Lab 9 : CS 387 Jan-Apr 2024

Understanding concurrent transactions and isolation schedules in postgres

Submission instructions: Note down the answers to various questions neatly, in a text file called answers.txt . Mark the answers on SAFE-Beta app when the quiz has started. Also submit the file answers.txt on vlab, as a backup.

Database to use: The examples below are to be run on the small university database, not on the large university database. The relevant DDL and data sql files are given. Call your database as univ9.

Part A: Understanding transaction commit and rollback

In this exercise, you will see how to rollback or commit transactions. By default PostgreSQL commits each SQL statement as soon as it is submitted. To prevent the transaction from committing immediately, you have to issue a command `begin;` to tell PostgreSQL to not commit immediately. You can issue any number of SQL statements after this, and then either commit; to commit the transaction, or rollback; to rollback the transaction. To see the effect, execute the following commands one at a time:

```
begin ;

select * from student where name = 'Tanaka';

delete from student where name = 'Tanaka';

select * from student where name = 'Tanaka';

rollback;

select * from student where name = 'Tanaka';
```

Q-A1: At the end of the above statements, is there a student with name ‘Tanaka’ in the database? **Q-A2:** Explain your observation.

Background on concurrent transactions, transaction isolation levels

- PostgreSQL implements concurrent transactions at different configurable isolation levels.
- In the read committed isolation level, each statement sees the effects of all preceding transactions that have committed, but does not see the effect of concurrently running transactions (i.e. updates that have not been committed yet)
- Read committed isolation provides only a low level of consistency and it can cause problems with transactions. It is safer to use the serializable level if concurrent updates occur with multiple statement transactions.
- In snapshot isolation, where a transaction gets a conceptual snapshot of data at the time it started, and all values it reads are as per this snapshot.
- In snapshot isolation, if two transactions concurrently update the same data item, one of them will be rolled back.
- Snapshot isolation does NOT guarantee serializability of transactions. For example, it is possible that transaction T1 reads A and performs an update B=A, while transaction T2 reads B and performs an update A=B. In this case, there is no conflict on the update, since different tuples are updated by the two transactions, but the execution may not be serializable: in any serial schedule, A and B will become the same value, but with snapshot isolation, they may exchange values.
- Oracle uses snapshot isolation for concurrency control when asked to set the isolation level to serializable, even though it does not really guarantee serializability.
- Microsoft SQL Server supports snapshot isolation, but uses two-phase locking for the serializable isolation level.
- PostgreSQL versions prior to 9.1 used snapshot isolation when the isolation level was set to serializable.
- However, since version 9.1, PostgreSQL uses an improved version of snapshot isolation, called serializable snapshot isolation, when asked to set the isolation level to serializable. This mechanism in fact offers true serializability, unlike plain snapshot isolation.

Part-B: Concurrent transactions in the default isolation level

1. **Q-B1:** What is the default transaction isolation level in PostgreSQL? You can find this using the command:

   ```
   show transaction isolation level;
   ```

2. In this exercise you will run transactions concurrently from two different psql terminals, to see how updates by one transaction affect another. Use the default transaction isolation level to begin with.

3. Before beginning this exercise, ensure that student Tanaka has the original tot_cred of 120 (the original entry in the given data input)

4. Open two psql connections to the same database. Execute the following commands in sequence in the first window

   ```
   begin ;

   update student set tot_cred = 55 where name = 'Tanaka';
   ```

5. Now in the second window execute

   ```
   begin;

   select * from student where name = 'Tanaka';
   ```

6. **Q-B2:** What is the value of tot_cred for Tanaka seen in the second window? **Q-B3:** Can you figure out why you got the result that you saw? Explain briefly. **Q-B4:** Just based on this observation, what can you tell about the transaction isolation level in PostgreSQL?

7. Now execute commit in the first window, then in the second window. **Q-B5:** Do both transactions complete successfully (without rollback)? **Q-B6:** Why or why not? Explain the above answer briefly.

Part-C: Concurrent updates in the default isolation level

1. Before beginning this exercise too, ensure that student Tanaka has the original tot_cred of 120 (the original entry in the given data input)

2. Now, let us try to update the same tuple concurrently from two windows. In one window execute

   ```
   begin;

   update student set tot_cred = 44 where name = 'Tanaka';
   ```

3. Then in the second window, execute one after another:

```
begin;

select min(tot_cred) from student where name = 'Tanaka';

update student set tot_cred = (select min(tot_cred) from
student where name = 'Tanaka')+20 where name = 'Tanaka' ;
```

4. **Q-C1:** What happens right after the update command is issued in the second window? **Q-C2:** Why? Explain the above observation briefly.

5. Now execute commit in the first window. **Q-C3:** What happens in the second window on executing commit in the first window?

6. Now execute commit in the second window. **Q-C4:** What is the final value of tot_cred for Tanaka? **Q-C5:** What is the reason for the above observation?

Part-D: Concurrent transactions in the serializable isolation level

1. Before beginning this exercise too, ensure that student Tanaka has the original tot_cred of 120

2. Now in both windows, execute the command

   ```
   set transaction isolation level serializable;
   ```

3. **Q-D1:** To set the transaction isolation level, do you have to execute the above set command before or after the begin command?

4. Re-execute the same queries as in Part-C but at the serializable isolation level, and see what happens. **Q-D2:** What happens right after the update command is issued in the second window? **Q-D3:** What happens in the second window on executing commit in the first window? **Q-D4:** What is the final value of tot_cred for Tanaka? **Q-D5:** What is the reason for the above observation?

Part-E: Concurrent updates at the serializable isolation level

1. Before beginning this exercise, ensure that Mozart has a salary of 40000 and Einstein has a salary of 95000 – i.e. the original values in the given data. Run the query:

   ```
   select id, salary from instructor where id in ('22222',
   '15151') ;
   ```

   and verify the results

2. Open two psql terminals like earlier, one each for a two different, concurrent transactions

3. Begin a transaction in each window, both with the serializable isolation level

4. In window-1, execute:

```
update instructor set salary = (select salary from
instructor where id = '22222') where id = '15151';
```

5. In window-2, execute:

```
update instructor set salary = (select salary from
instructor where id = '15151') where id = '22222';
```

6. Commit in window-1 followed by window-2. **Q-E1:** Did transaction-1 complete successfully? **Q-E2:** Why? **Q-E3:** Did transaction-2 complete successfully? **Q-E4:** Why? **Q-E5:** What is the final salary of Mozart? **Q-E6:** What is the final salary of Einstein?