ECE 356 Winter 2019: Lab 3: Transactions and Performance

Consider a small variation of the schema from the midterm. It is a small database for a university course scheduling system, providing details about courses, offerings, instructors, classrooms and departments.

Instructor (instID, instName, deptID, sessional)
Course (courseID, courseName, deptID, prereqID)
Prereq (courseID, prereqID)
Offering (courseID, section, termCode, roomID, instID, enrollment)
Classroom (roomID, building, room, capacity)
Department (deptID, deptName, faculty)

Explanation:

- **Instructor** defines a unique instructor ID, his/her name, department and sessional status.
- **Course** defines a unique course ID, the course name, the department offering that offers the course.
- **Prereq** defines any prerequisites.
- **Offering** defines an actual offering of a course; the offering comprises the courseID being offered, the section number (integers starting at 1), the term code (the standard UW 4-digit term code: the first three digits define the year (add 1900 to get the year), and the fourth digit is the month in which the course starts (1 (Jan), 5 (May), or 9 (Sept) for Winter, Spring, and Fall offerings, respectively), the room where the section meets, the instructor, and the number of students enrolled.
- **Classroom** defines a unique room ID, together with the building, room number and room capacity.
- **Department** identifies a unique department ID and its name.

SQL code for this database, together with primary and foreign keys, has been provided in the source file `createUni.sql`.

In such a database it would be reasonable that we would want to collect multiple SQL statements together and form a single transaction. For example, there are two sections of ECE 356, but the enrollment between them is quite unbalanced. It would therefore be reasonable to want to move some of the enrollment from one section to another. Doing so, however, should be contingent on there being sufficient room within the classroom receiving the additional students. We would therefore want something like the following:

```
BEGIN;
update Offering set Enrollment = Enrollment - 20
       where courseID="ECE356" and section=2 and termCode=1191;
update Offering set Enrollment = Enrollment + 20
       where courseID="ECE356" and section=2 and termCode=1191;
COMMIT;
```

though obviously we would need some additional work to ensure that the capacity had not been exceeded.

In order to do transactions using the command-line interface (CLI) the first thing that is required is to turn off `autocommit` since by default the CLI treats each operation as a transaction. To find the value of any system variable, use the "`show variables`" command:

```
mysql> show variables like "autocommit";
show variables like "autocommit";
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)
```

Since we do not want it on, we turn it off:

```
mysql> set autocommit=0;
set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like "autocommit";
show variables like "autocommit";
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | OFF   |
+---------------+-------+
1 row in set (0.01 sec)
```

See `https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html` for further details on system variables. This change enables the use of transactions with the CLI (or when executing from a `.sql` source file.

We now wish to examine the effect of different isolation levels. The default isolation level is `REPEATABLE-READ`. To change it, use the command:

```
set [global|session] transaction_isolation [level]
```

where the isolation level is one of:

```
READ-UNCOMMITTED
READ-COMMITTED
REPEATABLE-READ
SERIALIZABLE
```

Now, make two separate connections to the same database. In one, we will change the enrollment for one of the courses within a transaction:

```
BEGIN;
update Offering set Enrollment = Enrollment - 20
       where courseID="ECE356" and section=2 and termCode=1191;
```

Now, before doing the second update, and before doing the commit, in the second session connected to the database, determine what happens when you attempt to see what the enrollment is for the course. In particular, you should determine what the different results are for the different possible combinations of isolation level and report these results in your Lab 3 submission.

For the second part of this lab, you are required to create a stored procedure that does the following:

- Accepts five input parameters: courseID, section1, section2, termCode, quantity
- execute as a transaction, the following:
- if (courseID,section1,termCode) or (courseID,section2,termCode) do not exist in Offering, or quantity is 0 or less or section1 = section2, set the error code to -1.
- attempt to reduce the enrollment in section1 by “quantity”; if the result is a negative enrollment, set the error code to -2.

- attempt to increase the enrollment in section2 by "quantity"; if the result is that enrollment in section2 exceeds room capacity, then set the error code to -3.
- if there are any errors, rollback the transaction
- if there are no errors, set the error code to 0 and commit the transaction.

You should call your procedure "switchSection" and you should develop a set of test cases to verify its correct functionality. You should submit your code and test cases as part of your Lab 3 submission.

The third part of this lab requires you to become familiar with the performance_schema part of MySQL. In particular, we want to know what the actual timed measurements are for queries with and without indexes. For this, we will use the Lahman Baseball database and focus on the query:

```
select nameFirst,nameLast,max(RBI) from Batting
                 inner join Master using (playerID)
                 where HR = 0 limit 1;
```

To determine database performance, you need to study the "MySQL Performance Schema" chapter in the Reference Manual. (It is chapter 22 in v5.6, 25 in v5.7, and 26 in v8.0). To help get a quick start, the following will be useful:

- performance_schema.setup_objects contains the list of things being measured. If you add your events in your database to this, MySQL will start to measure the performance of transactions in your database:

  ```
  insert into setup_objects values ('EVENT','lahman2016','%','YES','YES');
  ```

  (In my case I added "lahman2016" as that is the name of the database I want information about.)
- Timing information can be acquired from event_transactions_history where the time a transaction took to execute is timer_end - timer_start; this is measure in picoseconds, but is only actually accurate to microseconds, so you should divide the result by 1,000,000 to get it in microseconds.
- First, determine the performance without any indexes. Then consider various possible indexes, including those implied by primary and foreign keys. When measuring the performance, you should do at least five measurements, average them, and also report the minimum and maximum.

You should submit a report of what your performance measurements are as the third part of your Lab 3 submission.