COMP9120 Relational Database Systems

Tutorial Week 9 Solution: Transaction Management

Exercise 1. Transaction Support in SQL

The transaction concept is reflected in several parts of SQL. In SQL, you can group several statements into a transaction. Many SQL dialects, e.g. with SQL Server or also PostgreSQL, use an explicit **BEGIN TRANSACTION** (or **BEGIN**) command to start a transaction. A transaction is requested to finish successfully using **COMMIT** or aborted using **ROLLBACK**.

a) Try out the following small script in PostgreSQL that tries to add three new lecture theatres in the new law building to our University database (using the University schema from previous tutorials): what classrooms starting with LS are shown before and after the **ROLLBACK** keyword?

```
BEGIN;
INSERT INTO Classroom VALUES ('LS101', 300, 'sloping');
INSERT INTO Classroom VALUES ('LS104', 100, 'sloping');
INSERT INTO Classroom VALUES ('LS106', 100, 'sloping');
-- check what we have so far:
SELECT * FROM Classroom WHERE ClassroomId LIKE 'LS%';
-- simulate a problem and abort our transaction
ROLLBACK;
-- check what is kept in the database
SELECT * FROM Classroom WHERE ClassroomId LIKE 'LS%';
```

You see that the database is UNCHANGED after an aborted (rolled-back) transaction.

b) Now try the same script, but with the **ROLLBACK** statement replaced with **COMMIT**.

The changes should now persist even after the transaction end.

Exercise 2. Serializability and Update Anomalies

Given the following relation

Offerings(uosCode, year, semester, lecturerId)

with this example instance:

uosCode	year	semester	lecturerId
COMP5138	2012	S1	4711
INFO2120	2011	S2	4711

Consider the following hypothetical interleaved execution of two transactions T1 and T2 in a DBMS where concurrency control (such as locking) is not done; that is, each statement is executed as it is submitted, using the most up-to-date values of the database contents.

T1	**SELECT** * **FROM** Offerings **WHERE** lecturerId = 4711
T2	**SELECT** year **INTO** :yr **FROM** Offerings **WHERE** uosCode = 'COMP5138'
T1	**UPDATE** Offerings **SET** year=year+1 **WHERE** lecturerId = 4711 **AND** uosCode = 'COMP5138'
T2	**UPDATE** Offerings **SET** year=:yr+2 **WHERE** uosCode = 'COMP5138'
T1	**COMMIT**
T2	**COMMIT**

Indicate:

a) the values returned by the **SELECT** statements and the final value of the database;

T1: sees both tuples of initial table content: (COMP5138, 2012, S1, 4711) and (INFO2120,2011,S2,4711)
T2: sees the original year of COMP5138: 2012
Final database state: (COMP5138, 2014, S1, 4711), second tuple unmodified

b) whether the execution produces any update anomalies

lost-update problem for T1's change of the year of 'COMP5138' (overwritten by T2)

c) whether the execution is conflict serializable or not.

non-conflict serializable: the end-effect of this is neither the same as T1 before T2 or T2 before T1

Exercise 3. Transaction Isolation experiment

In this exercise, we are going to give you some intuition of transaction management. You will begin by opening pgAdmin in two windows.

Execute the following two transactions line-by-line as seen in the following figure (press 'Execute' after each line) and write down the results of the SELECT COUNT() statements:

Solution:

The output of the following table will vary depending on how many rows you have in student table. Let's assume you have N rows in student table.

Window 1	**Window 2**	**Count**
BEGIN TRANSACTION;	BEGIN TRANSACTION; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	
SELECT COUNT(*) FROM student;		N
	SELECT COUNT(*) FROM student;	N
INSERT INTO student VALUES (); //insert test values		
SELECT COUNT(*) FROM student;		N+1
	SELECT COUNT(*) FROM student;	N
COMMIT;		
SELECT COUNT(*) FROM student;		N+1
	SELECT COUNT(*) FROM student;	N
	COMMIT;	
	SELECT COUNT(*) FROM student;	N+1