Assignment 3: SQL DDL, UDFs, and other TLAs[1]

Keep track of your time spent on this assignment! You will get +3 points for filling out the feedback survey at the end of the assignment.

Submission Notes

Like last time, you should format your submission so that we can automate the testing of your SQL queries. We will provide the template files for you to fill in. When you are finished with your assignment, create a tarball named **`cs121hw3-`*`username`*`.tgz`**, so that we know who the file is from.

As before, if you want to make any comments about your answer, write them as SQL comments after the "**`-- [Problem xx]`**" annotation for that question. If the problem is particularly complex, you are encouraged to give explanatory comments; you may receive partial credit for them if your answer is wrong. Do **NOT** put any code before the first problem annotation!

A Note about Column Names

You might have noticed something about the column names used in class so far; columns with the same meaning (and domain) have the same name in different tables. This is not by accident; in fact it makes it very easy to write queries against databases when the columns are named in this way.

This naming convention is called the **unique-role assumption**: each attribute name has a unique meaning in the database. Following this convention makes schemas very easy to understand, and it also allows the use of natural joins and the **`USING`** clause in your queries. Unfortunately, most schemas do not follow this convention, and they are correspondingly more difficult to understand.

You should always follow this unique-role convention in CS121. I definitely encourage you to follow this convention in any of your other database projects as well.

Part A: Nested and Correlated Queries (18 points)

*Complete this part in the file **`correlated.sql`**. (This section uses the banking schema given to you last week; you may want to reload a clean version of the DB to use, before answering these questions.)*

As discussed in class, SQL's broad support of nested queries gives us a powerful tool for constructing very sophisticated queries, but there is a potential performance issue that lurks within this capability. The database's job is not just to generate a result, but also to generate the result as quickly as possible. Thus, the database query-optimizer must give particular attention to subqueries. This generally means that the database will try to evaluate a nested query once and then reuse the result, and the database will also attempt to apply equivalence rules to further constrain the subquery.

One of the more difficult kinds of subqueries to optimize is called a **correlated subquery**. Here is the example given in class:

[1] Three-Letter Acronyms

```
SELECT customer_name FROM depositor d
WHERE NOT EXISTS (SELECT * FROM borrower b
                  WHERE b.customer_name = d.customer_name);
```

What the database would really like to do is to evaluate the inner query only once, but it can't because the result of the inner query depends on the current tuple being considered by the outer query. This means that the inner query must be evaluated *once for each tuple* that the outer query considers, a very slow process to complete. This kind of evaluation process is called **correlated evaluation**.

The good news is that databases are frequently able to **decorrelate** such subqueries automatically, by transforming them into an equivalent expression that uses a join instead of correlated evaluation. The principle behind the transformation is this: The correlated subquery is computing some dependent value based on each tuple produced by the outer query; can we instead compute these dependent values once, as a batch, and then join them against the outer query? If we can, the query can be decorrelated and evaluation is *very* fast. If this cannot be done, the database is stuck with generating the result using correlated evaluation.

One feature introduced in SQL92 is the ability to use scalar subqueries in the **SELECT** clause. For example, here is a query that counts how many accounts each customer has:

```
SELECT customer_name,
       (SELECT COUNT(*) FROM depositor AS d
        WHERE d.customer_name = c.customer_name) AS num_accounts
FROM customer AS c;
```

Scalar subqueries in the **SELECT** clause are very desirable because they frequently make a query very easy to understand. However, notice that this is again a correlated subquery; in fact, most subqueries embedded in an outer **SELECT** clause will be correlated. Of course, the above query can also be stated as an outer join and an aggregate operation, like this:

```
SELECT customer_name, COUNT(account_number) AS num_accounts
FROM customer NATURAL LEFT JOIN depositor
GROUP BY customer_name;
```

(Note that we must change the argument to **COUNT()**, so that we can still get counts of 0.)

This example is relatively easy to decorrelate, but the following problems are more involved.

Scoring: Parts a-c are 4 points each; d is 6 points. Total: 18 points.

a) Briefly state what this query is computing. Then, create a decorrelated version of the same query.

```
SELECT customer_name,
       (SELECT COUNT(*) FROM borrower b
        WHERE b.customer_name = c.customer_name) AS num_loans
FROM customer c ORDER BY num_loans DESC;
```

b) Briefly state what this query is computing. Then, create a decorrelated version of the same query.

```
SELECT branch_name FROM branch b
WHERE assets < (SELECT SUM(amount) FROM loan l
                WHERE l.branch_name = b.branch_name);
```

c) Using correlated subqueries in the **SELECT** clause, write a SQL query that computes the number of accounts and the number of loans at each branch. The result schema should be (*branch_name*, *num_accounts*, *num_loans*). Order the results by increasing branch name.

Hint: The results should contain three zeros.

d) Create a decorrelated version of the previous query. Make sure the results are the same.

Part B: Auto Insurance Database (15 points)

For this problem, you will need to create and test the SQL DDL commands to create a simple auto insurance database in MySQL:

person(*driver_id*, *name*, *address*)
car(*license*, *model*, *year*)
accident(*report_number*, *date_occurred*, *location*, *description*)
owns(*driver_id*, *license*)
participated(*driver_id*, *license*, *report_number*, *damage_amount*)

Columns that are part of each table's primary key are underlined. For example, in the *owns* table, both *driver_id* and *license* are in the table's primary key.

Dropping and Creating Tables

Write this DDL in a single file named **make-auto.sql**. Frequently, when creating schemas like this, you will want to support running the DDL file against an existing database. However, if you try to create a table that already exists, the database server will report an error. Therefore, you can write your file as follows:

```
DROP TABLE IF EXISTS tbl1;
DROP TABLE IF EXISTS tbl2;
...

CREATE TABLE tbl1 (
    ...
);

...
```

Always drop tables in an order that respects referential integrity. For example, with the tables *account*, *depositor* and *customer*, we know that *depositor* references both *account* and *customer*. Therefore, *depositor* must be dropped before *account* or *customer* can be dropped. In general, the referencing table must always be dropped before the referenced table is dropped.

Additional Specifications

You will need to choose an appropriate type for each column. Details are given below, so that you can make good choices. **Also, make sure to clearly document your table definitions; you will lose points for poorly documented DDL.** (You will understand why, the first time you have to figure out an existing database schema with no docs…)

Your DDL should include all primary key and foreign key specifications. The *owns* and *participated* tables are the referencing relations in the schema, since they specify the relationships between entities in the other tables. *(Note: Cascade specifications are given below!)*

Important Note:

> MySQL requires that foreign keys specify both the table name and the column name, even when the foreign key uses the primary-key column of the referenced table. Unfortunately, if you fail to do this, MySQL will not give you a helpful error; it will be cryptic and not very helpful. You have been warned! ☺

Here are some additional specifications to follow:

- The following columns should allow **`NULL`** values. All other columns should not allow **`NULL`** values.

  car.model
  car.year
  accident.description
  participated.damage_amount

- All *driver_id* values to be stored are exactly 10 characters.

- All car license values are exactly 7 characters.

- Make *report_number* an auto-incrementing integer column in the database. (See the MySQL documentation for details on how to do this. Look for **`AUTO_INCREMENT`** in the **`CREATE TABLE`** page.)

- The *date_occurred* column should be able to store both date and time value for when the accident occurred.

- The *damage_amount* value is a monetary value, so use an appropriate type for this.

- The *location* value will be a nearby address or an intersection. It doesn't need to be more than a few hundred characters, but it will likely vary significantly in size from report to report.

- The *description* field is for an accident report, which would tend to be at least several thousand characters.

Finally, your schema should also include the following cascade options:

- The *owns* table should support both cascaded updates and cascaded deletes, when any referenced relation is modified in the corresponding way.

- The *participated* table should only support cascaded updates, but not cascaded deletes.

You might want to perform some basic testing to ensure that the database behaves appropriately.

Part C: Homework Submission System (67 points)

Complete this part in a single file called ***hwsub.sql****.*

The following questions use the schema and data for a homework management system used for a popular CS course (properly anonymized, of course). The schema includes the following tables:

- **section** (recitation sections)
- **student** (student accounts)
- **assignment** (homework assignments, quizzes, and exams)
- **submission** (graded submissions from students)
- **fileset** (files in each submission)

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.

The schema and data for the homework system are contained in the provided SQL file **make-grades.sql**. Download this file and run it on your database. (The SQL file uses a nonstandard MySQL syntax for importing the data, since using simple **INSERT**s would make the import take several minutes. As it is, it will still take a few seconds to complete. When loading large datasets, you often need to exploit DBMS-specific capabilities for the sake of performance.)

Problem 1: Warm Up Queries

(Scoring: parts a-b are 3 points each; part c is 2 points; parts d-e are 1 point each; part f is 4 points, and part g is 2 points. Total: 16 points)

Once the data is imported, try these queries to get a feel for what the schema and data are like:

a) Compute what would be a "perfect score" in the course, by adding up the perfect score values for all assignments, quizzes and exams.

b) Write a query that lists every section's name, and how many students are in that section.

c) Create a view named **totalscores**, which computes each student's total score over all assignments in the course. The view's result should contain each student's username, and the total score for that student. Note that **submission** records have a **graded** flag that will be set to 1 if the grade-value is valid; you should use this flag in computing your results. Make sure to give the total-score value a useful name!

Note: It is possible for a student to submit work for a single assignment multiple times, but there will be exactly one submission record for every (student, assignment) pair, and that single submission record will contain the actual grade the student received. Thus, you really don't have to worry about multiple submissions from a given student for a single assignment on this problem. The view-definition is in fact quite straightforward.

d) This course happens to be pass/fail, and a passing grade is a total score of 40 or higher. Using the **`totalscores`** view, a view called **`passing`** which lists the usernames and scores of all students that are passing.

(Hint: MySQL may not allow a view to refer to another view, so if that is the case, feel free to define your passing and failing views independent from any other view.)

e) Similarly, create a view called **`failing`**, which lists the usernames and scores of all students that are failing.

Here is a more challenging problem. **Include the results in your submission as a comment,** so that we know you have run the queries:

f) Write a SQL query that lists the usernames of all students that failed to submit work for at least one lab assignment, but still managed to pass the course. (Lab assignments have a short-name starting with the characters "lab" in the **`assignment`** table, so you will have a condition similar to "**`shortname LIKE 'lab%'`**" in your SQL.) Remember that you must use the **`fileset`** relation to tell whether a student has submitted work or not; simply looking for a row in the **`submission`** relation is not sufficient.

*Hint: Use the **`passing`** view to tell if a student passed the course. Don't Repeat Yourself.*

g) Finally, for kicks, update and rerun this query to show all students that failed to submit either the midterm or the final, yet still managed to pass the course. **Include both the query and the result of running it in your submission.**

Problem 2: Dates and Times *(5 points per part; 15 points total)*

Here are some queries that involve date and time values:

a) Write an SQL query that reports the usernames of all students that submitted work for the midterm after the due date. Note that the actual submission times are contained within the **`fileset`** relation. Make sure that each username is reported only once.

b) Write an SQL query that reports, for each hour in the day, how many lab assignments (assignments whose short-names start with **`'lab'`**) are submitted in that hour. Again, submission times are stored in the **`fileset`** relation. (You can use **`EXTRACT`** expressions in a **`GROUP BY`** clause, if necessary. In general, you can group by arbitrary expressions; you are not limited to only grouping on specific columns.)

Your result's schema will be something like (*hour*, *num_submits*), and the hour value should be in the range [0, 23].

c) Write an SQL query that reports the total number of final exams that were submitted in the 30 minutes before the final exam due date. Don't hard-code the final's due-date into the SQL command; fetch it from the database as part of the query, using the final exam's short-name "final".

Problem 3: Schema and Data Migration

(Scoring: parts a-c are 4 points each. Total: 12 points)

The next version of the student grades database will be slightly different from the current version, so a series of schema-alteration and DML commands must be devised to migrate the existing course database to the new schema.

Starting with a clean version of the existing grades database, write the DDL and DML commands for altering the existing database. That is, you are not simply editing the original `.sql` file; you are writing **`ALTER TABLE`** commands and other necessary commands for changing the schema that is already in the database.

Of course, you should test your commands to make sure they work.

a) Add a column named "**`email`**" to the student table, which is a **`VARCHAR(200)`**. When you create the new column, allow **`NULL`** values.

Populate the new email column by concatenating the student's username to "@school.edu". (You can use the MySQL **`CONCAT()`** function for this, or the ANSI SQL-standard string-concatenation operator **`||`**, double-pipe.)

Finally, impose a **`NOT NULL`** constraint on the column. (See the **`ALTER TABLE ... CHANGE COLUMN`** syntax.)

b) Add a column named "**`submit_files`**" to the assignment table, which is a **`BOOLEAN`** column. Make the default value **`TRUE`**.

Write an **`UPDATE`** command that sets all "daily quiz" assignments (assignments whose **`shortname`** value starts with "dq") to have *submit_files* = **`FALSE`**.

c) Create a new table "**`gradescheme`**" with the following columns:
- *scheme_id* (integer, primary key)
- *scheme_desc* (variable-length character field, max of 100 characters, not null)

Write the commands to insert the following rows into the new table:

(0, "Lab assignment with min-grading.")
(1, "Daily quiz.")
(2, "Midterm or final exam.")

In the **`assignment`** table, rename the **`gradescheme`** column to be named "**`scheme_id`**", but leave it as an **`INTEGER`** column. (Make sure it does not allow **`NULL`** values.)

Finally, add a foreign key constraint from **`assignment.scheme_id`** to the new **`gradescheme.scheme_id`** column. (You don't have to give the foreign key constraint a name if you don't want to.) The values you inserted into the **`gradescheme`** table should match what is already in the **`assignment`** table, so you shouldn't get any errors when you run this command.

Problem 4: Creating User-Defined Functions *(part a: 5 points, part b: 10 points)*

MySQL has a wide variety of helpful date and time functions[2]. You will find them very helpful for both this and the next problem. Also, you may want to refer to the MySQL stored routine syntax[3] if you run into any issues.

To make sure your answers for this and the next exercise actually work, you can use the provided **`test-dates.sql`** file. It includes a simple table of date values, and what the **`is_weekend`** and **`is_holiday`** functions ought to output for those dates. You can even write a single SQL statement to use this data with your functions to immediately identify inputs that cause your functions to fail.

a) Write a user-defined function named **`is_weekend`** that takes a **`DATE`** value and returns a **`BOOLEAN`**. The function should return **`TRUE`** if the specified date falls on a weekend (Saturday or Sunday), or **`FALSE`** if the date falls on a weekday (Monday through Friday). Your function will be declared something like this:

```
-- Set the "end of statement" character to ! so that
-- semicolons in the function body won't confuse MySQL.
DELIMITER !

-- Given a date value, returns TRUE if it is a weekend,
-- or FALSE if it is a weekday.
CREATE FUNCTION is_weekend(d DATE) RETURNS BOOLEAN
BEGIN

    -- TODO:  YOU FILL IN THE REST!

END !

-- Back to the standard SQL delimiter
DELIMITER ;
```

Note the **`DELIMITER`** commands: they are necessary because the commands within stored routines usually require semicolons, but MySQL's parser isn't smart enough to figure out when the entire procedure ends. Thus, the standard approach with MySQL is to change the delimiter to something else, like "!", and then switch it back to ";" at the end of the "create stored routine" operation.

b) Write another user-defined function named **`is_holiday`**, that takes a **`DATE`** and returns a **`VARCHAR(20)`** describing what holiday the specified date falls on. Here are the holidays your function should recognize, along with the string that should be returned:

- January 1: "New Year's Day" (note the apostrophe in the result; you can escape the apostrophe with a backslash, **`\'`**)
- The last Monday in May: "Memorial Day"
- July 4: "Independence Day"
- The first Monday in September: "Labor Day"
- The fourth Thursday in November: "Thanksgiving"

If the specified date is none of the above holidays, the function should return **`NULL`**.

---

[2] http://dev.mysql.com/doc/refman/5.5/en/date-and-time-functions.html
[3] http://dev.mysql.com/doc/refman/5.5/en/stored-programs-defining.html

Make sure to write complete and concise documentation for your function!

Here are some hints:

- Use a local variable to hold the function's result. Set it to **`NULL`** initially. Then, depending on various tests, set the value to the appropriate string literal. Finally, return the value of this local variable as the function's result.

  (Or, you can just return string literals as well, if you don't want to fuss around with any variables…)

- A good approach would be to store certain useful values into local variables, before the task of determining what holiday the date falls on. For example, you might extract the month, the day of the week, the day of the month, and so forth. Having these values available makes it much easier to write these tests.

- When you have a requirement like "the first Monday of the month" or "the last Monday of the month", it is helpful to note that such a day must fall within a specific range of dates. For example, Martin Luther King Jr. Day falls on the third Monday of January. Thus, all of the following conditions must be true:
  - The passed-in date must be in January.
  - The passed-in date must be a Monday.
  - The day of the month must be in the range [15, 21], inclusive. **The third Monday of the month must always fall within these dates.** (Convince yourself of this if you are not sure.) A **`BETWEEN`** clause can easily verify such a condition, and you are able to (and encouraged to!) use **`BETWEEN`** in the condition for SQL conditional expressions.

Again, you can use the **`test-dates.sql`** file to verify that your **`is_holiday`** function works properly.

Problem 5: Using your UDFs in Queries *(part a: 4 points, part b: 5 points)*

Once you have the previous exercise finished, write these two simple queries against the grades database to see how well your user-defined functions work. Both of these queries will only involve the **`sub_date`** column of the **`fileset`** table.

NOTE: Your functions are defined to take **`DATE`** values, as they should, but the **`fileset.sub_date`** column is a **`TIMESTAMP`**. Thus, you may get "truncation" errors from the database. If so, you can simply use the **`DATE()`** function to convert the timestamps into **`DATE`** values, like this: **`is_weekend(DATE(sub_date))`**

a) Write a query that reports how many filesets were submitted on the various holidays recognized by your **`is_holiday()`** function. Your query will use your function in the **`SELECT`** clause, and group by the result. Give all columns reasonable names. If you did things correctly, you should see 58 Thanksgiving submissions, and the rest of the submissions having **`NULL`** as the holiday.

b) Write another query that reports how many filesets were submitted on a weekend, and how many were not submitted on a weekend. Again, this is a single query containing both results; you will be grouping on the result of your **`is_weekend()`** function.

Since **`is_weekend()`** just produces a single **`BOOLEAN`** value, use the SQL **`CASE`** syntax[4] to generate a string **`'weekend'`** when **`is_weekend()`** returns **`TRUE`**, and **`'weekday'`** when **`is_weekend()`** returns **`FALSE`**. (If you give the entire **`CASE`** expression a name in your **`SELECT`** clause, you should be able to group on it with no problems.)

If everything is written correctly, you should see 371 weekend submissions and 2386 weekday submissions.

Feedback Survey (+3 bonus points)

Complete the feedback survey for this assignment on the course website, and 3 points will be added to your score (max of 100/100).

[4] http://dev.mysql.com/doc/refman/5.5/en/control-flow-functions.html