In-class activities: Dec 4, 2017

Today, you will experiment with transactions. In the first part, we will have a guided activity that explores how transactions behave and interact under different isolation levels. In the second part, you will study the code of an application communicating with a database backend. We will note problems that occur without transactions and how to use transactions to fix them.

Pacing:

In a flipped classroom, the activities give you some structure, but the objective is that everyone can adjust the pace according to their needs. That means that you are in charge of your pace. However, we urge you to maintain focus and efficiency through the activities to ensure that you don’t fall behind.

Quiz:

As usual, there will be a short end-of-class quiz. As before, you should indicate your 3-person team on the quiz. We keep track of that based on table and laptop numbers. Note your table number, indicated on the whiteboard and monitor assigned to your table. There are two laptop numbers, one on yellow sticker and one on white black sticker. Please denote the last 2 digits of the black sticker. Indicate both numbers (yellow and black sticker) if possible, but at least one is necessary. If you cannot find these numbers, please talk to the instructors or TAs before you turn in your quiz.
Step 1: Multiple transactions

You will be simulating 2 separate transactions on your machine. You will do that by starting psql in two separate terminal windows. We will be denoting the windows with the numbers 1 and 2 in these instructions, to make it clear which commands you enter in which window.

Make sure you can see and connect to the database practicedb. If you don’t see it, you can get the data from our first set of activities on Sep 18.

Connect to practicedb in both windows, by running \c practicedb in both.

Make sure you have two psql instances running in the two terminal windows, both connected to the practicedb database. Keep track of which window you designate as “1”, and which you designate as “2”. Subsequent instructions ask you to enter different commands in the two windows, in particular order.
Step 2: When are changes visible?

Start a transaction in window 1:

```
start transaction;
```

Now check the contents of the product table in window 2:

```
select * from product;
```

Now initiate a price update in table product in window 1:

```
update product
set price = price+10
where pname='Gizmo';
```

Now, check again the contents of the product table in window 2:

```
select * from product;
```

Did anything change?

Now, check the contents of the product table in window 1:

```
select * from product;
```


Now, return to window 1 and commit the transaction:

```
commit;
```

Check again the contents of the product table in window 2:

```
select * from product;
```

Explain your observations.
Step 3: Competing changes

Start a transaction in **window 1**:  

```
start transaction;
```

And another in **window 2**:  

```
start transaction;
```

Transaction 1 updates John’s phone number in **window 1**:  

```
update employees
set phone = '1234'
where name = 'John';
```

Transaction 2 updates Cecilia’s phone number in **window 2**:  

```
update employees
set phone = NULL
where name = 'Cecilia';
```

It then proceeds to also update John’s phone number (**window 2**):  

```
update employees
set phone = '9876'
where name = 'John';
```

What happened? Were the transactions able to perform these changes?

Now, go ahead and commit transaction 1 in **window 1**:  

```
commit;
```

Did the commit affect transaction 2? What do you think happened? What are the contents of the Employees table in each window?

Now, let’s rollback transaction 2 (**window 2**):  

```
rollback;
```

What are the contents of the Employees table in **window 2**?
**Step 4: Deadlock**

Start a transaction in **window 1**:

```sql
start transaction;
```

And another in **window 2**:

```sql
start transaction;
```

Transaction 1 updates John’s phone number in **window 1**:

```sql
update employees
set phone = '0000'
where name= 'John';
```

Transaction 2 updates Cecilia’s and John’s phone numbers in **window 2**:

```sql
update employees
set phone = NULL
where name= 'Cecilia';
```

```sql
update employees
set phone = NULL
where name= 'John';
```

We reached this same state in step 3, and transaction 2 is waiting for transaction 1. But now, transaction 1 proceeds to update Cecilia’s phone number (**window 1**):

```sql
update employees
set phone = '5555555'
where name= 'Cecilia';
```

**What happened?**

Now try to commit both transactions.

**What happened? Why? Which changes are reflected in the database?**
**Step 5: Unrepeatable read**

Start a transaction in **window 1**:

```sql
start transaction;
```

And another in **window 2**:

```sql
start transaction;
```

Transaction 1 updates John's phone number in **window 1**:

```sql
update employees
set phone = '555-3245'
where name= 'John';
```

Transaction 2 reads the contents of the Employees table in **window 2**:

```sql
select * from employees ;
```

Now, transaction 1 commits, in **window 1**:

```sql
commit;
```

Transaction 2 again reads the contents of the Employees table in **window 2**:

```sql
select * from employees ;
```

What happened? Check the postgres reference on isolation levels here:
[https://www.postgresql.org/docs/current/static/transaction-iso.html#MVCC-ISOLEVEL-TABLE](https://www.postgresql.org/docs/current/static/transaction-iso.html#MVCC-ISOLEVEL-TABLE)

**What isolation level do you think your transactions are using?**

**Verify that you identified the correct isolation level using the following command:**

```sql
SELECT current_setting('transaction_isolation');
```

**Why do you think this isolation level is chosen as the default?**

Commit or rollback transaction 2 before proceeding to the next step.
Step 6: Changing isolation levels

Start a transaction in **window 1** and set the isolation level to repeatable read:

```sql
start transaction;
set transaction isolation level repeatable read;
```

And another in **window 2** and set the isolation level to repeatable read:

```sql
start transaction;
set transaction isolation level repeatable read;
```

Let’s now repeat updates similar to step 5. Transaction 1 updates John’s phone number in **window 1**:

```sql
update employees
set phone = '123-4567'
where name= 'John';
```

Transaction 2 reads the contents of the Employees table in **window 2**:

```sql
select * from employees ;
```

Now, transaction 1 commits, in **window 1**:

```sql
commit;
```

Transaction 2 again reads the contents of the Employees table in **window 2**:

```sql
select * from employees ;
```

**How are things different this time?**

Commit transaction 2 in **window 2**, and then read the employee table again (this query is effectively a transaction in itself):

```sql
commit;
select * from employees ;
```

**What data do you see now?**
Step 7: Video rental application over IMDB and a customer database

Download the code for this exercise from our schedule page. The file is most likely in your Downloads folder. Unzip the file, open a terminal window, and cd to the directory VideoStore. Most likely: cd Downloads/VideoStore

You need to add the jar file to your classpath, and compile the included code with the following commands:

```bash
export CLASSPATH=.:postgresql-9.2-1002.jdbc4.jar
javac -g VideoStore.java Query.java
```

Before you can run the application, you need to create a new database. Start Postgres and create a database called customer:

```sql
create database customer;
```

Connect to the new database with `\c customer` and either import or copy-paste the contents of `setup.sql`.

The database has the following schema:

```sql
RentalPlans(pid, name, max_movies, fee)
Customers(cid, login, password, fname, lname, pid)
MovieRentals(mid, cid, status)
```

RentalPlans contains information on the available plans the Video Store offers to their customers; they have a plan id (pid), a name for the plan, a maximum number of movies a customer with that plan can rent (max_movies), and the monthly cost of the plan (fee). Customers contains information on customers; customer id, login and password information, first and last names, and finally, the rental plan the customer is signed up for (pid). MovieRentals records the movie ids (mid) that a customer (cid) has rented, and their status, which can be open or closed. Open means the movie is currently rented out to the customer, whereas closed means it has been returned.

You are now ready to start the Video Store app from your terminal window. We will login as user george with password 123.

```bash
java VideoStore george 123
```
Step 8: Video rental transactions

Spend a few minutes to explore the app. It is a very basic command-line interface for the Video Store. From the app, you can:

- Search for movies by words or strings in the title name.
- View a list of rental subscription plans, and change your plan.
- Rent a movie by its IMDB ID number.
- Return a rented movie, again by its IMDB ID number.
- (Fastsearch is not implemented, and currently does nothing.)

Note that george (the user you are logged in as) is signed up for the basic rental plan, which only allows for one rental at a time. You can see the rentals he has by running return with no argument. In order to rent another movie (without returning this one), George would need to sign up for a more expensive plan. He can do that with the command, e.g., plan 2.

Once you’ve tried out a few commands to get a sense of how the app works, open Query.java, which implements the functionality for most of the functions.

In the beginning, you have the declaration of many prepared statements, which are parameterized queries. Starting at line 282, you will see some methods, labeled as transactions, that implement various actions, such as login, returning or renting movies, etc. Some of them are implemented as SQL transactions (e.g., transaction_return) and others are not, and they don’t necessarily need to be.

**transaction_choose_plan**

Take a closer look at the method transaction_choose_plan. This method is called when a user requests to sign up for a different plan. Since a user cannot have more movies rented than a plan allows, the method first checks whether the user is currently renting more movies than the maximum number of rentals under the new plan. If that’s the case, the system responds with an error, otherwise it proceeds with updating the user’s plan.

You can test this functionality through the app by upgrading / downgrading plans.

```
Unfortunately, there is an issue with the implementation of this method. The way it is implemented, a user could actually end up with more movies than their plan allows. Can you explain how this may happen? Can you demonstrate this issue in practice?

Hint: What if George is logged in to two different instances of the app? What can he do to end up with more movies than his plan allows? Add System.in.read(); somewhere in this method to allow for a pause (waiting for user input) to demonstrate the concurrency issue.
```
Step 9: Challenge – Implement the transaction correctly

Can you revise the method `transaction_choose_plan` to fix the concurrency bug?

If you have a hard time and need to see the solution, it is hidden on this page in white text. Copy-paste the seemingly empty space below to an editor to get the code for this method.

Do not reach for the solution hastily. Put the time to try this out and work through it before you look at the answer.
Optional extra credit group homework

I have decided to offer this mini group homework assignment as an additional extra credit opportunity. Completing this is entirely optional. We are on a tight grading timeframe, so we can only accommodate submissions in groups of three students. Successful completion of the assignment will earn each student in the group 1% of extra credit towards the total course grade. The assignment problem (described below) can be completed with fewer than 50 lines of code. Submissions will be handled through Moodle.
Due date for this extra credit is December 11, 2017, by 11:59pm.

Step 10: Fast search

You may have noticed that the search function of the Video Store is often fast when you provide very specific titles (e.g., Harry Potter and the Chamber of Secrets), but it can get much slower when you provide less specific keywords (e.g., Harry Potter). This is because the existing search function performs a lot of the data processing logic within the Java code: it first retrieves movies that contain the provided string in the title, and then it iterates over this result and issues, for each movie, separate queries to retrieve the directors and actors. So, when many movies contain the provided string or keyword in the title, the application issues many separate queries and things can get slow.

Your task is to implement the fastsearch functionality, by pushing more of the data processing logic into the database engine. The fastsearch command in the interface invokes transaction_fast_search from Query.java, so that is the method that you should modify. Your fastsearch should return only (1) the movie information (id, title, year), (2) its actors, and (3) its director. It does not need to return the rental status. (This is the same information as the search function returns, except the rental status.) Your fast search function should only issue 2 or 3 SQL queries, in contrast with regular search, which issues many queries, depending on how many movies match the search.

Note: Fastsearch may end up not beating (and may even be slower than) regular search for some searches. This is normal. However, fastsearch should be significantly faster than search for less specific searches (e.g., Nixon, nowhere, etc.).

Hint: You can do it with three queries: One query finds all movies matching the keyword; one query finds all directors of all movies matching the keyword; one query finds all the actors of all the movies matching the keyword. Execute each of these three queries separately. You then need to merge the results of the three queries in the Java code. The merge will be easier if your SQL queries sort the answers by the movie id. (There is also a way to write fastsearch with only two, or even only one single SQL query, but it gets messy with questionable benefits.)
Setup:
To work on this outside of class, you will need to:
1. Set up postgres on your own machine:
2. Set up the IMDB database (available on Moodle).
3. Modify the file dbconn.conf in the VideoStore directory with your postgres username and password if you use one.

To compile and run the code:
**Linux / Mac**
cd /where/you/unzipped/the/code
export CLASSPATH=.postgresql-9.2-1002.jdbc4.jar
javac -g VideoStore.java Query.java
java VideoStore user password

**Windows**
cd \where\you\unzipped\the\code
[replace the directory below with your JDK's bin directory]
path C:\Program Files\Java\jdk1.6.0_25\bin;%path%
set CLASSPATH=.postgresql-9.2-1002.jdbc4.jar
javac -g VideoStore.java Query.java
java VideoStore user password

**Deadline:** December 11, 2017, by 11:59pm

**What to submit:** Query.java. We will not accept any other files. Do not group files in a zip or other archive and do not submit in any other format.

**What to change:** Query.java contains comment sections in 3 places that indicate where you should enter code for fast search. The main part is in the corresponding method, transaction_fast_search, but you also need to declare the appropriate queries and prepare statements earlier. Do not change or remove the timing commands (we will check). In total, you need fewer than 50 lines of code.

**Groups:** Only submissions from groups of 3 students will be accepted. We may consider limited exceptions for groups of 2 if there are extenuating circumstances, and only with prior agreement. We will not accept any individual submissions.

**Where to submit:** Moodle. You will first need to create a self-selected group to gain access to the submission link. Only one person per group should submit.

**Evaluation:** We will compile your Query.java with the rest of the code. We will test performance with a set of sample searches.

*Submissions that do not follow the submission requirements will not be graded.*
*Submissions that do not compile will not be graded.*