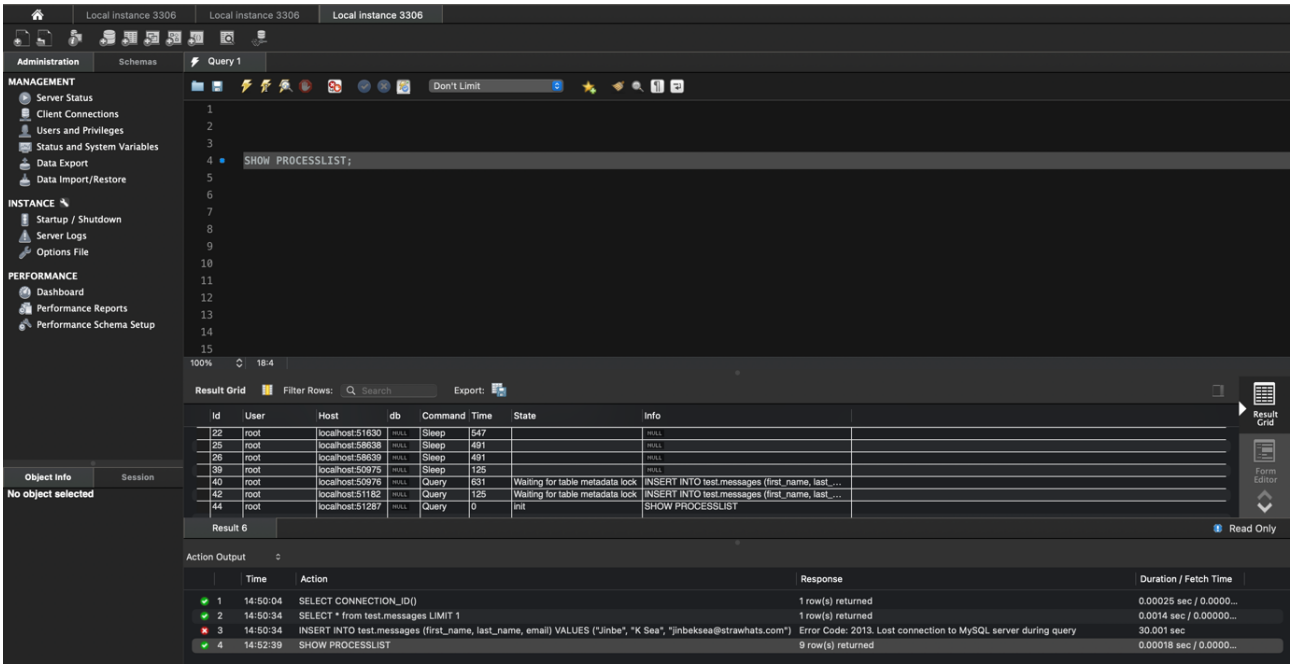
We can observe that the connection ID for this session is 42. Now, we can try the same read and write operation on the “messages” table to observe whether this new session would be allowed to perform these operations.



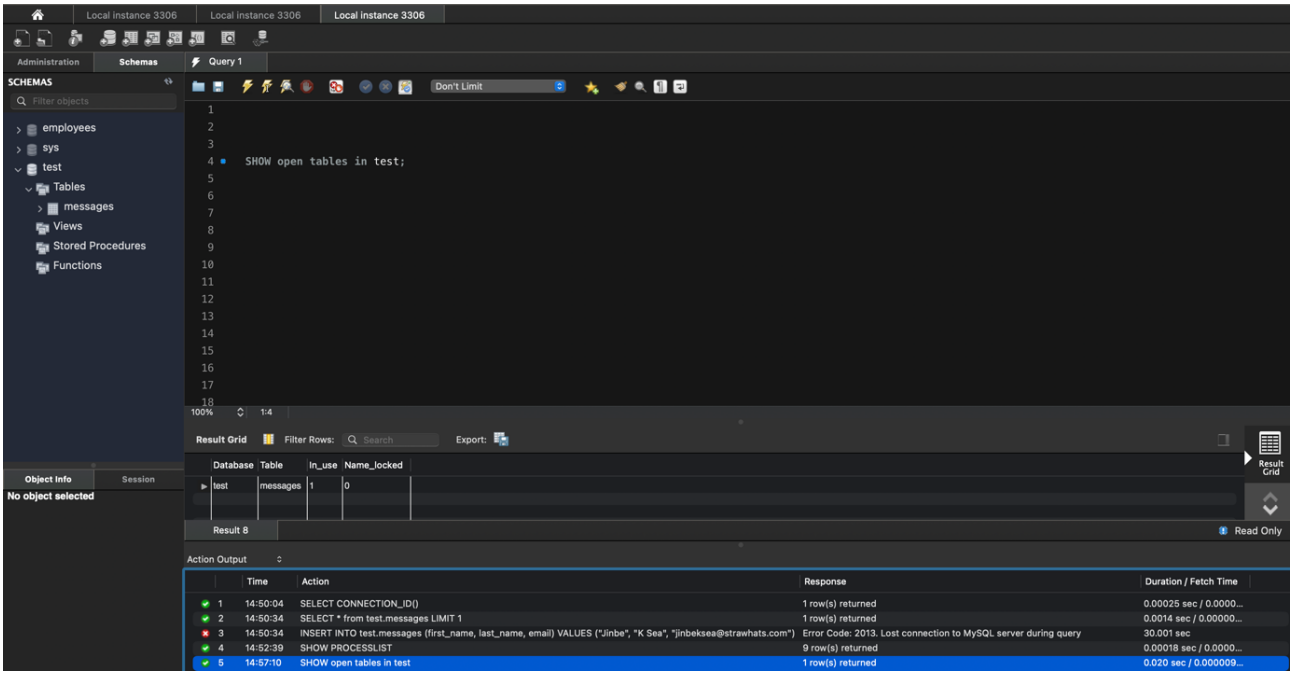
We can observe from the Action Output logs at the bottom that the read query (SELECT) worked but the write query (INSERT) is processing. This will continue till the connection eventually times out, indicating an error.



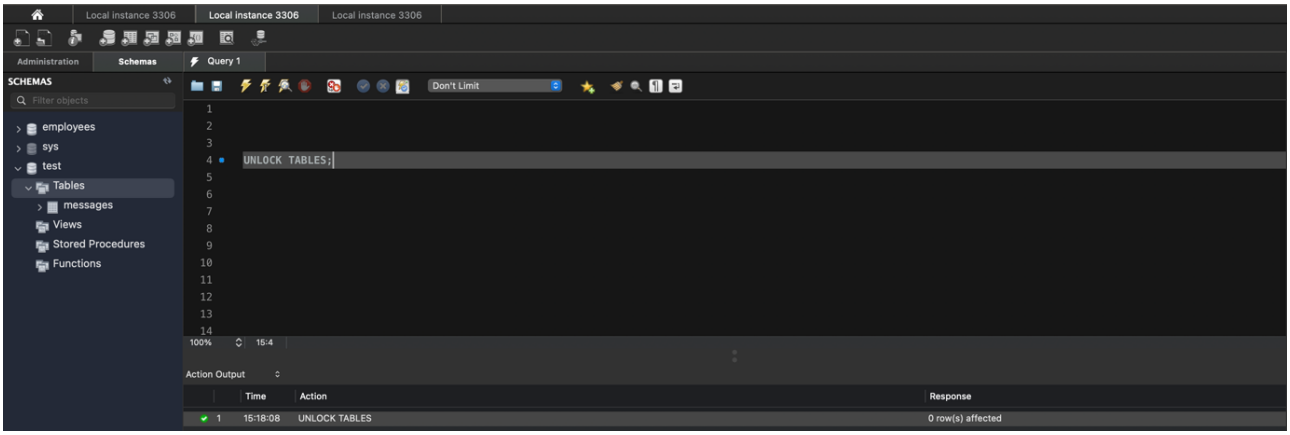
The error being displayed is stating that the connection to the MySQL server was lost (you will be asked to login again – meaning, a new session) but is not meaningful enough to understand what caused this issue. Often enough, we will come across scenarios where the MySQL/DB logs are not descriptive enough at first glance. To better understand the underlying issues, we will need to dig further into the issues by looking at the Database native process logs. One way to do so is by using the SHOW PROCESSLIST; SQL query as follows:



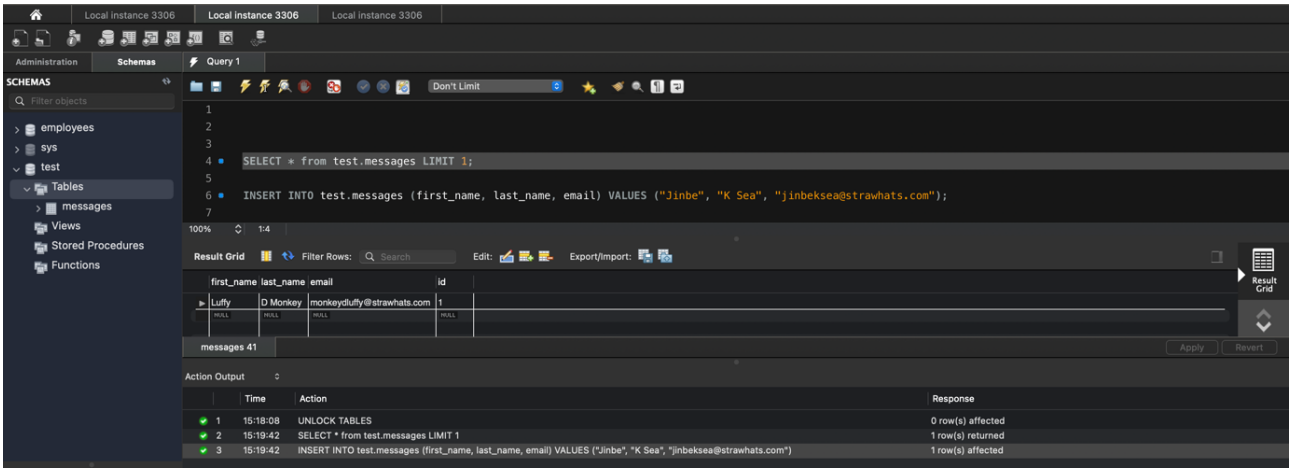
We can observe from the returned rows of the PROCESSLIST that the Session Id of 42 failed to insert data into the messages table as it was waiting for the table lock to be granted to it. This indicates us to check the current lock on the table, which can be identified by using the SHOW open tables command as follows:



Usually there are a few ways to deal with locking based issues. In our case, since one of the sessions is holding onto the lock for the “messages” table, we will have it release the lock as follows:



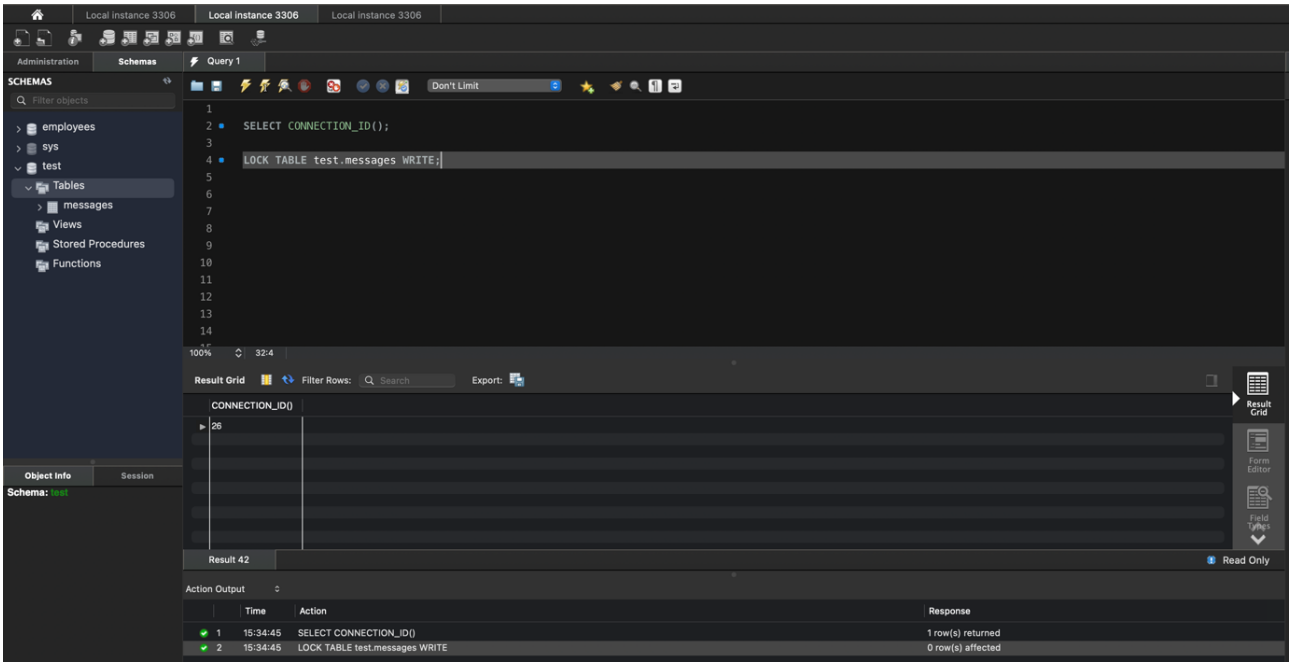
If we try the read and write operation now, we can observe that it works.



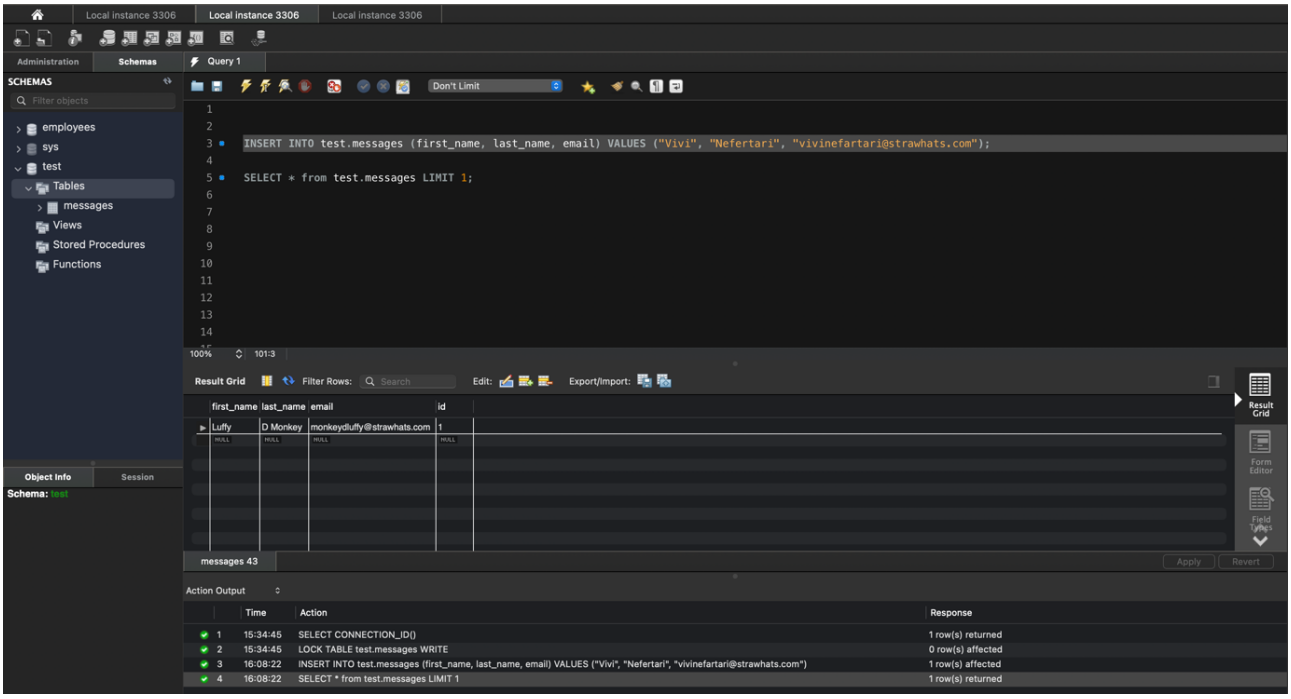
Task: If we have two queries that get a READ lock on a table and then perform a SELECT operation, will the SELECT queries run successfully?

Table Level WRITE Lock:

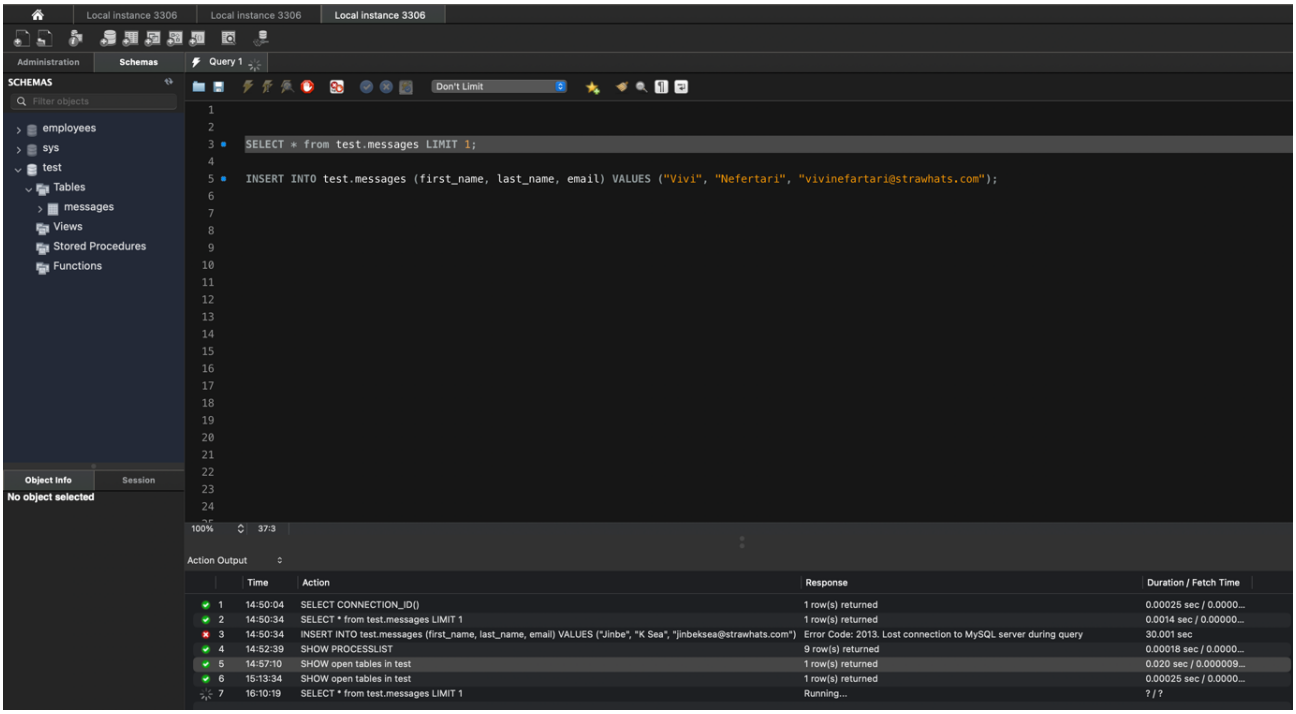
In this section, we will create a WRITE lock on the “messages” table and perform write operations, while testing write operations using another session simultaneously. To start – we first view the session ID as well as attain the WRITE lock on the “messages” table as follows:



We can observe that the session ID is 26 and it has attained the write lock on the “messages” table. Now, we can try inserting a row of data into this table and then reading it (using the SELECT command).



Both the write and read operations work in the current session which holds a WRITE lock on the “messages” table. If we try the same operations from **a new session**, we will observe that neither the read nor the write SQL commands would execute (they will potentially timeout).



We can observe the PROCESSLIST for these queries (from the new session) and we will observe that they are stuck waiting for the lock. If we go back to the original session which retrieved the lock on the “messages” table and release the lock there using the UNLOCK TABLES; SQL command, any subsequent transactions waiting for the lock on the “messages” table to be removed will finally start executing (if they haven’t timed out yet).

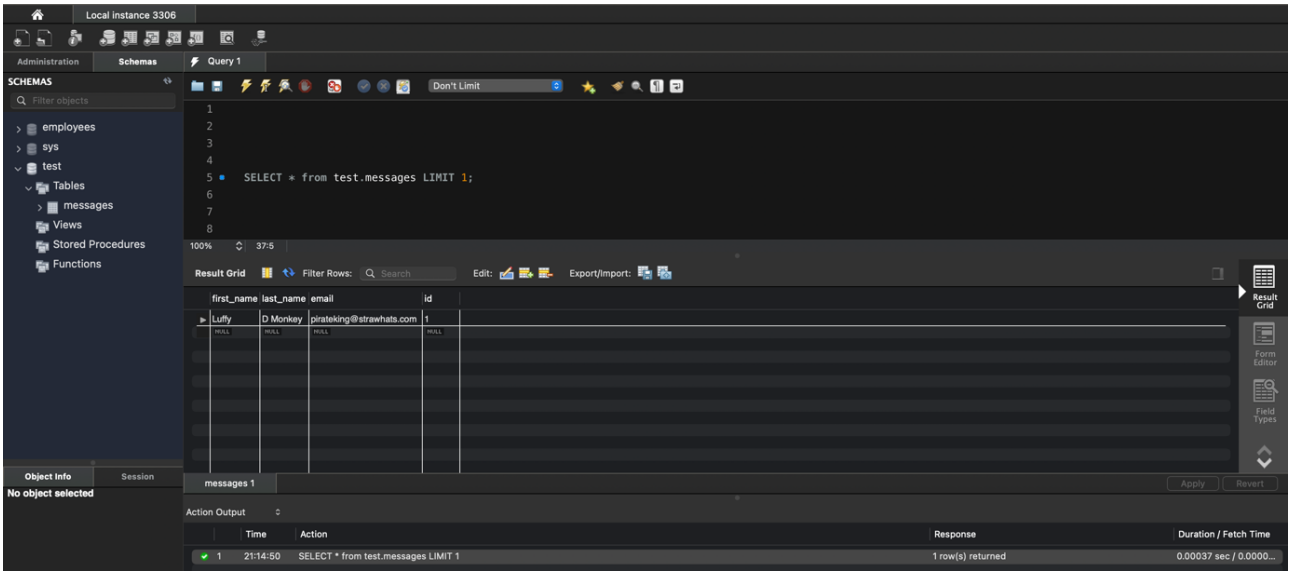
Isolation:

There are 4 isolation levels possible in MySQL, namely:

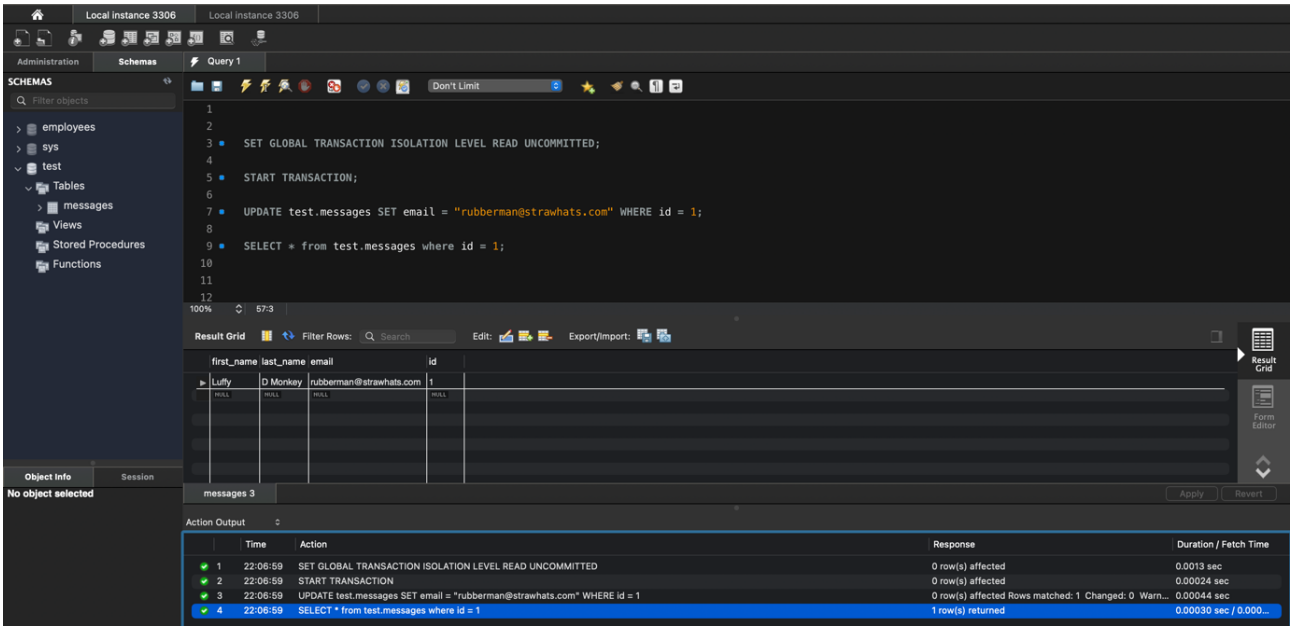
READ UNCOMMITTED:

This isolation level does not have locks present when multiple transactions are interacting with the same data. Due to this property, dirty reads are allowed meaning one transaction can read uncommitted changes of another transaction.

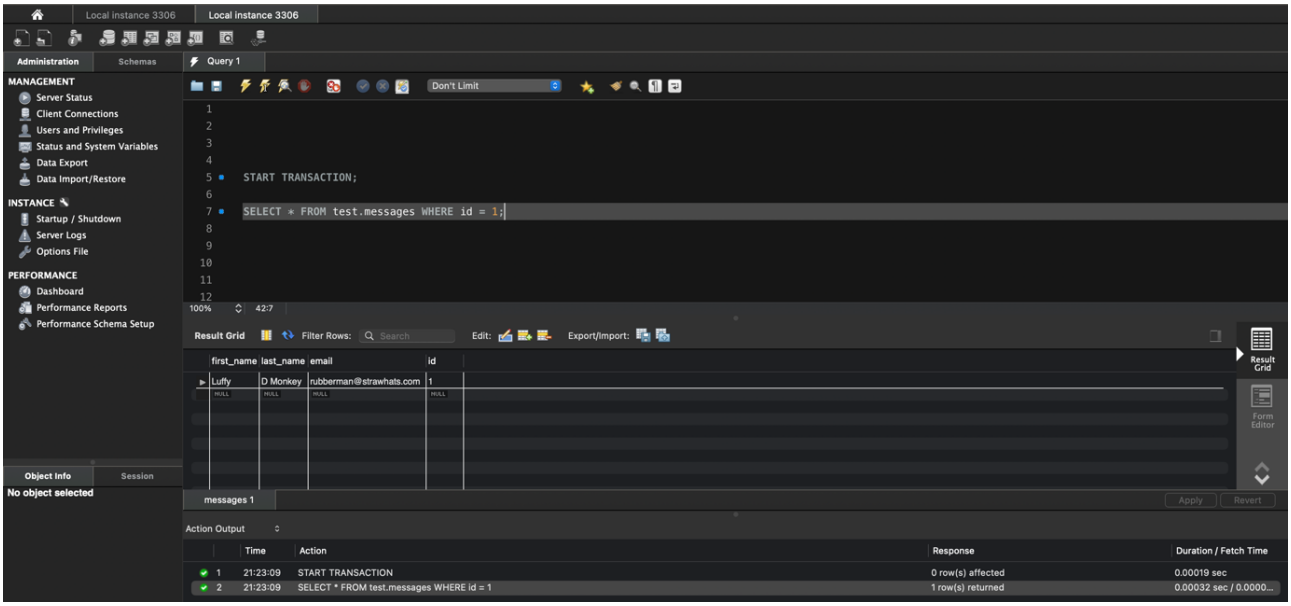
For instance, lets look at the email Id of row with Id = 1.



We will start a transaction and update the email Id of this row to a new value as follows:



Similarly, open a new session and start a transaction in that new session. Read the data for the row with id = 1, and you should observe the updates made by the first session (which still has not been committed/roll backed yet).

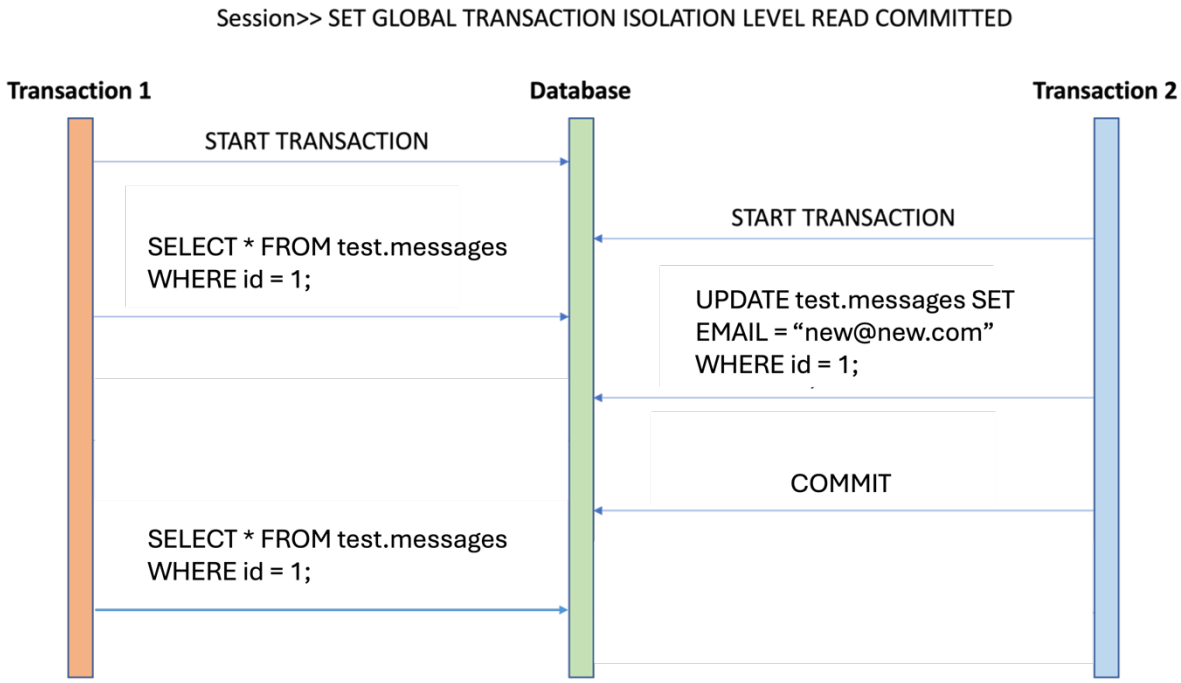


Now if the first session rolls back, following which the second session commits / continues processing then the second session would have read dirty data from the first session.

Similarly, there are three other levels, as given below. Try running through them yourself as per the sequence given in the diagrams in the following sections.

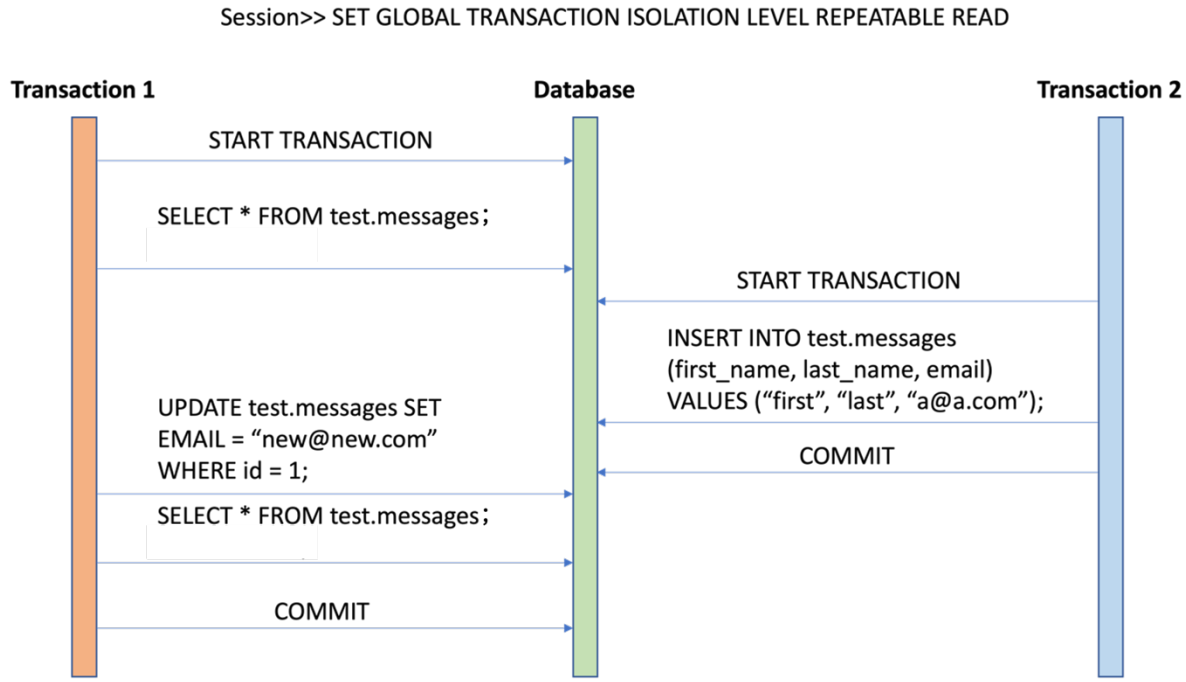
READ COMMITTED:

This isolation level avoids the premise of dirty reads but it does allow non-repeatable reads, where subsequent read commands could return different values. We can set this isolation level using the command: SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;



REPEATABLE READ:

This isolation level avoids non-repeatable reads and is the default isolation level for MySQL. However, it does allow phantom reads where one session is repeating a read operation on the same data but is returned new records. We can set this isolation level using the command: SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;



SERIALIZABLE:

This isolation level is the strictest form of isolation. This level enables transactions to run concurrently while creating an effect that transactions are running in serial order.

This can be enabled by using the following SQL command:

SET GLOBAL TRANSACTION ISOLATION LEVEL serializable;