Transactions Lab I

Tom Kelliher, CS 417

The purpose of this lab is for you to gain some understanding of how transactions work, see for yourselves how the various SQL isolation levels correspond to the ACID properties described in the textbook, and experiment with concurrent transactions to observe how they interact. "Pair-up" for this lab, using your project team as the "pair."

This is a graded lab. There are several boxed questions throughout the lab. Record your answers to these questions, using LibreOffice Writer or LaTeXon phoenix, or a similar piece of software. Number your answers using the subsection numbers shown in each box. Turn in hard copy of one answers document per pair. Record the names of the members of the pair at the beginning of the answers document.

1 Preliminaries

Complete this section individually.

1. Download `accounts.sql` and movies.sql from the course web site to one of your directories on phoenix.

2. You've got a decision to make here regarding how `psql` handles your transactions for you. There's no "preferred" way — it all boils down to which of the following alternatives is least likely to trip you up while you're working on the lab and on your project.

   `psql` defaults to "autocommit" mode, meaning that a `COMMIT` is automatically executed after each SQL command that you run. You don't want this behavior occurring during a transaction. To disable this behavior, run this command each time you start `psql`:

   ```
   \set AUTOCOMMIT off   -- The capitalization matters here.
   ```

   With this approach, you have to remember to run this command every time you start `psql`.

   Alternatively, you can create the file `.psqlrc` in your home directory and put the command above into it. `psql` runs the commands in this file automatically each time your start it. The risk with disabling `AUTOCOMMIT` is that you have to remember to explicitly run `COMMIT` before exiting `psql`, or you'll lose any committed work — including any tables that you created and populated during the session. If you choose this alternative, you may want to remind yourself by using `\echo` in your `.psqlrc` file:

   ```
   \echo ***** Warning --- Disabling AUTOCOMMIT *****
   \set AUTOCOMMIT off
   ```

   The third alternative is to leave the `AUTOCOMMIT` setting on and start each of your transactions with the `BEGIN` SQL statement. When `psql` processes `BEGIN`, it disables `AUTOCOMMIT` until `COMMIT` or `ABORT` is entered, or an error occurs, then it re-enables `AUTOCOMMIT` mode.

3. Executing transactions concurrently requires that you run two `psql` sessions in separate shell sessions and that you be able to single-step the SQL commands in each transactions. One way to single-step transactions is to simply enter the commands one-at-a-time into each `psql` session, but that is tedious and error-prone. It's easier to enter the commands into a `.sql` script and then run the script from within `psql`:

   ```
   \i tr1.sql
   ```

   But, how do you single-step the script's execution? The answer is to run `psql` in single-step mode by passing it the `-s` switch:

   ```
   psql -s
   ```

4. Finally, if you're constantly forgetting to terminate your SQL statements with a semicolon, you can run `psql` in single-line mode, using the `-S` switch:

   ```
   psql -S
   ```

   This causes `psql` to interpret end-of-line as a semicolon.

   Note that you can use as many switches as you want when you start a program:

   ```
   psql -s -S
   ```

   For information on other `psql` options, refer to `psql`'s man page:

   ```
   man psql
   ```

2 Transactions Basics

Work individually on this part, using your personal database, so long as it doesn't contain a table named `movies` or `accounts`. If you can't remember the names of your tables, use `\d` to view them. If you have already have a table named `movies` or `accounts`, the simplest thing to do is to create and use a new database for this lab:

```
create database tpktrlab; -- Use YOUR initials, not mine!
\c tpktrlab   -- Connect to this database.
```

To connect directly to this database when starting `psql`, you would start `psql` this way:

```
psql tpktrlab
```

2.1 "Crashing" a Transaction, I

1. Start `psql` and run the `movies.sql` script to create and populate the `movies` table. View the table's data.

2. Start a transaction. (If you've disabled `AUTOCOMMIT`, then you're already in a transaction. Otherwise, issue the `BEGIN` command.)

3. Update the table data in one or more rows. (Personally, I think the freshness and score attributes for *Shrek* are way too high.)

4. Exit (`\q`) `psql` without committing or aborting the transaction. This simulates a "crashed" transaction. Re-start `psql` and view the table data.

2.1. Did your changes survive the "crash"? Explain.

2.2 "Crashing" a Transaction, II

1. Repeat the previous transaction experiment, this time committing the changes before "crashing."

2.2. Did your changes survive the "crash"? Explain.

2.3 Using Savepoints

1. Start a transaction and update a few movies records. View the movies table to verify the updates.

2. Use the `SAVEPOINT` command to create a savepoint.

3. Insert two new rows into movies. Verify the inserts.

4. Create a second savepoint.

5. Simulate an erroneous update by running

```
update movies set score=42;
```

   Verify that the score attribute of each of the rows of movies is now 42.

6. Use `ROLLBACK TO SAVEPOINT` to roll back the transaction to your second savepoint. View the result.

2.3. Describe and explain your observations for this step and the following steps.

7. Use `ROLLBACK TO SAVEPOINT` again to roll back the transaction to your first savepoint. View the result.

8. Try to roll forward to your second savepoint. View the result.

9. Commit the transaction and view the movies table.

3 Concurrent Transactions

Work in your pairs for the remainder of this lab. Use your project database for this work. From your personal account on phoenix, you can connect to your project database using your project username by starting `psql` this way:

```
psql project_database project_username
```

To simulate concurrent transactions, you'll run two `psql` sessions, with one transaction running in each session. One member of each pair should control one session with a second member controlling the second session.

3.1 Verifying Repeatable Read Isolation

1. Start one `psql` session and run the `movies.sql` script. Start a transaction (TR1).
2. Start a second `psql` session and start a transaction (TR2).
3. Using the `SET TRANSACTION` command, set the isolation level of TR2 to repeatable read:

   ```
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
   ```

   and view the movies table.
4. In TR1, update one or more rows of movies, but don't commit TR1. Verify the update.
5. In TR2, view the movies table.
6. Commit TR1.
7. In TR2, view the movies table again. Abort TR2.

3.1. Describe and explain your observations.

3.2 Verifying Read Committed Isolation

1. Repeat the previous experiment, this time using the `READ COMMITTED` isolation level for TR2.

3.2. Describe and explain your observations.

3.3 Verifying Serializable Isolation

1. Design a transaction schedule to verify that the `SERIALIZABLE` isolation level is not subject to nonrepeatable reads and phantom reads. Run your transaction schedule.

3.3. Describe and explain the result of your experiment. Include a copy of the transaction schedule that you used.

3.4 Determining `psql`'s Default Isolation Level

1. Design and conduct an experiment to determine `psql`'s default isolation level.

3.4. Describe and explain the result of your experiment. Include a copy of the transaction schedule(s) that you used.