CS1555 Recitation 12 - Solution

Objective:
1. To get started with JDBC and demonstrate transaction concurrency control on PostgreSQL.
2. To practice Recovery.

PART 1:

Transactions are characterized by the ACID properties (Ref: Wikipedia):

- o Atomicity — Each transaction is treated as a single "unit".
- o Consistency — Any data written to the database must be valid according to all defined rules.
- o Isolation — Concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.
- o Durability — Once a transaction has been committed, it will remain committed even in the case of a system failure.

Isolation is ensured by Concurrency Control, which synchronizes the execution of transactions to ensure each executes in an isolation manner.

By SQL standard and PostgreSQL, transactions can execute under different isolation levels:

Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom Read**	**Serialization Anomaly**
Read uncommitted	Allowed, but not in PostgreSQL	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PostgreSQL	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

PostgreSQL supports multi-version concurrency control. When an MVCC database needs to update an item of data, it will not overwrite the original data item with new data, but instead creates a newer version of the data item. Thus, there are multiple versions stored. Therefore, dirty reads will never happen, i.e. PostgreSQL's Read Uncommitted mode behaves like Read Committed.

Read Committed is the default isolation level in PostgreSQL. In read committed isolation level, two successive SELECT commands can see different data, even though they are within a single transaction.

The Repeatable Read isolation level only sees data committed before the transaction began. This is a stronger guarantee than is required by the SQL standard for this isolation level and prevents both nonrepeatable read and phantom read.

The Serializable isolation level provides the strictest transaction isolation. It works exactly the same as Repeatable Read except that it also prevents serialization anomalies.
(More details at: https://www.postgresql.org/docs/current/transaction-iso.html)

In additional to setting isolation levels, you may also place locks explicitly. There are two kinds of locks: (1) read lock or shared lock and (2) write lock or exclusive lock.

• SHARED
Shared locks allow multiple transactions to read data, but do not allow any transaction to change that data. Multiple transactions can hold shared locks simultaneously.

• EXCLUSIVE
An exclusive lock allows only one transaction to update a particular piece of data (insert, update, and delete). When one transaction has an exclusive lock on a row or table, no other lock of any type may be placed on it.

Before we start:

- Download `rec12db.sql, TranDemo1.java, TranDemo2.java` from class website.
- Download the PostgreSQL JDBC Driver from below link:
  - https://jdbc.postgresql.org/download/postgresql-42.2.5.jar
- Put above files into one folder
- Open DataGrip and 2 terminal windows.
  - In DataGrip, we run queries to keep track of changes in the database.
  - In the Terminal 1, we run `TranDemo1.java`
  - In the Terminal 2, we run `TranDemo2.java`

Example 0: Getting Started

- Edit the `TranDemo1.java` and `TranDemo1.java`, change the username and password to your username and password that you use to login to the PostgreSQL server.
- Compile the Java files using:
  - `javac -cp postgresql-42.2.5.jar TranDemo1.java`
  - `javac -cp postgresql-42.2.5.jar TranDemo2.java`
- Execute `rec12db.sql` in DataGrip.
- In Terminal 1, run the below command:
  - `java -cp postgresql-42.2.5.jar:. TranDemo1 0`
- Now read the demo source file to learn how it works. Note in the file:
  - How to connect to the DB.
  - How to execute an SQL statement.
  - How to iterate through the results set.

Notes:

- To run any of the examples, pass the example number as an argument.
- We will start running 2 transactions concurrently by running TranDemo1 and TranDemo2.
  - Run TranDemo1 first and then run TranDemo2 while TranDemo1 is still running.
  - Notice in the source codes how to group SQL statements into one transaction, commit/rollback the transaction and how to set isolation level for the transaction.
  - The sleep (milliseconds) function is used to force the statements in both transactions to execute in the order we want.

<u>Example 1: Multi-version Concurrency of PostgreSQL</u>

TranDemo1 (read committed)	TranDemo2 (read committed)
update class set max_num_students = 5 where classid = 1 sleep... **rollback**	SELECT * FROM class where classid = 1

Question: What is the max_num_students as read by TranDemo2?

<u>Answer:</u> Because PostgreSQL supports multi-version concurrency control, TranDemo2 read the committed value of max_num_students (i.e., before the update of TranDemo1). Therefore, TranDemo2 does not read dirty data and also does not have to wait for TranDemo1 to release the write lock (exclusive lock).

<u>Example 2: (Implicit) Unrepeatable Read Problem</u>

TranDemo1 (read committed)	TranDemo2 (read committed)

select max_num_students, cur_num_students
from class **where** classid = 1
sleep...
 	
if (cur_num_students < max_num_students)
    update class
    set cur_num_students = cur_num_students +1
    where classid = 1
else
    print 'the class is full'
commit
	
	select max_num_students, cur_num_students
	from class where classid = 1
	
	sleep...
	
	if (cur_num_students < max_num_students)
	    update class
	    set cur_num_students = cur_num_students +1
	    where classid = 1
	else
	    print 'the class is full'
	commit

Question: What is the value of cur_num_students for class with classid = 1? Compare it to the max_num_classes.

<u>Answer:</u> Both of the two transactions registered for class 1, updating the cur_num_students to 3 even though the maximum number of students allowed in this class is only 2. The reason is, both of them read 1 as the current number of students in the class. This is called an implicit case of unrepeatable read: at the time TranDemo2 tried to update cur_num_students, it read that the value is still 1 while it has been updated to 2 (i.e., if it reads the value again at this point, the value will be different).

Example 3: Serializable Isolation Level:

The same as Example 2, but each transaction has the isolation level of **serializable**
Before running example 3, reset the database by rerunning rec12db.sql in the first terminal window.

Question: What is the value of cur_num_students for class with classid = 1 now? Do both transactions perform the update?

Answer: Only TranDemo1 can register (i.e. update cur_num_students,) successfully. TranDemo2 got the error message: "ERROR: could not serialize access due to concurrent update". In serializable isolation level, PostgreSQL allows a transaction T to update a data item only if no other transaction is committing an update on that data item since T started. In this example, TranDemo1 has committed its update to the data item, preventing TranDemo2 from executing its update statement. Therefore, the data is still consistent: the cur_num_students is kept less than or equal to the max_num_students. The program should be able to catch this type of error from PostgreSQL and re-run the transaction instead of notifying the application user of the error.

Example 4: Using "for Update"

The same as Example 2, but each transaction uses the following statement to select max and current number of students:

```
SELECT max_num_students, cur_num_students
FROM class where classid = 1
for update;
```

Again, before running example 4, reset the database by rerunning rec12db.sql in the first terminal window.

Question: What is the value of cur_num_students for class with classid = 1 now? Do both transactions perform the update?

Answer: Only TranDemo1 can register (i.e. update cur_num_students,) successfully. TranDemo2 got the user-friendly error message: "the class is full". When a transaction executes a "select…for update" statement, it acquires an exclusive lock on the selected rows. This means that when TranDemo1 read the cur_num_students of class 1, it also kept an xlock on the corresponding row(s). Later, when TranDemo2 tried to read cur_num_students, it has to first ask for the xlock. Because TranDemo1 has the xlock, TranDemo2 has to wait until TranDemo1 commits and releases the lock. Therefore, the outcome of this example is the same as when the 2 transactions run sequentially.

<u>Example 5: Deadlock</u>

TranDemo1 (read committed)	TranDemo2 (read committed)
update class **set** max_num_students = 10 **where** classid = **1**	
sleep...	**update** class **set** max_num_students = 20 **where** classid = **2**
update class **set** max_num_students = 10 **where** classid = **2** **commit**	sleep...
	update class **set** max_num_students = 20 **where** classid = **1** **commit**

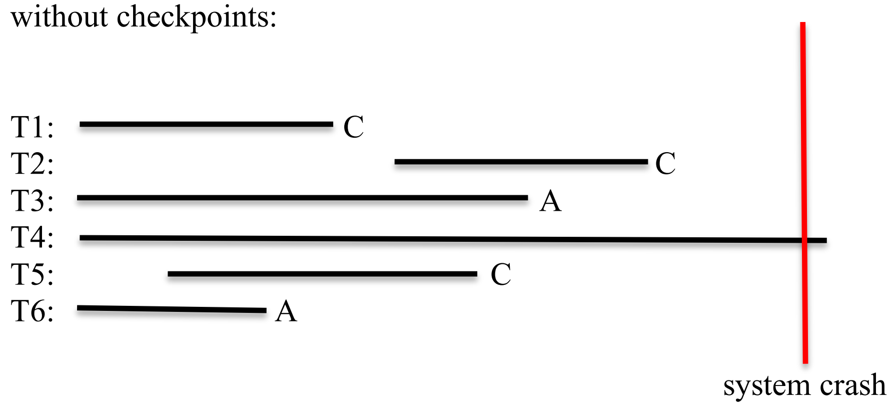
<u>Question:</u> What is the value of max_num_students? Do both transactions perform their updates?

<u>Answer:</u> Deadlock happens. One of the two transactions is selected as the victim and rollbacks. The victim transaction receives an error message that a deadlock is detected. The other transaction runs normally.

PART 2:

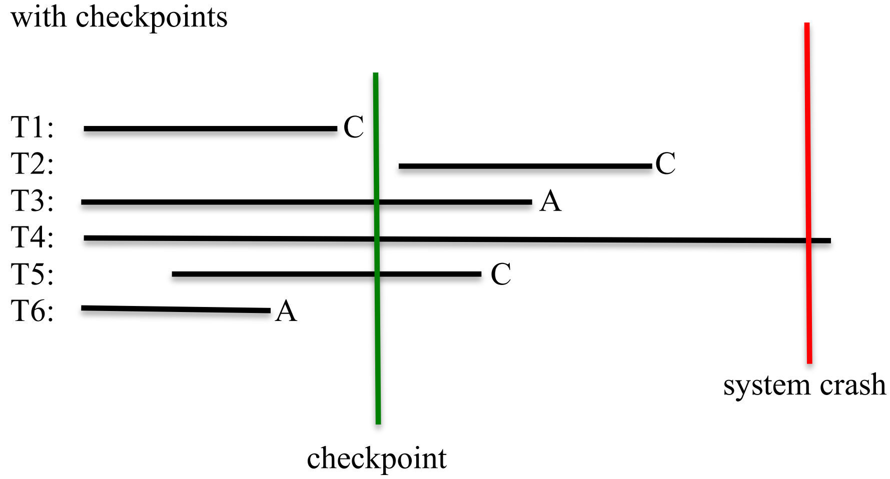
For the following transaction executions, state what the system should do when it restarts after a crash:

a. without checkpoints:



REDO (to preserve DURABILITY)	T1, T2, T5
UNDO (to preserve ATOMICITY)	T3, T4, T6

b. with checkpoints



REDO (to preserve DURABILITY)	T2, T5
UNDO (to preserve ATOMICITY)	T3, T4

T1 is not included anymore because it committed before the checkpoint;
T6 is not included anymore because it aborted before the checkpoint and the system already rolled back everything at the checkpoint