COMP9120 Relational Database Systems **Semester 2, 2016**

Tutorial Week 9 Example Solution: Transactions

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` command to start a transaction. In Oracle, the beginning of a transaction is implicit. A transaction is requested to finish successfully using `COMMIT` or aborted using `ROLLBACK`.

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

```
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. While your transaction is still active (before the rollback command) you see your own changes.

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 serializable or not.

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

Exercise 3. Transactions with JDBC

In JDBC, by default each single SQL statement is executed as a separate database transaction and is automatically committed immediately after its execution. You can change this behaviour as follows:

(i) Turn off auto-committing of statements made to the JDBC connection with `conn.setAutoCommit(false);`

(ii) With auto-commit disabled, a transaction is implicitly started with the first SQL statement issued against the database. All following SQL statements share this open transaction now.

(iii) You must add `conn.commit()` (or `conn.abort()`) calls after the last statement that should be part of a transaction.

Now have a look at the transaction support in our JDBC client from last week.

a) Check whether the JDBC program `JDBCclient.java` from last week is using transactions.

Yes and no. We have no explicit transactions in the example JDBCclient.java so far, but rather use the default behaviour of JDBC which is AutoCommit for each SQL statement separately. But that's not a good thing if the intermediate steps take the database into an inconsistent state.

b) Extend the JDBC program from last week to use explicit transactions for each function.

The example below shows how to delimit a transaction in java using the setAutoCommit(false) and commit() methods. Within listUnits() :

```
conn.setAutoCommit(false);
/* Implicit BEGIN TRANSACTION because
we switched off auto-commit */

  /* prepare a dynamic query statement */
  PreparedStatement stmt = conn.prepareStatement(
    "SELECT uosCode, uosName, credits, semester, year "
     + "  FROM UoSOffering JOIN UnitOfStudy USING (uosCode)"
     + " ORDER BY uosCode,year,semester");

      /* execute the query and loop through the resultset */
  ResultSet rset = stmt.executeQuery();
  int nr = 0;
  while ( rset.next() )
  {
     nr++;
     System.out.println(rset.getString("uosCode")
        + " - " + rset.getString("uosName")
        + " ("  + rset.getInt("credits")
        + "cp) "+ rset.getInt("year")
        + "-"   + rset.getString("semester"));
  }

  if ( nr == 0 )
     System.out.println("No entries found.");

/* END OF TRANSACTION */
conn.commit();
```