

## CS1555 Recitation 8

---

### Objective:

1. To get started with JDBC and demonstrate transaction concurrency control on PostgreSQL
  2. To practice Views
- 

### **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needs to update an item of data, it will not overwrite the original data item with new data, but instead creates a newer version of the data item.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 addition 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.

## **PART 2:**

### **Before we start:**

- Download `rec8db.sql`, `TranDemo1.java`, `TranDemo2.java` from course 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 `TranDemo2.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 `rec8db.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.

### Example 1: Multi-version Concurrency of PostgreSQL

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

Question: What is the max\_num\_students as read by TranDemo2?

### Example 2: (Implicit) Unrepeatable Read Problem

TranDemo1 (read committed)	TranDemo2 (read committed)
<pre>select max_num_students, cur_num_students from class where classid = 1 sleep...  if (cur_num_students &lt; max_num_students)     update class     set cur_num_students = cur_num_students + 1     where classid = 1 else     print 'the class is full' commit</pre>	<pre>select max_num_students, cur_num_students from class where classid = 1  sleep...  if (cur_num_students &lt; max_num_students)     update class     set cur_num_students = cur_num_students + 1     where classid = 1 else     print 'the class is full' commit</pre>

Question: What is the value of cur\_num\_students for class with classid = 1? Compare it to the max\_num\_classes.

### 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 rec8db.sql in DataGrip.

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

#### Example 4: Using for Update of

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 rec8db.sql in DataGrip.

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

#### Example 5: Deadlock

TranDemo1 (read committed)	TranDemo2 (read committed)
<b>update</b> class <b>set</b> max_num_students = 10 <b>where</b> classid = <i>1</i> sleep...  <b>update</b> class <b>set</b> max_num_students = 10 <b>where</b> classid = <i>2</i> <b>commit</b>	<b>update</b> class <b>set</b> max_num_students = 20 <b>where</b> classid = <i>2</i>  sleep...  <b>update</b> class <b>set</b> max_num_students = 20 <b>where</b> classid = <i>1</i> <b>commit</b>

Question: What is the value of max\_num\_students? Do both transactions perform their updates?

### **PART 3:**

#### **Before we start:**

- Download and run the below to build the database:
  - o Student\_DB.sql

1. Create a view called student\_courses that lists the SIDs, student names, number of courses in the Course\_taken table.

2. Create a materialized view called mv\_student\_courses that lists the SIDs, student names, number of courses in the Course\_taken table.

3. Execute the following commands. Compare the query results and time used of the two select statements.

```
insert into course_taken (course_no, sid, term, grade)
values ('CS1555', '129', 'Fall 18', null);
commit;

--REFRESH MATERIALIZED VIEW mv_student_courses;
select * from mv_student_courses;
select * from student_courses;
commit;
```

4. Reset the database by running the Student\_DB.sql and recreate the views. Comment back the line beginning with “REFRESH” in the above commands and execute the commands. Compare the query results of the two select statements.