Assignment 3: Correlated Queries and SQL DDL

Due: Saturday, Feb. 3rd 11:30PM 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 without approved extenuating circumstances, 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. You can find starter files, including **`check.py`** and a template for Part A in the a3-starter.zip posted on Canvas. Part B/C DDL files do not follow the same template structure, so you'll need to start these from scratch.

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. If your code unexpectedly fails the tests on CodePost, double-check your conventions here and also **make sure you don't have a ; (semicolon) character in your comments**.

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) OFTEN 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 (think about **`NATURAL JOIN`** and **`USING`**!)

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.

Unless otherwise specified, 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.

Reminders for Good MySQL Conventions

Part of your grade will depend on proper use of MySQL conventions, including the following:

- Use upper-casing conventions for keywords (e.g. **`SELECT`**, **`FROM`**, **`CREATE TABLE`**, **`INT`**)
- Do not use **`NOT NULL`** for primary keys (by default, a primary key cannot be **`NULL`**, so adding this constraint would be redundant)
- Use *consistent* indentation. A good convention is to indent subqueries, and otherwise keep keywords in the same indentation as the **`SELECT-FROM-WHERE`** level they belong in. You can refer to the lecture slides and code for some good examples.

- Allocate some time to simplify queries if needed; in A2, we've been seeing quite a few solutions that are overly-complicated, especially when overusing subqueries or joins.
- Do not use **`WITH`** in this assignment or temporary tables (**`CREATE TEMPORARY TABLE`**) you got practice with temporary tables in A2, but solutions in A3 can be simpler without it.
- Do not create any unspecified tables (persistent or temporary).
- Do not have any blank lines *within* a query.
- Do not use tab characters, and use 2- or 4-space indentation consistently; you can configure VSCode to use spaces instead of tabs if your default settings are tabs, and can double-check your conventions with the Find/Replace feature in VSCode to replace any \t with spaces (you may need to use the regex option for \t)
- **Do not include `CREATE DATABASE/USE` in your files. This may mess with the grading scripts.**

Part A: Nested and Correlated Queries (24 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. It can help to reason about this when thinking about the equivalent relational algebra expression. 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; you can compare MySQL's overview of optimizing subqueries between 5.7 and 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. In the next few exercises, you will get practice decorrelating subqueries, which, while some recent DBMS versions may automatically optimize, is an important learning objective of this unit.

Scoring: Parts a-d are 6 points each. Total: 24 points.

a) As a 1-2 sentence comment, briefly state what this query computes; for full credit, it must be clear that you understand what the 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 loan_count
FROM customer c ORDER BY loan_count DESC;
```

b) Again, 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*, *loan_counts, num_accounts*). Order the results by increasing branch name.

   <u>Hint</u>: *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 (no points)

So, what are the performance advantages of decorrelation in MySQL? In Lecture 6, El 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 encourage you to explore this feature to compare the performance when writing different queries which compute the same thing.

Part B: Auto Insurance DDL (44 points)

In Parts B and C, 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 setup files: **setup-auto.sql** (for Part B) and **setup-spotify.sql** (for Part C), 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 documentation/rationale…)

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 Notes:

- 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! ☺
- For full credit, each of your tables must include brief comments above the **`CREATE TABLE`** line to describe the table. You should also use brief comments to specify attribute domains in your **`CREATE TABLE`** definitions (above the attribute) when the DDL isn't self-describing (e.g. you don't need to include a comment for *driver_id*, but you should have a brief comment describing the *model* attribute; for example, "The model of the gar, e.g. "Toyota" or Lexus", may be NULL" is a useful comment to specify the *model* domain in the *car* table (and you may use this example in your solution). Refer to the lecture code for more examples of documenting DDL. You can refer to the DDL from lecture's AirBNB demos for some good examples.
- For foreign key constraints, do not use the inline **`REFERENCES`** syntax; this is not compatible with MySQL. Instead, use the **`FOREIGN KEY (<attrs>) REFERENCES other_table(<attrs>)`** syntax shown in class, following a table's attribute definitions.

1. Auto Insurance Database DDL (34 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*, *summary*)
owns(*driver_id*, *license*)
participated(*driver_id*, *license*, *report_number*, *damage_amount*)

By convention, 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.

car.model
car.year
accident.summary
participated.damage_amount

All other columns should not allow **`NULL`** values.

- 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`** overview)
- The *date_occurred* column should be able to store both date and time value for when the accident occurred (use the **`TIMESTAMP`** datatype for this)
- The *damage_amount* value is a monetary value (we're using USD currency), so use an appropriate type for this. You can assume the amount is less than $1,000,000, but may be just under this value (ouch).
- 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 *summary* 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 (you can find more information on **`CASCADE`** in Lecture 8 slides, posted in advance).
- 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. Do not use cascade constraints where they don't make sense.

2. Auto Insurance Modification Queries (10 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 inserted appropriately 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_2>),
    ...
    (<value_list_n>);
```

- Write one **`UPDATE`** statement for each *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. This statement *should* cause an error assuming you have written your DDL correctly. **For full credit, make sure your final answer is commented with -- so that your `test-auto.sql` does not raise an error when tests import it, but such that removing -- will otherwise be valid *syntax* for the `DELETE` statement.**

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, as a comment
```

X. (Optional) Database Design Tradeoffs (up to +5 points, max 100)

You can earn up to 5 bonus points (set capped at 100/100) exploring design trade-offs with the schemas specified above. In **`partb-tradeoffs.txt`**, provide 3-5 sentence answers for each of the following. For the full 5 additional points, your answers must be clearly justified; answers that observe conflicting design tradeoffs are encouraged. You may pull from your own experiences as a developer or a user. We're not looking for expert database administration answers here, just that

you're able to draw what you know and what trade-offs have been discussed in class to reinforce design and implementation thinking at the DDL level.

a) What is one application that the auto insurance DDL could support? In your answer, provide an example query (either in English or in SQL; if both, you'll be eligible for more credit) and what schemas/attributes would be relevant to an example use case. For example, an application that the AirBNB DDL from lecture could support would be a website listing all properties in a searched neighborhood, with filter options to search by minimum and maximum nights.
b) In the real world, databases sometimes are required to change to keep up with a variety of unexpected variables. For example, COVID required many companies and institutions to adapt to a virtual or hybrid model (e.g. K-12 students taking classes online). Another example would be companies offering online tools or applications on different IOT devices, such as tablets, smart watches, and VR. Even more common would be companies extending to international customers. The design phase is essential to think about robustness of your schemas when using the relational model, but sometimes it's necessary to refactor.

   Identify 1 scenario that, if taken priority, would require an adjustment to the auto insurance schemas. Your answer should clearly relate to the DDL requirements above, and why the current schemas would not support that scenario.

   Then, identify 1-2 changes that could be made to the schemas (either to the attribute level, such as a different datatype or constraint, or the table schema level, such as adding/updating tables and any key constraints). For example, El discussed how the **`name`** attribute in the university database would not support students/faculty added later who have more than 20 characters in their name, even though it does attempt to support individuals who have 0 or 2+ last names. One possible change would be to make the **`VARCHAR`** size limit longer (e.g. 100 characters) or to break down name into **`first_name`** and **`surname`**, allowing for surname to be **`NULL`** (though first_name would still be **`NOT NULL`**).

Part C: Spotify Playlist DDL and Data Importing (32 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`**) on Canvas which contains information you can use to populate your tables. Note that this database could be improved with a *user*s table which the *playlists* references by the *added_by* username, but to keep things simple without including real Spotify user information, we are omitting that in this exercise.

Note: If you are having issues with character encodings, make sure the CSV files are saved as CSV UTF-8 (under Save As…). You can alternatively change encoded characters with Excel/Python. The provided CSV file is already in UTF-8 encoding.

1. Spotify Playlist DDL (22 points)

The table schemas you will implement are as follows:

tracks(track_uri, playlist_uri, track_name, artist_uri, album_uri, duration_ms, preview_url, popularity, added_at, added_by)
artists(artist_uri, artist_name)
albums(album_uri, album_name, release_date)
playlists(playlist_uri, playlist_name)

Again, columns that are part of each table's primary key are underlined. *tracks* is the only referencing relation (but references multiple tables). Also note that *tracks* has a 2-attribute primary key, since a unique track may be in more than one playlist (but we are requiring no duplicate tracks in a single playlist).

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 (e.g. if a track doesn't have a preview url)
  tracks.preview_url
  tracks.added_by
  All other columns should not allow **`NULL`** values.

- All track, artist, album, and playlist uri (not 'url') are exactly 22 characters (a Spotify standardization for identifiers)

- 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 (hint: use **`TIMESTAMP`**, not **`DATETIME`** to correctly represent the formats in the given .csv file).

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

- The *tracks* 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 (10 points)

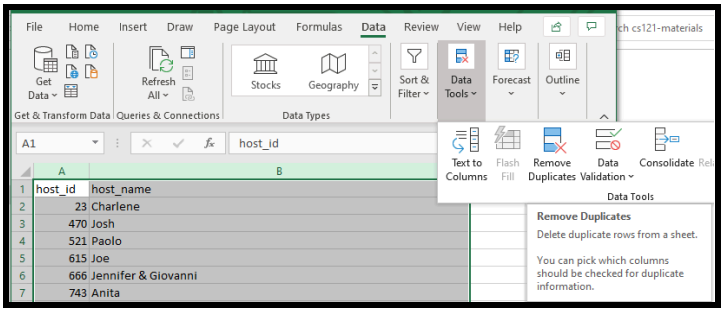
In this part, you will test your finished DDL and get practice importing a CSV dataset into your MySQL database (you can then write some queries you are interested in, but we are moving on from

`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). Note that this public tool is known to have some issues depending on your browser cookie settings, but clearing cookies may solve any problems you'd run into.

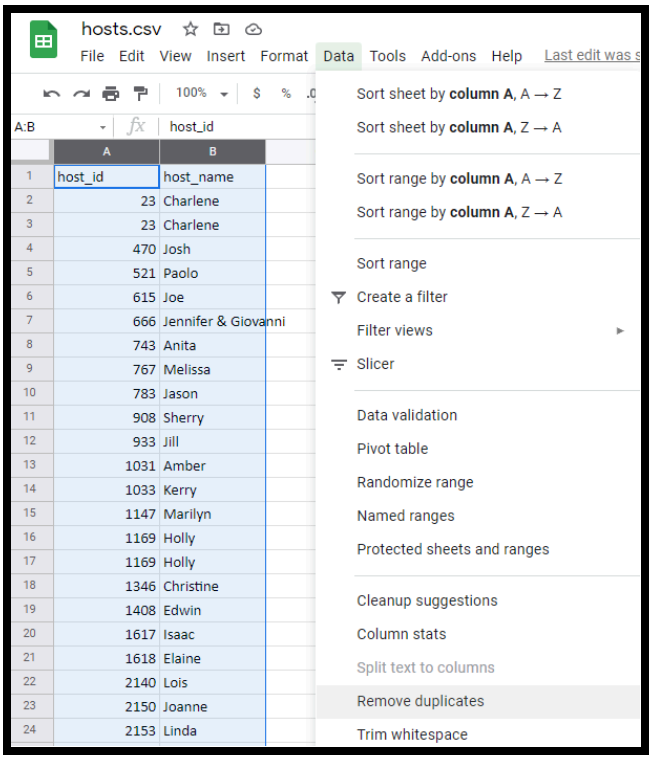
First, you will need to do a small amount of cleanup to the CSV file to import into your tables. This is very common when importing real datasets into a database, especially when working with a flat source file that will need to be broken apart to better represent the relational model. Luckily, it's 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). We've also already done some of the more tedious preliminary cleanup, such as standardizing some character encodings and removing any " characters students might run into problems with depending on their MySQL installation settings. You are also free to write a script to do this for you if you would like. You should not remove any data, but we won't be too strict on changing character encodings/empty column settings if needed (there are ways to configure MySQL to support these, but it varies and we don't want students to spend too much time in this exercise on configurations). Part C is designed for you to be able to complete all in Google Sheets or a similar application prior to downloading as .csv files to import with load-spotify.sql.

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). Again, make sure that these are saved as CSV UTF-8 files (default for Google Sheets).
- 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 **`seattle_listings.csv`** dataset was modified to remove host-specific data into a hosts.csv file, but kept the **`host_id`** foreign key). Use the same requirements for foreign key constraint syntax as in Part B.
- 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`

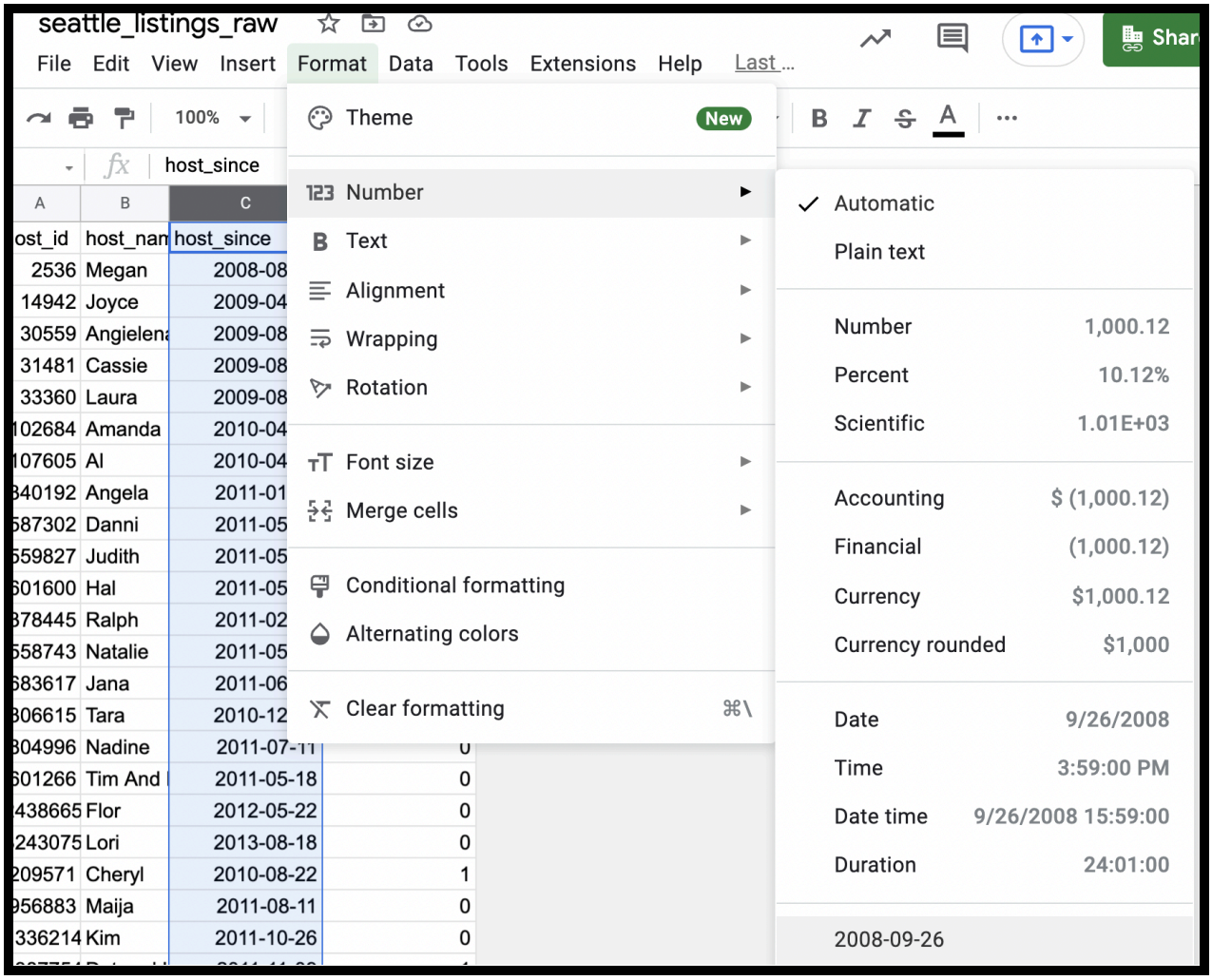


- In Google Sheets (recommended), you can remove duplicates going to `Data > Remove Duplicates`



Be careful with the tracks.csv file; you'll want to select the checkboxes for both the Track URI and Playlist URI so that duplicates are defined as any rows matching on both of these columns. 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.

Important note about dates in Excel/Google Sheets: These programs tend to mangle date values, such as changing dashed format to slash format. Format for correct dates accepted in MySQL should be YYYY-mm-dd and for added_by should be YYYY-mm-dd hh:mm:ss. Converting these in Google Sheets was demonstrated in Lecture 8, but you can reformat before downloading the CSV files as follows:



Important Note about UTF-8 Encodings: MySQL will also expect imported CSV files in UTF-8 encoding. Google Sheets will automatically download CSV files in UTF-8, but if you use another editor like Excel, you may need to specify this encoding in Advanced Options when exporting as CSV.

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, we recommend using the local file-loading approach shown in Lectures 7-8 (you can find a **`load-spotify.sql`** script in the **`a3-starter.zip`** to run after **`setup-spotify.sql`**; otherwise, use CSV importing with phpMyAdmin). For example, assuming you started the mysql console in the same directory where you have your **`artists.csv`** file, you can load it into your *artists* table as follows (and do the same thing for your other three tables):

```
LOAD DATA LOCAL INFILE 'artists.csv' INTO TABLE artists
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED BY '\n' IGNORE 1
ROWS;
```

On Windows, you may need to change `'\n'` to `'\r\n'`.

As noted in the provided **load-spotify.sql** comments, if you run into any permissions issues with using **LOCAL**, you can set the flag to allow it with:

```
SET GLOBAL local_infile = 'ON';
```

If you see any warnings when importing (either in the command line or in phpMyAdmin/MySQL Workbench), make sure to look into these carefully; these should still be manually verified that the row does in fact lack values that are allowed to be missing. Warnings when importing are usually bad, except for "row doesn't have enough data" warnings. You can usually fix the latter by making sure your .csv file has an empty new line at the end.

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.

Here is an example output from start to finish in the command line (similar output in phpMyAdmin). Make sure there are no warnings, or any deleted/skipped rows! If you see any warnings, you can use **SHOW WARNINGS** which will report any warnings from the last SQL command (common ones you might run into are duplicate rows that should have handled as described above, or key constraint violations):

```
mysql> source setup-spotify.sql;
Query OK, 0 rows affected (0.00 sec)
...
Query OK, 0 rows affected (0.03 sec)
mysql> source load-spotify.sql;
Query OK, 1716 rows affected (0.04 sec)
Records: 1716  Deleted: 0  Skipped: 0  Warnings: 0

Query OK, 3874 rows affected (0.04 sec)
Records: 3874  Deleted: 0  Skipped: 0  Warnings: 0

Query OK, 6 rows affected (0.00 sec)
Records: 6  Deleted: 0  Skipped: 0  Warnings: 0

Query OK, 5953 rows affected (0.19 sec)
Records: 5953  Deleted: 0  Skipped: 0  Warnings: 0
```

And as a reminder, your setup-spotify.sql should have the 4 DROP TABLE statements at the top so that the database is reset when sourcing the file (without warnings). Similar to Part B's setup-auto.sql, be careful with the order of these statements so that referential integrity (foreign key) constraints are satisfied.

You can confirm your ordering is correct after running the following right after the above command:

```
mysql> source setup-spotify.sql;
Query OK, 0 rows affected (0.03 sec)
```

```
Query OK, 0 rows affected (0.01 sec)
...
Query OK, 0 rows affected (0.01 sec)
mysql> SHOW TABLES;
+---------------------+
| Tables_in_spotifydb |
+---------------------+
| albums              |
| artists             |
| playlists           |
| tracks              |
+---------------------+
4 rows in set (0.00 sec)

mysql> SELECT COUNT(*) FROM tracks;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.01 sec)
```