Database Tutorial 5: Relational Algebra and Transactions
Sample Solution - 7 December 2018

1. Let's assume a following schema

   Airports(code, city)
   FlightCodes(code, airlineName)
   Flights(departureAirport, destinationAirport, departureTime, arrivalTime, code)
   departureAirport → Airports.code
   destinationAirport → Airports.code
   code → FlightCodes.code

   a. Below is an SQL query that finds all airports that have departures or arrivals (or both) of flights operated by Lufthansa or SAS (or both). Express this query by a relational algebra expression.

```
SELECT DISTINCT served
FROM ((SELECT destinationAirport AS served, airlineName
        FROM FlightCodes JOIN Flights ON Flights.code = FlightCodes.code)
UNION
       (SELECT departureAirport AS served, airlineName
        FROM FlightCodes JOIN Flights ON Flights.code = FlightCodes.code)) AS D
WHERE D.airlineName = 'Lufthansa' OR D.airlineName = 'SAS';
```

$$\delta(\pi_{served}(\sigma_{airlineName='Lufthansa' \text{ OR } airlineName='SAS'}(\rho_{D(served,airlineName)}(\pi_{destinationAirport,\ airlineName}(FlightCodes \bowtie_{Flights.code=FlightCodes.code} Flights) \cup \pi_{departureAirport,\ airlineName}(FlightCodes \bowtie_{Flights.code=FlightCodes.code} Flights))))$$

   b. Translate the following relational algebra expression to an SQL query:

$$\pi_{First.departureTime,\ Second.arrivalTime}((\rho_{First}(Flights)) \bowtie_{First.destinationAirport=Second.departureAirport} (\rho_{Second}(Flights)))$$

```
SELECT First.departureTime, Second.arrivalTime
FROM Flights AS First JOIN Flights AS Second
ON First.destinationAirport = Second.departureAirport;
```

2. A common situation in flight booking is the following:
   - User A wants to find a flight from X to Y.
   - The system shows available flights. One of them has only one seat left.
   - Before A makes a choice, user B also wants to find a flight from X to Y.

- The system shows the same list of flights to B.
- B selects immediately the flight that has only one seat left.
- After this, A tries to select the same flight, but this fails, because B has taken the last seat.

Explain this situation in transaction concepts:

- What kind of interference has taken place (dirty or unrepeatable read, phantom, …)?
- Which isolation levels would permit this situation?

Non-repeatable read, permitted by READ COMMITTED and READ UNCOMMITED

3. Authorization, SQL Injection, Transactions. Consider an existing database with the following database definition in a PostgreSQL DBMS:

```
CREATE TABLE Users (
    id INTEGER PRIMARY KEY,
    name TEXT,
    password TEXT
);

CREATE TABLE UserStatus (
    id INTEGER PRIMARY KEY REFERENCES Users,
    loggedin BOOLEAN NOT NULL
);

CREATE TABLE Logbook (
    id INTEGER REFERENCES Users,
    timestamp INTEGER,
    name TEXT,
    PRIMARY KEY (id, timestamp)
);
```

a. A database user "Alice" is granted the following permissions:

```
GRANT SELECT(id, name, password) ON Users TO Alice;
GRANT SELECT(id, loggedin) ON UserStatus TO Alice;
GRANT SELECT(id, timestamp, name) ON LogBook TO Alice;
GRANT INSERT(id, timestamp, name) ON LogBook TO Alice;
```

Alice now executes the following SQL statement:

```
INSERT INTO LogBook
        SELECT u.id, 201701101400, u.name
        FROM (UserStatus us JOIN Users u ON us.id = u.id)
        WHERE us.loggedin = TRUE;
```

We want Alice to only have exactly the privileges that are necessary to complete this SQL statement. Does Alice have too few, exactly enough, or too many privileges? What minimal set of permissions should she be granted instead, if not the same as listed above?

Alice has too many privileges, since she does not need to read the password in the Users table, nor the LogBook entries. The minimally required set of permissions is:

GRANT SELECT(id, name) ON Users TO Alice;
GRANT SELECT(id, loggedin) ON UserStatus TO Alice;
GRANT INSERT(id, timestamp name) ON LogBook TO Alice;

b. Users of a web application are allowed to query this database for a certain user id. This function implemented in JDBC using the following code fragment:

```
…
String query=
        "SELECT * FROM UserStatus WHERE id = ' " + userinput + " ' ";
PreparedStatement stmt = conn.prepareStatement(query);
ResultSet rs = stmt.executeQuery();
…
```

Does this code contain an SQL injection vulnerability? If it does not, why not? If it does, how would you correct the code?

Yes this code contains an SQL injection vulnerability. The vulnerability can be removed by either correctly sanitizing or escaping the data in the *userinput* variable. A better solution is to use a PreparedStatement with placeholder:

```
…
String query = "SELECT * FROM UserStatus WHERE id = ?";
PreparedStatement stmt = conn.preparedStatement(query);
Stmt.setString(1, userinput);
ResultSet rs = stmt.executeQuery();
…
```

c. The JDBC code below is supposed to run a batch job in the following way: Loop through an array of users, set each users password hash to a certain value and add a logbook message for that user. If any of the users do not exist, the whole batch job should be rolled back.

The queries seem to work fine, and if there is an invalid id in the array the error message is printed, but all the other ids in the array are still changed! What goes wrong and how can it be fixed (Hint: two separate things need to be fixed, think about what happens both before and after the error).

```
int[] blockList = {11,42,55}; // Ids to be locked
PreparedStatement update = conn.prepareStatement(
  "UPDATE Users SET password='qiyh4XPJGsOZ2MEAy' WHERE id=?");
PreparedStatement insert = conn.prepareStatement(
  "INSERT INTO Logbook VALUES (?,now(),'Account locked')");
conn.setAutoCommit(false); // Enable transactions

for(int user : blockList){ // For each user in blocklist ...
    update.setInt(1,user);
    insert.setInt(1,user);
```

```
    int res = update.executeUpdate();
    if (res == 0){ // 0 rows affected - user does not exist!
      System.err.println("Error, missing id: "+user);
      conn.rollback();
    } else{
      insert.executeUpdate();
      conn.commit();
    }
}
```

The two problems in the code are:
1. A commit is performed at the end of each iteration, starting a new transaction. The commit needs to be moved outside the loop.
2. The rollback does not terminate the loop, so it keeps going in a new transaction. Triggering the error needs to terminate the loop using break/return/throw.

Correct code:

```
for(int user : blockList){
    update.setInt(1,user);
    insert.setInt(1,user);
    int res = update.executeUpdate();
    if (res == 0){
      System.err.println("error, missing id: "+user);
      conn.rollback();
      break;
    }
    insert.executeUpdate();
}
conn.commit();
```