INFO20003 Week 10 Lab

Transactions

The idea of transactions is to group several SQL statements into one atomic operation. What we mean by atomic is that all SQL DML statements must succeed, or all SQL DML statements must fail.

The atomicity of SQL transactions can be controlled by using commands. These commands can be typed directly into a MySQL query browser. **START TRANSACTION** is the command to start a transaction. **COMMIT** is the command that makes all changes permanent (all statements must succeed). If there is a problem or we wish to discard the changes made by the transaction, we can enter the keyword **ROLLBACK.** COMMIT and ROLLBACK only apply to DML changes when databases are configured not to run in autocommit mode.

The same transaction controls (START TRANSACTION, COMMIT and ROLLBACK) can be achieved in MySQL Workbench by using the transaction control buttons.

Objectives:

By the end of this lab, you should be able to:

1. Practice SQL transactions using the MySQL Workbench Query Browser
2. Run multiple MySQL Workbench connections to one database to demonstrate concurrency

Section 1: Set up MySQL Workbench

Turn off safe updates

By default, MySQL Workbench will only run UPDATE and DELETE queries that reference the primary key from the WHERE clause. This is called *safe updates*.

In this lab, we will be running DML statements that have conditions on columns other than the primary key. To allow this behaviour, safe updates must be turned off.

- **Task 1.1** Go to **Edit** > **Preferences** (**MySQL Workbench** > **Preferences** on macOS).

◆ **Task 1.2** Select **SQL Editor** from the left panel, then scroll to the bottom of the right panel.

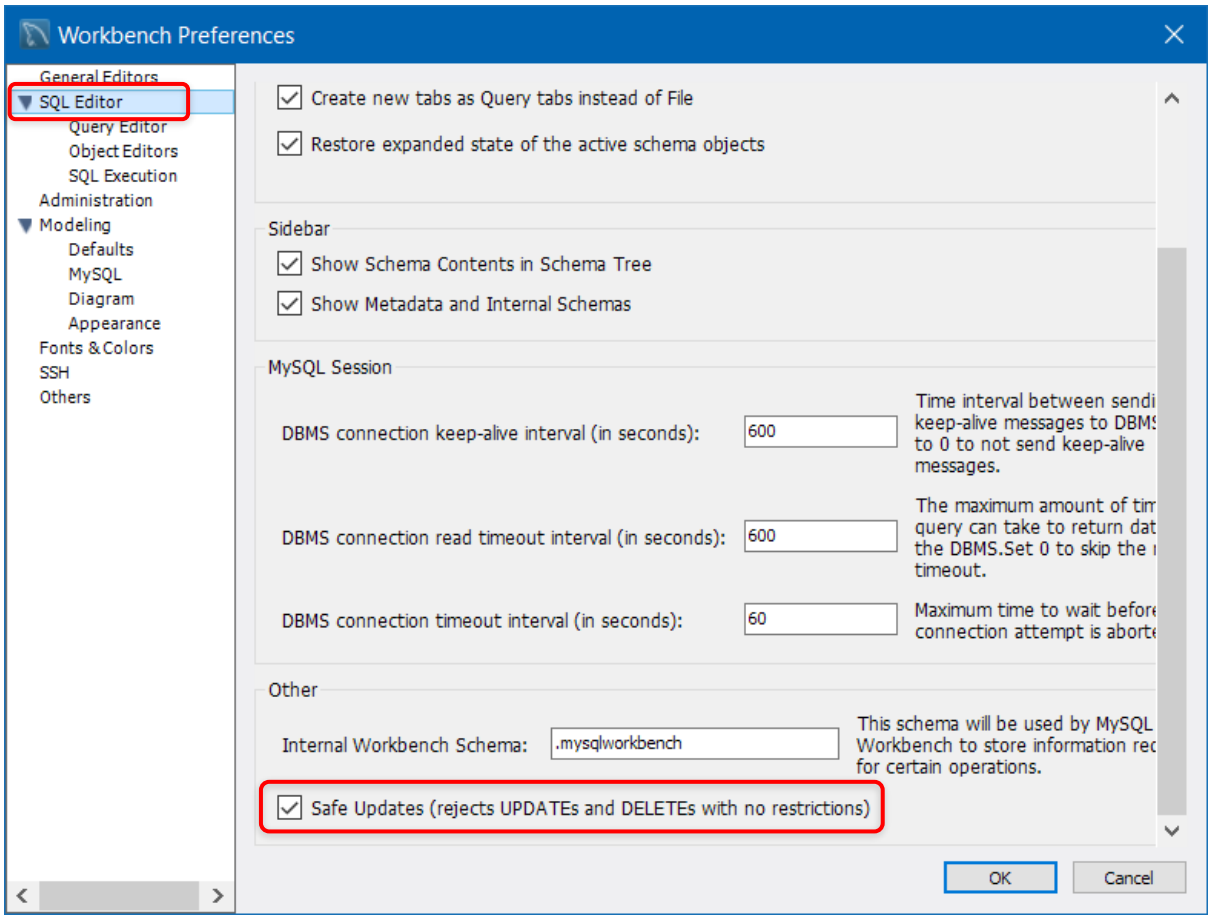

Figure 1: Turning off safe updates in the Preferences dialog

◆ **Task 1.3** Uncheck the **Safe Updates** checkbox (Figure 1) and click **OK**.

◆ **Task 1.4** Close and reopen MySQL Workbench.

Disable autocommit

The default behaviour of MySQL Workbench is to automatically commit each DML statement. To build an atomic transaction composed of more than one statement, we must switch autocommit off.

◆ **Task 1.5** Turn off autocommit by clicking the Autocommit button highlighted in Figure 2.

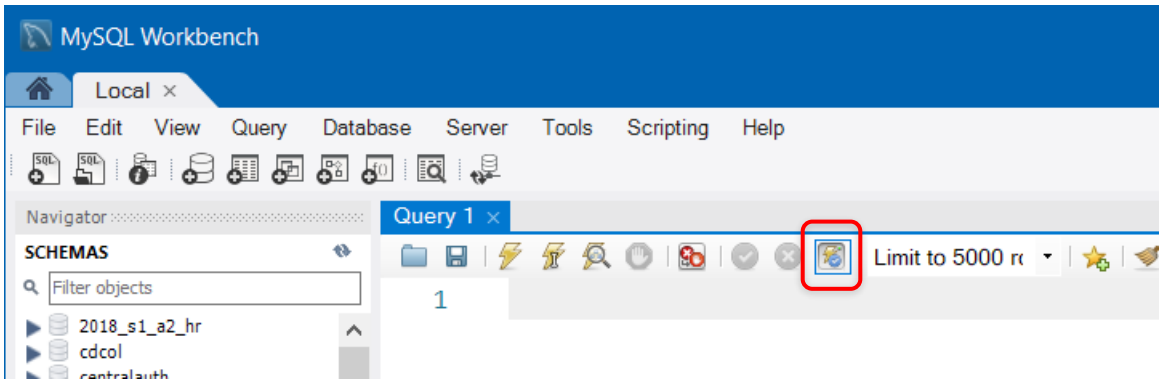

Figure 2: The autocommit control button in Workbench

When autocommit is **off**, the manual transaction control buttons – commit (tick) and rollback (cross) – become available:

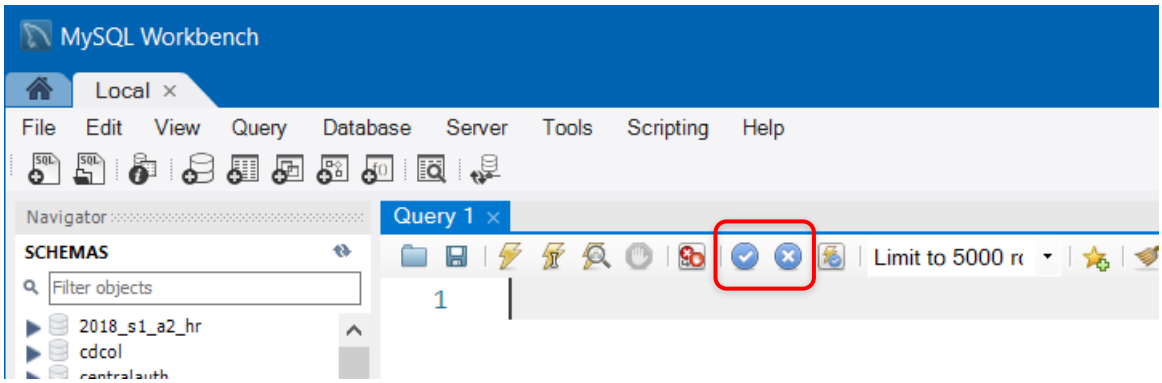

Figure 3: The transaction control buttons become available when autocommit is turned off.

Remember:

To COMMIT your current transaction (one or more DML statements), click the tick button.

To ROLLBACK your current transaction, click the cross button.

Install the schema

To continue with the lab, we will install two tables, *movie* and *soundtrack*, into our database schema.

◆ **Task 1.6** Open MySQL Workbench and connect to your preferred MySQL server.

◆ **Task 1.7** Download and run the *movie.sql* script from LMS.

Confirm that you have 175 rows in the *soundtrack* table and 20 rows in the *movie* table:

```
SELECT COUNT(*)
FROM soundtrack;

SELECT COUNT(*)
FROM movie;
```

Because we turned autocommit off, a transaction is currently in progress. The rows inserted by the script are not yet stored durably in the database. You need to **commit** the insert of the rows, either by clicking the tick button in the toolbar, or by executing the COMMIT statement:

```
COMMIT;
```

The transaction is now complete and the DML changes (inserted rows) are now permanent.

Section 2: Single-user transactions and rollback

In this section, we will start a transaction and experiment with durability and atomicity using the ROLLBACK command.

Your first transaction

◆ **Task 2.1** Confirm that autocommit is off and the manual transaction control buttons are active.

◆ **Task 2.2** Query the *movie* table for information about the movie 'Titanic':

```
SELECT *
FROM movie
WHERE name = 'Titanic';
```

	movie_id	name	genre	release_date	worldwide_gross	director_name	studio
▶	1	Titanic	Romance	1999	18353000.00	James Cameron	Paramount Pictures

◆ **Task 2.3** Write an UPDATE statement to change the genre from 'Romance' to 'Comedy':

```
UPDATE movie
SET genre = 'Comedy'
WHERE name = 'Titanic';
```

If this statement fails with Error Code 1175, you need to disable safe updates (see Section 1). After disabling safe updates, you must **reconnect** to the server.

◆ **Task 2.4** Re-run the SELECT statement to confirm that the genre has changed:

```
SELECT *
FROM movie
WHERE name = 'Titanic';
```

	movie_id	name	genre	release_date	worldwide_gross	director_name	studio
▶	1	Titanic	Comedy	1999	18353000.00	James Cameron	Paramount Pictures

◆ **Task 2.5** Roll back the transaction by using the cross button, or by typing:

```
ROLLBACK;
```

◆ **Task 2.6** Run the SELECT statement again:

```
SELECT *
FROM movie
WHERE name = 'Titanic';
```

The change you made to Titanic's genre has been rolled back to its original value, 'Romance'. The DML statement changes have been undone.

***Note**: In MySQL Workbench, when autocommit is off, each time you start a new session to the database or you type COMMIT or ROLLBACK, Workbench implicitly issues a BEGIN TRANSACTION statement. You will never need to type this in a MySQL Workbench session.*

Durability: Making changes permanent

◆ **Task 2.7** Study Transaction 1:

```
-- TRANSACTION 1

-- Statement 1
SELECT *
FROM movie
WHERE name = 'Titanic';

-- Statement 2
UPDATE movie
SET genre = 'Comedy'
WHERE name = 'Titanic';

-- Make all changes permanent
COMMIT;

-- The explicit COMMIT ends transaction 1
```

◆ **Task 2.8** Run Transaction 1 in MySQL Workbench, then confirm that the genre of 'Titanic' is now 'Comedy'.

◆ **Task 2.9** Study, then run, Transaction 2:

```
-- TRANSACTION 2
UPDATE movie
SET name = 'James Cameron''s Titanic'
WHERE name = 'Titanic';

-- Confirm that Titanic is a comedy film in the Movie table,
-- and the Title has been changed
SELECT name, genre
FROM movie
WHERE name LIKE '%Titanic';

-- Undo the title change
ROLLBACK;
-- Rollback caused the end of transaction 2
```

```
-- Begin a new transaction - Transaction 3
SELECT name, genre
FROM movie
WHERE name LIKE '%Titanic';
```

The genre has been permanently changed to 'Comedy', but the title changes have been undone. Transaction 1 was committed and made permanent. Transaction 2 was rolled back and the changes were undone. The third transaction displays all committed data visible to all users.

As a single user using the database, rollback is the most useful feature of transactions, because it enables you to undo changes you make if you make an error in a DML statement.

◆ **Task 2.10** Continuing Transaction 3, enter the following commands one at a time. Each DML statement is followed by a SELECT statement to confirm that changes have been made.

```
SELECT *
FROM movie
WHERE name = 'Titanic';

-- Change Titanic's genre from Comedy to Melodrama
UPDATE movie
SET genre = 'Melodrama'
WHERE name = 'Titanic';

SELECT *
FROM movie
WHERE name = 'Titanic';

-- Insert a new movie into the table
INSERT INTO movie
VALUES (DEFAULT, 'Baby Driver', 'Action', 2017, 1033463.00,
  'Edgar Wright', 'Released');

SELECT *
FROM movie
WHERE name = 'Baby Driver';

-- Delete 'Spider-Man' from the table
DELETE FROM movie
WHERE movie_id = 18;

SELECT *
FROM movie
WHERE movie_id = 18;
```

◆ **Task 2.11** Rollback the changes (thereby ending the transaction) and confirm that all the DML changes have been undone:

```
ROLLBACK;
```

```
SELECT *
FROM movie
WHERE name = 'Titanic';
-- Still listed as a comedy

SELECT *
FROM movie
WHERE name = 'Baby Driver';
-- No rows returned

SELECT *
FROM movie
WHERE movie_id = 18;
-- movie_id 18 is still in the Movie table
```

All the changes have been undone.

NOTE: *This is a simple but important point. Rollback doesn't undo the last DML statement that was executed; rather, it undoes **all** DML statements that have been executed in the current transaction.*

Section 3: Multi-user transactions and locking

Transactions allow more than one user to safely access the same data. To see this working, you will need to connect into your database twice using two different instances of MySQL Workbench.

The default behaviour of MySQL Workbench is to run one copy on a computer at a time. To test transactions and concurrency, we must change the default behaviour of MySQL Workbench.

- **Task 3.1** Go to **Edit** > **Preferences** (**MySQL Workbench** > **Preferences** on macOS).
- **Task 3.2** Select **Others** from the left panel.
- **Task 3.3** Check the **Allow more than one instance of MySQL Workbench to run** checkbox (Figure 4, on next page) and click **OK**.

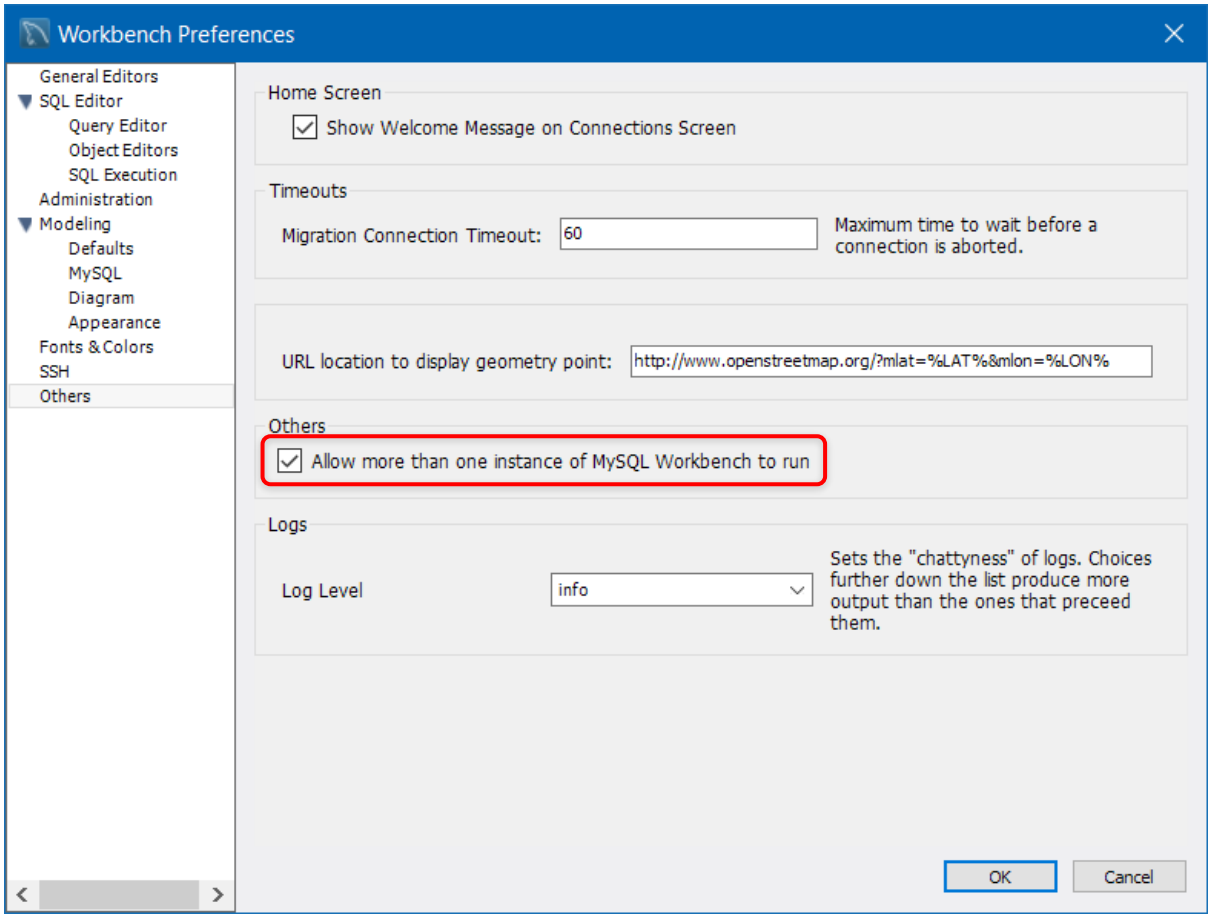

Figure 4: Allowing more than one copy of Workbench to be open simultaneously

◆ **Task 3.4** Close Workbench, then open it twice so that you have two Workbench windows.

For clarity, let's call these Window 1 (for User 1) and Window 2 (for User 2).

Isolation: Keeping transactions separate

◆ **Task 3.5** Connect to your database in both windows.

◆ **Task 3.6** In **Window 1**, turn off autocommit and start a transaction by running some DML:

```
UPDATE movie
SET director_name = 'Alan Smithee'
WHERE movie_id = 19;
```

◆ **Task 3.7** In **Window 1**, run a SELECT statement to see the changed data:

```
SELECT *
FROM movie
WHERE movie_id = 19;
```

	movie_id	name	genre	release_date	worldwide_gross	director_name	studio
▶	19	Star Wars	Action	2004	7979000.00	Alan Smithee	20th Century Fox

◆ **Task 3.8** In **Window 2**, do not turn off autocommit yet. Run the same SELECT statement:

```
SELECT *
FROM movie
WHERE movie_id = 19;
```

	movie_id	name	genre	release_date	worldwide_gross	director_name	studio
▶	19	Star Wars	Action	2004	7979000.00	George Lucas	20th Century Fox

Note that Window 2 (User 2) does not see the changes that User 1 has made in Window 1.

This illustrates one of the ways in which transactions control multi-user access (concurrency) to the same data. Uncommitted changes are only seen by the user making the changes.

◆ **Task 3.9** In **Window 1**, COMMIT the changes to make them permanent.

```
COMMIT;
```

◆ **Task 3.10** Run the SELECT statement in both windows:

```
SELECT *
FROM movie
WHERE movie_id = 19;
```

```
SELECT *
FROM movie
WHERE movie_id = 19;
```

Both users now see the changed data.

Locking: Enforcing the ACID principles

◆ **Task 3.11** In **Window 1**, run this DML statement to change the worldwide gross of one of the movies:

```
UPDATE movie
SET worldwide_gross = 10000000
WHERE name = 'Shrek 2';
```

◆ **Task 3.12** Display the worldwide gross of ‘Shrek 2’ in each window:

```
SELECT *
FROM movie
WHERE name = 'Shrek 2';
```

```
SELECT *
FROM movie
WHERE name = 'Shrek 2';
```

User 1 sees their own change (Window 1). User 1 is working inside an uncommitted transaction, so User 2 does not see any change (Window 2).

◆ **Task 3.13** In **Window 2**, try to change the same row:

```
UPDATE movie
SET worldwide_gross = 5000
WHERE name = 'Shrek 2';
```

User 2 waits for a response from the database. When User 1 changed this row, MySQL gave User 1 an exclusive lock on the row. User 2 must wait until User 1 has released the lock (by performing a COMMIT or ROLLBACK) before User 2's UPDATE statement can proceed.

◆ **Task 3.14** Wait.

After about 60 seconds, User 2's query will time out. The UPDATE has been abandoned so that User 2 is not delayed indefinitely.

◆ **Task 3.15** Roll back User 1's transaction in **Window 1**:

```
ROLLBACK;
```

Locking and ROLLBACK

Let's do the same thing again, but this time, we will not wait for the query to time out.

◆ **Task 3.16** In **Window 1**, change the worldwide gross of 'Shrek 2' again:

```
UPDATE movie
SET worldwide_gross = 10000000
WHERE name = 'Shrek 2';
```

◆ **Task 3.17** Turn off autocommit mode in **Window 2**.

A transaction will begin when User 2 executes a statement.

◆ **Task 3.18** In **Window 2**, try to change the same row:

```
UPDATE movie
SET worldwide_gross = 5000
WHERE name = 'Shrek 2';
```

◆ **Task 3.19** While the UPDATE query in Window 2 is pending, roll back **Window 1**'s transaction:

```
ROLLBACK;
```

User 1's update is undone, and User 2's update is immediately executed.

◆ **Task 3.20** In **Window 2**, confirm that the gross of 'Shrek 2' is now set to 5000:

```
SELECT *
FROM movie
WHERE name = 'Shrek 2';
```

◆ **Task 3.21** In **Window 2**, roll back the transaction:

```
ROLLBACK;
```

Locking and COMMIT

What if we try once more, this time committing our changes in Window 1?

◆ **Task 3.22** In **Window 1**, change the worldwide gross of 'Shrek 2' again:

```
UPDATE movie
SET worldwide_gross = 10000000
WHERE name = 'Shrek 2';
```

◆ **Task 3.23** In **Window 2**, try to change the same row:

```
UPDATE movie
SET worldwide_gross = 5000
WHERE name = 'Shrek 2';
```

◆ **Task 3.24** While the UPDATE query in Window 2 is pending, commit **Window 1**'s transaction:

```
COMMIT;
```

User 1's update takes place. Then User 1's lock is released, meaning that User 2's update is performed immediately afterwards.

◆ **Task 3.25** In **Window 2**, confirm that the gross of 'Shrek 2' is now set to 5000:

```
SELECT *
FROM movie
WHERE name = 'Shrek 2';
```

◆ **Task 3.26** In **Window 2**, roll back the transaction:

```
ROLLBACK;
```

◆ **Task 3.27** In **Window 2**, confirm that the gross of 'Shrek 2' has returned to the value from Window 1:

```
SELECT *
FROM movie
WHERE name = 'Shrek 2';
```

End of Week 11 Lab