# Assignment 3: Correlated Queries and SQL DDL

**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.**

**Due:** Sunday, Jan. 31st 2AM PST

## Submission Notes

As with Assignment 2, you will submit your solutions to CodePost for this assignment. There will be 3 different CodePost assignments to submit, corresponding to the 3 parts in A3: Part A, A3: Part B, and A3: Part C. You can submit the file for each part on https://codepost.io/. Late submissions are not accepted, and will need to be submitted as a rework.

Like last time, you should format your submission so that we can automate the testing of your SQL queries. Test your SQL with the **`check.py`** from Assignment 2 before submitting.

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! There will be a deduction on submissions that do not follow this requirement.

## 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.** We definitely encourage you to follow this convention in any of your other database projects as well.

# Part A: Nested and Correlated Queries (20 points)

*Complete this part in the file **`correlated.sql`**. (This section uses the banking schema given to you in Assignment 2 - 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:

```
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. (in fact, MySQL's decorrelation capabilities have improved quite a bit by version 8.0).

The principle behind a decorrelation 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-d are 5 points each. Total: 20 points.*

a) Briefly state what this query computes. 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 computes. 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.**

**(Optional) Measuring performance in MySQL**

So, what are the performance advantages of decorrelation in MySQL? In Lecture 5, Melissa showed how to use **SHOW PROFILE** in MySQL to see the runtime results of different queries. There are many ways to analyze performance, but this is one of the simplest (we'll explore more later in the course). An example is shown below - **profiling=1** sets the session variable **profiling** to 1 (true).

```
mysql> profiling=1;

mysql>  SELECT * FROM account AS a1, account AS a2, account AS a3
… 216000 rows in set (0.09 sec)
mysql> SELECT * FROM account;
… 60 rows in set (0.00 sec)
mysql> SHOW PROFILES;
+----------+------------+-------------------------------------------------------------+
| Query_ID | Duration   | Query                                                       |
+----------+------------+-------------------------------------------------------------+
|        1 | 0.08265825 | SELECT * FROM account AS a1, account AS a2, account AS a3 |
|        2 | 0.00030525 | SELECT * FROM account                                       |
+----------+------------+-------------------------------------------------------------+
```

We recommend you exploring this feature to compare the performance when writing different queries which compute the same thing.

# Part B: Auto Insurance DDL (30 points)

In Parts A and B, you will get practice writing DDL commands for two example databases, each which will use a variety of different constraints appropriate for the data being represented and supporting possible changes (updates and deletes) across tables.

The first database is an example auto insurance database, which does not have any corresponding datasets, but is a good exercise on using different attribute domains and constraints.

The second database is a Spotify database, which is designed in a similar process to the breakdown of the AirBNB dataset demonstrated in lecture.

## Dropping and Creating Tables

You will write two files: **`setup-auto.sql`** and **`setup-spotify.sql`**, each which should be used in their own databases (e.g. **`autodb`** and **`spotifydb`**).

Frequently, when creating schemas in setup files, 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 in each problem, 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.** For example, in the auto insurance database, 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! ☺

**1. Auto Insurance Database DDL (22 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.

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 under **`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 <u>both</u> cascaded updates and cascaded deletes, when any referenced relation is modified in the corresponding way.
- The *participated* table should only support cascaded updates, but <u>not</u> cascaded deletes.

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

**2. Auto Insurance Modification Queries (8 points)**

When writing DDL commands to create tables in a database, it is very important that you test them with example data (creating your own test data if you do not have data available yet).

Write the following queries in a file called **`test-auto.sql`** to insert data into your tables, making sure that you use appropriate values for the table constraints.

Insert at least three records into each of the five tables. For each referencing table, make sure that the data insert appropriate references the other table(s). Also make sure that your **`INSERT`** statements are in the correct order so that referenced rows exist.

For full credit, you should:

- Insert at least one row with a **`NULL`** value to represent some unknown information in that row of data. **Do not** have any INSERT statements that explicitly include **`NULL`** as a value (hint: use the **`INSERT (<values>) INTO table(<attr>`**) to specify the values inserted - any missing values for attributes that can be null or have default values set will be set to the appropriate default values). You can also use the following short-hand to insert multiple rows in a single INSERT statement:

```
INSERT INTO <table> (<attrs>)
VALUES
    (<value_list_1>),
    (<value_list_1>),
    ...
    (<value_list_n>);
```

- Write one **`UPDATE`** statement for both *person* and *car* tables to test your **`CASCADE`** constraint of *owns*.
- Write one **`DELETE`** statement to remove a row from the *car* table (e.g. in the case that a car is totaled and sent to the scrapyard) to test that the *participates* table *does not* support cascaded deletes.

Your answers for this problem should include three Problem comments:

```
-- [Problem 2a]

15 INSERT statements (you may use multi-row INSERT syntax)

-- [Problem 2b]

2 UPDATE statements

-- [Problem 2c]

1 DELETE statement
```

# Part C: Spotify Playlist DDL and Data Importing (20 Points)

Next, you will write the DDL commands to set-up the Spotify Playlist database in **`setup-spotify.sql`**. This database breakdown includes various information about song tracks associated with different playlists created by users, including information about artists and albums. After writing the DDL commands, you will use an example .csv dataset (**`playlist_data.csv`**) which contains information you can use to populate your tables. Note that this database could be improved with a *user*s table which the *playlist* references by the *added_by* username, but to keep things simple without including real Spotify user information, we are omitting that in this exercise.

### 1. Spotify Playlist DDL (12 points)

The table schemas you will implement are as follows:

*track(track_uri, track_name, artist_uri, album_uri, playlist_uri, duration_ms, preview_url, added_at)*
*artist(artist_uri, artist_name)*
*album(album_uri, album_name, release_date)*
*playlist(playlist_uri, playlist_name, added_by)*

Again, columns that are part of each table's primary key are underlined. *track* is the only referencing relation (but references multiple tables).

Note: In the example dataset, you'll see columns called Track URI, Artist URI, and Playlist URI. The Spotify Web API (which this data is pulled from) provides these as unique identifiers for the corresponding entities (in fact, you can append a URI to https://open.spotify.com/track/<track_uri> in your browser to visit the corresponding track, and similarly for /artist and /playlist). We could add an auto-incremented ID number for each table, but since these are already unique and our data represents Spotify Playlist data, we will use these as our primary keys.

#### Table Specifications

Similar to the Auto Insurance Database, you will need to choose an appropriate type for each column and your DDL should include all primary key and foreign key specifications - additional details are given below.,

- The following columns should allow **`NULL`** values. All other columns should not allow **`NULL`** values.
  *track.preview_url*
  *playlist.added_by*

- All track, artist, album, and playlist names may be up to 250 characters.

- Any date columns should be stored as date values (without times), except *added_at* should include both date and time.

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

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

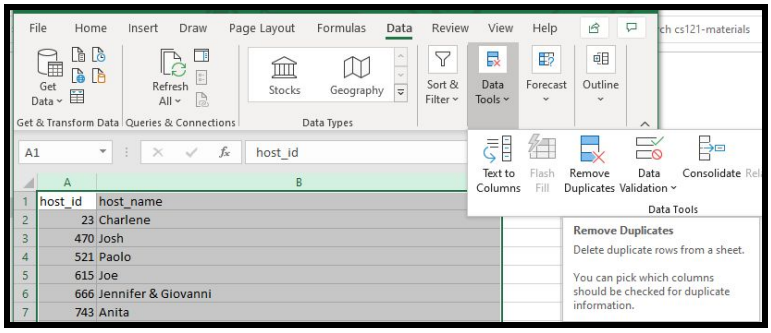
## 2. Importing CSV Data into a Relational Database (8 points)

In this last part, you will test your finished DDL and get practice importing a CSV dataset into your MySQL database (you can then write some queries you are interested in, but we are moving on from SELECT queries in this assignment). In this part, you will submit your four CSV files created from the provided **`playlist_data.csv`** file. Alternatively, you have the option to get your own Spotify playlist datasets from http://watsonbox.github.io/exportify/ which is a useful tool to get CSV data from your Spotify playlists and, with a SQL database, write some queries on your own music interests (optional).

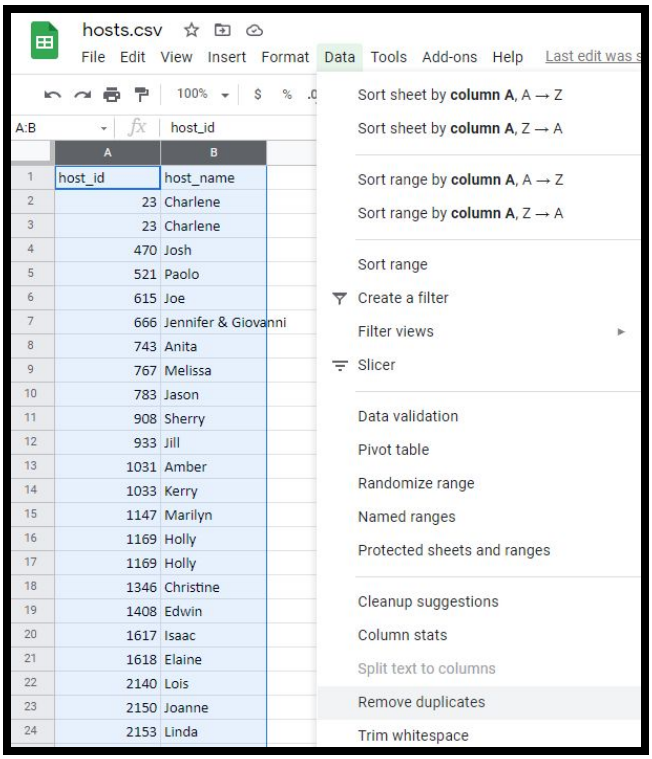
First, you will need to do a small amount of cleanup to the CSV file to import into your tables. This is very common when importing real datasets into a database, especially when working with a flat source file that will need to be broken apart to better represent the relational model. Luckily, it's quite a bit cleaner than a lot of other publicly-available datasets, so you can do this with a simple spreadsheet editor (similar to the approach demonstrated in lecture). You are also free to write a script to do this for you if you would like.

Specifically, you will need to:

- Remove any columns that are not represented in your tables
- Split the CSV file into 4 CSV files (one for each table)
- For any table that has a referencing relation, make sure to keep the foreign key column(s) in the table when you copy them as primary keys with associated data in another file (similar to how the AirBNB listings.csv dataset was modified to remove host-specific data into a hosts.csv file, but kept the host_id foreign key).
- You may have some tables that will have duplicate primary keys as a result (and you should understand why). Make sure to remove these duplicates.
    - In Excel, you can remove duplicates by highlighting the columns, going to `Data > Data Tools > Remove Duplicates`



- If you prefer to use Google Sheets, you can remove duplicates similarly going to `Data > Remove Duplicates`




When you have broken the original CSV into 4 files, each should have exactly the same column counts and domains as specified in your DDL.

Next, you'll be ready to import! There are a few ways to do this, depending on whether you prefer to use the command line or a tool like phpMyAdmin. You can find instructions for both ways on this page.

For the command line, you will need to save your .csv files to be imported in the location where MySQL allows files to be loaded. Note that you may run into an error that MySQL server is running with the `--secure-file-priv` option. You can either try editing your MySQL .conf file to turn off this option, or move your .csv files to the directory specified by secure-file-priv, which can be found with `SHOW VARIABLES LIKE "secure_file_priv";`

For example:

```
mysql> SHOW VARIABLES LIKE "secure_file_priv";

+------------------+-----------------------+
| Variable_name    | Value                 |
+------------------+-----------------------+
| secure_file_priv | /var/lib/mysql-files/ |
+------------------+-----------------------+
1 row in set (0.00 sec)

mysql> LOAD DATA INFILE '/var/lib/mysql-files/seattle_hosts.csv' INTO TABLE hosts
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED BY '\n' IGNORE 1 ROWS;
Query OK, 2372 rows affected (0.01 sec)
Records: 2372  Deleted: 0  Skipped: 0  Warnings: 0
```

Note that the .csv files have to be moved to this folder before loading, which you can do with:

```
$cp *.csv /var/lib/mysql-files/
```

in the bash terminal before entering the mysql console again (assuming you are in the same directory as your .csv files).

Once you have imported your files, you should be able to see your four tables populated (which you can test with simple **`SELECT`** statements).

For this problem (C.2), you will submit the four .csv files to CodePost: **`tracks.csv`**, **`artists.csv`**, **`albums.csv`**, and **`playlists.csv`**. For full credit, they must be formatted such that they could be imported into your database without errors.

## Feedback Survey (+3 bonus points)

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