Assignment 4: Advanced SQL Programming

We will supply templates for the files in this assignment. As usual, they must follow the style and formatting specified in Assignment 2. When you are finished, submit a tarball or zip-file named **`cs121hw4-`*`username`*** so we know who submitted the file.

Suggested Reading

Some supplemental material is included on the course website about materialized views and incremental view maintenance; this can be very helpful for completing Exercise 2. (Although, you will likely be able to figure it out on your own without difficulty.)

Overview

The first problem uses the grades database, specified in **`make-grades.sql`**, and the second problem uses the banking database, specified in **`make-banking.sql`**. You should probably reload those datasets before starting the assignment.

Both exercises require you to create procedural SQL objects. Since both stored procedures and triggers can contain multiple statements, they will have semicolons in their bodies, and thus you will need to have a couple of **`DELIMITER`** commands to load your data properly. For example:

```
-- Note that this occurs before we switch the delimiter.
DROP TRIGGER IF EXISTS trg_foo;

DELIMITER !

-- Do fun stuff and junk.
CREATE TRIGGER trg_foo ...
BEGIN
    ...
END; !  /* Semicolon on this line can be left off too. */

DELIMITER ;
```

Coding Guidelines

When you are creating things like indexes, stored procedures, and triggers, it is very helpful to prefix the object's name with a string indicating the type of object it is. We strongly recommend you follow this guideline in this assignment. For example, a trigger's name would start with "trg_", an index's name would start with "idx_", and a stored procedure's name would start with "sp_". This is a good general practice to follow.

Make sure that all of your code is clearly and concisely commented. Don't write a novel, but don't leave out the comments either.

Follow consistent indentation rules. Please indent with spaces and not tabs!

If your code is sloppy or under/over-commented, points will be deducted. If you want feedback on your coding style before submitting your work, feel free to ask!

Part A: Submit Intervals (40 points)

Scoring: min/max functions are 10pts each (20pts total); avg function is 12pts; index is 8pts.

This exercise uses the grade database specified in **`make-grades.sql`**, so you might want to perform a clean reload of this set of tables.

Since students are allowed to submit work for an assignment multiple times, it would be interesting to know some stats about how much time passes between resubmissions for a particular assignment. We would like to be able to write a query like this:

```
SELECT sub_id,
       min_submit_interval(sub_id) AS min_interval,
       max_submit_interval(sub_id) AS max_interval,
       avg_submit_interval(sub_id) AS avg_interval
FROM (SELECT sub_id FROM fileset
      GROUP BY sub_id HAVING COUNT(*) > 1) AS multi_subs
ORDER BY min_interval, max_interval;
```

Create the user-defined functions referenced above:

1) **`min_submit_interval`**
2) **`max_submit_interval`**
3) **`avg_submit_interval`**

Write them in a file **`submit-stats.sql`**. Each of these functions takes an **`INTEGER`** argument specifying the ID of the submission being investigated. The return-value should be the result in seconds. The max and min functions should return an **`INTEGER`** result, and the average function should return a **`DOUBLE`** value.

(Note: The **`fileset.sub_date`** column is a **`TIMESTAMP`** value; you can use the MySQL **`UNIX_TIMESTAMP()`** function[1] to compute time intervals in seconds.)

Remember that this database has an unusual design. Quoting from Assignment 3:

> The design of this database schema is definitely not ideal; students have a **`submission`** record for *every* assignment, whether they did the assignment or not! One must tell whether the student has submitted work for an assignment by whether the **`submission`** record has any associated **`fileset`** records. In addition, there are assignments that don't require submissions, but that still receive grades; quizzes, for example. In those cases, the **`submission`** record will have a grade assigned to it, but there will be no **`fileset`** records associated with the submission.

In other words, each **`fileset`** record represents a single submission (i.e. a set of files) that a student made for a particular assignment.[2] All **`fileset`** records with the same **`sub_id`** are different submissions from the same student, for a specific assignment. Every fileset record has a **`sub_date`** value specifying when the set of files was submitted.

---

[1] http://dev.mysql.com/doc/refman/5.5/en/date-and-time-functions.html#function_unix-timestamp

[2] In the original database from which this data was collected, **`fileset`** records also included a byte-stream stored in a **`BLOB`** column, constructed by ZIP-compressing all files the student was including in that submission for the assignment.

The result should be **NULL** if the specified submission has less than two filesets associated with it. (Note that the above query excludes submissions with less than two filesets; you will need to test your functions' behaviors separately.)

Here are some implementation tips:

- For the min and max functions, the cleanest and fastest implementation (by a long shot) will be to use a cursor within each of these functions.

  When you use the cursor, you will need to make sure your code detects when the cursor hits the end of its results in every situation; i.e. if the cursor has no rows, or if the cursor only has one row, or if the cursor has more than one row. Presuming you have set up a **CONTINUE HANDLER** as outlined in the Lecture 9 slides, you can write code like this:

```
...
FETCH cur INTO first_val;
WHILE NOT done DO
    FETCH cur INTO second_val;
    IF NOT done THEN
        ... /* do your thing */
    END IF;
END WHILE;
...
```

- You can actually implement the average function without using a cursor. Here's a hint: compute the average manually, as the total interval size divided by the number of intervals. Don't forget that the number of intervals is *one less than* the number of filesets associated with the submission.

Improving Performance

Once you have implemented these functions, try out the above query. If you want to check your functions, the first few values of the result are:

sub_id	min_interval	max_interval	avg_interval
5438	3	270203	90101.333333
5758	4	876015	386447.5
5872	5	20	10.333333333
4485	6	1959428	979717
5324	7	488365	122155
6803	7	1208246	604126.5
5344	8	485179	131585
5379	8	523015	261511.5
5307	8	778900	259638.66666
6983	10	10	10
6789	11	36	20.166666666
...	...	...	...

You will notice that it takes around 5 seconds to complete (perhaps more, depending on your implementation).

4) Given the above query and the definitions of your functions, create an index on **fileset** that will dramatically speed up the query. (With one index, you can make the above query take under 1 second.)

Index Hints:

- You might think that min/max simply can't be made fast in certain circumstances, but it can, if you create the proper index! You will need to find the relevant tidbits in this page of the MySQL 5.5 documentation: http://dev.mysql.com/doc/refman/5.5/en/where-optimizations.html

  Then, try the queries you use in your min/max/avg functions, using a specific **`sub_id`** value, (not as part of a group-by over the entire **`fileset`** table, or else you won't see any benefit), and use the **`EXPLAIN`** command to ensure that your queries will in fact be efficient. (You might want to read about what "Select tables optimized away" means in the **`EXPLAIN`** documentation: http://dev.mysql.com/doc/refman/5.5/en/using-explain.html)

- You should take note that the order of operations is extremely critical in whether the database can use an index or not! For example, when you are trying out your indexes, you will notice a *big* difference in the **`EXPLAIN`** plans for these two queries:

```
SELECT MIN(UNIX_TIMESTAMP(sub_date)) FROM fileset WHERE sub_id = 5344;
SELECT UNIX_TIMESTAMP(MIN(sub_date)) FROM fileset WHERE sub_id = 5344;
```

Part B: Branch Statistics Materialized View (60 points)

Scoring: index=7pts; table = 3pts; view = 4pts; insert = 4pts; triggers = 42pts total (approx. 14pts each).

Here is a simple view definition, reporting various statistics about bank branches with associated accounts:[3]

```
CREATE VIEW branch_account_stats AS
  SELECT branch_name,
         COUNT(*) AS num_accounts,
         SUM(balance) AS total_deposits,
         AVG(balance) AS avg_balance,
         MIN(balance) AS min_balance,
         MAX(balance) AS max_balance
  FROM account GROUP BY branch_name;
```

We would like to implement this as a materialized view, but of course MySQL doesn't have materialized views. Thus, you will need to create a table called **`mv_branch_account_stats`** to hold the materialized results of the view **`branch_account_stats`**, and you will need to add triggers to the **`account`** table that update the materialized results as records change in the account table.

Furthermore, your solution should try to be as efficient as reasonably possible. You must do incremental view maintenance on the materialized results wherever possible. Additionally, your solution will benefit from an index on **`account`**, so you should include this index definition in your solution.

Here are some further guidelines for this problem:

[3] You might note that this view does not include all branches in the bank, only branches with accounts. This is to keep the problem from being too complicated; it's complicated enough as it is…

- Your solution will require a table to hold the materialized data, and also a view that presents it in the above-specified form. The reason for this is the "average" value, which will be discussed shortly. You will benefit from creating a primary key on this table, which will give you several benefits including ensuring you don't represent a particular branch multiple times!

- You will need an initial DML statement to populate the initial state of this materialized view. It should be an **`INSERT` … `SELECT`** statement based on the contents of the **`account`** table.

- You will need to create separate **`INSERT`**, **`UPDATE`**, and **`DELETE`** triggers for managing this view, to ensure that it is always updated properly. They should be after-row triggers on the **`account`** table, since your trigger won't have to modify any data in the rows being altered in the **`account`** table.

- **Don't worry about handling changes to the `branch` table!** Referential integrity constraints will protect the view anyway; accounts referencing a particular branch would need to be deleted before the branch can be removed.

- Keep in mind that your triggers will sometimes need to add a new row to your materialized view, or remove a row from the materialized view:

  If a record is deleted from the **`account`** table, and it is the *last* account for a particular branch, then the delete-trigger needs to remove the summary information for that branch since the view only includes information for branches with at least one account.

  (Also, remember that you are supposed to use after-row triggers, so the **`account`** table will already reflect the deletion when your delete-trigger runs.)

  Similarly, if a record is added to the account table, and it is the *first* account for a particular branch, then the insert-trigger needs to add the summary information for that branch since the view will not yet include information for the branch.

- Additionally, it is possible that an account's branch-name might be changed. This should be treated as if the account were deleted from the old branch, then added to the new branch.

  Hint: you should factor common operations into helper-procedures for your triggers to call, so that you don't have complicated logic spread all over the place.

- Note that you can compute the average from the count and sum values. Therefore, feel free to leave the average out of the table containing the materialized results, and simply provide a definition of the **`branch_account_stats`** view that computes the average as a derived attribute. This is definitely part of the most efficient solution.

Finally, some miscellaneous hints:

- You might also find the MySQL **`GREATEST()`** and **`LEAST()`** functions[4] helpful. (Most databases have functions like these.)

- The **`TRUNCATE TABLE`** command does not run **`ON DELETE`** triggers! Don't expect that **`TRUNCATE TABLE account`** will run your delete triggers; instead, do **`DELETE FROM account`**.

[4] http://dev.mysql.com/doc/refman/5.5/en/comparison-operators.html

- Again, you can make min/max queries fast, if you create the proper index. You will again want to try out your queries with the **`EXPLAIN`** command, using a specific branch name to test with.

Problems:

1) Write the index definition.

2) Write the table definition for the materialized results **`mv_branch_account_stats`, using the column names specified** for the view **`branch_account_stats`**.

3) Write the initial SQL DML statement to populate the materialized view table **`mv_branch_account_stats`**.

4) Write the view definition for **`branch_account_stats`**, **using the column names specified**.

5) Write the trigger (and related procedures) to handle inserts.

6) Write the trigger (and related procedures) to handle deletes.

7) Write the trigger (and related procedures) to handle updates.

Write your answers in the **`make-branch-stats.sql`** file. You can assume that this file will be run after the **`make-banking.sql`** file has been loaded.