CS 121 Final Project Specification

Due: March 18th, 11:59PM PST

Overview: This is the Final Project specification, which includes a complete summary of requirements for your Final Project, some of which were introduced in the Final Project Proposal (along with application ideas). The Final Project requirements cover everything taught over the term, similar to previous Final Exams in CS 121, but provides you freedom to choose an application of interest and get practice integrating SQL with a Python application program. You will submit **all of your files** to CodePost, including the written portion which you can find a template for here. We strongly encourage partner work in this Final Project, and only one of you needs to submit with both of your names and emails at the top of **`app.py`**, **`reflection.pdf`**, and the README.

At minimum, you should be submitting the following (**note that the list may seem long, but they will naturally overlap as you work through the project from start to finish and are chosen to be difficult to <u>not meet</u> when designing and implementing small database application**):

- **<u>Parts A, C, G, L:</u> `reflection.pdf`** - Reflection component for your Final Project, including written portion requirements (e.g. your ER diagrams, RA, justifications/workflow, and normal form responses). You should provide your answers in a copy of the template found here.
- **<u>Part B:</u> `setup.sql`** - The DDL for your database
- **<u>Part D:</u> `load-data.sql`** - SQL with load statements to load your database
- **<u>Part E:</u> `setup-passwords.sql`** - A separate DDL file for implementing basic password management in your application (see appendix)
- **<u>Part F:</u> `grant-permissions.sql`** - A small SQL file to create users and grant permissions
- **<u>Part I:</u> `setup-routines.sql`** - SQL for stored routines and triggers
- **<u>Part H:</u> `queries.sql`** - SQL queries for your database
- **<u>Part J:</u> `app.py`** - Your command-line Python program
- **<u>Part K:</u> `readme.txt|md`** - README for staff to follow steps on using your application

Requirements for each file will be outlined next, and you can use the table of contents on the left of this document for quick access. In addition to requirements listed, we expect you to follow any requirements from previous assignments, so reviewing your feedback is encouraged, since the Final Project covers material throughout the term.

You do not need to finish these in order; most of your design planning should have been done in the Project Proposal, and the next natural step in the design and implementation process would be the ER diagrams, followed by implementing your DDL and then your queries. Students may find it preferable to start the Python application earlier with function stubs for the command-line functionality, or after you have the SQL portions completed.

Time estimates are included, but may vary depending on the amount of work you put into your Project Proposal and any overlap between sections (e.g. with a strong ER diagram your DDL-writing and Functional Dependency portions may be fairly quick). These estimates also are roughly made assuming no partner work (partner work is encouraged, and is expected to shorten time estimates quite a bit with a solid organization of responsibilities and collaboration).

Important: If you happen to not finish one of the sections, see the "Above and Beyond" section on "flex" points used for sections which go above and beyond for certain types of projects.

Part A. ER Diagrams

Estimated Time: 30-45 minutes

Required File: (part of written portion, `reflection.pdf`)

Overview: The ER diagrams for your database, similar to those of Assignment 6.

Reference Materials: Lectures 15-16, A6, #er-diagram-feedback

Notes: For this section **only**, we will allow (and encourage) students to share their diagrams on Discord (**#er-diagram-feedback**) to get feedback from other students on their ER diagrams given a brief summary of your dataset and domain requirements. This is offered as an opportunity to test your ER diagrams for accuracy and robustness, as another pair of eyes can sometimes catch constraints that are not satisfied or which are inconsistent with your specified domain requirements.

Requirements:

- Entity sets, relationship sets, and weak entity sets should be properly represented (also, do not use ER symbols not taught in class)
- Mapping cardinalities should be appropriate for your database schema, and in sync with your DDL
- Participation constraints should be appropriate and in sync with your DDL (total, partial, numeric)
- Use specialization where appropriate (e.g. *purchasers* and *travelers* inheriting from a *customers* specialization in A6)
- Do not use degrees greater than 3 in your relationships, do not use more than one arrow in ternary relationships.
- Use descriptive attributes appropriately
- Underline primary keys and dotted-underline discriminators
- Expectations from A6 still apply here
- Note: You do not need ER diagrams for views

Part B. DDL

Estimated Time: 30-45 minutes

Required File: `setup.sql`, Part B of **`reflection.pdf`**

Overview: Your DDL for your database, including indexes defined after your DDL.

Reference Materials: (DDL) A3, Midterm, and A6 feedback; Lectures 7-8, (Indexes) Lectures 13-14, A5

Requirements:

- You should have a *minimum* of 4 tables in your DDL with at least 12 attributes total
    - Views and materialized views do not count in this requirement unless granted permission from El; however, procedures and triggers with any materialized views *do* count towards the Part I requirements.
- Use cascading deletes and updates where appropriate (if it makes sense to have a cascading delete, you should have one).
- Use **`PRIMARY KEY`** and **`FOREIGN KEY`** where appropriate.
    - Remember not to use **`NOT NULL`** for PKs, **`REFERENCES`** without **`FOREIGN KEY`** is invalid for FKs in MySQL
    - Remember that **`SERIAL PRIMARY KEY`** is a useful shorthand for auto-incremented integer PKs (when defining an FK for a **`SERIAL PRIMARY`** KEY**,** you're FK will need to be **`BIGINT UNSIGNED`**, since **`SERIAL PRIMARY KEY`** is an alias for **`BIGINT UNSIGNED PRIMARY KEY AUTO_INCREMENT`**); you can find an example of this if you run into it in the 21wi Midterm DDL
- Use attribute types appropriately; e.g. don't use floating point types for something that should be fixed, use **`TINYINT`** for small ints, use **`CHAR`** vs. **`VARCHAR`** appropriately, etc.
    - You should have at least 5 different MySQL types across your tables
- Use **`NOT NULL`** and **`UNIQUE`** where appropriate (don't forget to refer to your ER diagram, which should be in sync with your DDL where reasonable)
- Use appropriate names for your tables and attributes, avoid keywords like **`date`** and **`int`** in your attribute names (e.g. use alternatives like **`order_date`**, **`app_date`**, **`household_size`** instead)
- Include brief comments for each of your tables, inline comments for less-obvious attributes (refer to **`setup-airbnb.sql`** for an example)
- **At least 1 index** should be defined at the bottom of this file. A small section in the written **`reflection.pdf`** will be provided to justify your index(es). If an index is poorly chosen and clearly not tested, you will not receive full credit. **This includes redundantly making an index on primary keys or foreign keys, which will already have indexes in MySQL.**

Part C. Functional Dependency Theory

Estimated Time: 20-35 minutes

Required File: (part of written portion, `reflection.pdf`)

Overview: A short written portion for coverage of the Functional Dependency unit.

Reference Materials: Lectures 17-18, A7

Requirements:

- Identify *at least 2 non-trivial functional dependencies* in your database
- Choose and justify your decision for the normal form(s) used in your database for at least 4 tables (if you have more, we will not require extra work, but will be more lenient with small errors). BCNF and 3NF will be the more common NF's expected, 4NF is also fine (but not 1NF).
    - Your justification will be strengthened with a discussion of your dataset breakdown, which we expect you to run into trade-offs of redundancy and performance.
- For two of your relations having at least 3 attributes (each) and at least one functional dependency, prove that they are in your chosen NF, using similar methods from A7.
    - If you have identified functional dependencies which are not preserved under a BCNF decomposition, this is fine
- Expectations from A7 still apply here.

Part D. `load-data.sql`

Estimated Time: 2 minutes

Required File(s): `load-data.sql`, CSV files if used

Reference Materials: Provided **`load-data.sql`** files from A3 and Midterm

Overview: This is the SQL with load statements to load your database (ideally, with CSV files including your datasets, but you can alternatively include INSERT statements)

Requirements:

- You will lose a non-trivial amount of points if you do not have data for us to test with, and you should have at least 50 records total across all of your tables (some students may have many more records if they are using a public dataset, which is fine)
- Your datasets may come from a public dataset, may "mashup" more than one dataset with relationships between another (e.g. annual statistics related to your domain-specific data)
- You may randomly-generate datasets (e.g. user data)
- You will need to cite the source of the data somewhere, ideally the written portion and your README
- We will run **`mysql> source load-data.sql`** after **`mysql> source setup-data.sql`**; for full credit, neither of these should produce warnings or errors.

Each **LOAD** statement should be **LOCAL INFILE**, using the following format (from the AirBNB example):

```
LOAD DATA LOCAL INFILE 'seattle_hosts.csv' INTO TABLE hosts
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS; -- If your CSV file has a row with column names
```

Part E. Password Management

Estimated Time: 30-45 minutes

Required File(s): `setup-passwords.sql`

Overview: A small portion of the Final Project will include Password Management, which was covered in Lecture 19 (embedded into the Final Project to replace last year's A8). An appendix and starting code for **`setup-passwords.sql`** is provided on Canvas to get you started.

Reference Materials: Lecture 19, Appendix

Requirements:

- Include a *users_info* table with usernames and *hashed* passwords along with a salt (schema provided)
    - You may extend that table to include an **`is_admin`** or **`role`** attribute if you have admin or other roles for users in your application (e.g. store managers, data managers, etc.)
    - If your original proposal had a *users* table, you can extend that and rename it to *users_info* with these added requirements
- Finish a procedure **`sp_add_user(new_username, password)`** to add a new user with their password hashed
- Finish a function **`authenticate(username, password)`** to return a **`TINYINT`** value of **`1`** (true) or **`0`** (false) based on whether a valid username and (unhashed) password have been provided.
- (Optional) Write a procedure **`sp_change_pw(username, new_password)`** to change a user's password

---

Part F. MySQL Users and Permissions

Estimated Time: 3 minutes

Required File: `grant-permissions.sql`

Overview: This is a small file to set up MySQL users with different privileges using GRANT statements, referring to the client and admin user(s) you identified in the Project Proposal.

Reference Materials: Lecture 12

These are different than the users of an application who might be represented in a `users_info` table related to your database schema. The username and password for each of the users defined in this file will be used to connect to the MySQL database in the **`get_conn()`** function in your Python program (you can submit a connection configuration for whichever user you'd like in your **`app.py`** program, we can test the different users manually). Also note that this is independent of the password management section; there is no hashing of passwords when setting up **database users** in a MySQL database.

Requirements:

- Create at least one each of a client and admin user with **`CREATE USER`**
- Grant permissions appropriate to each user (you can find advanced permissions if they are of interest to you here)

You can copy/paste the example from the Adoption Application demo in Lecture 19 (but change user names and database name appropriately):

```
CREATE USER 'appadmin'@'localhost' IDENTIFIED BY 'adminpw';

CREATE USER 'appclient'@'localhost' IDENTIFIED BY 'clientpw';

-- Can add more users or refine permissions

GRANT ALL PRIVILEGES ON shelterdb.* TO 'appadmin'@'localhost';

GRANT SELECT ON shelterdb.* TO 'appclient'@'localhost';

FLUSH PRIVILEGES;
```

Part G. Relational Algebra

Estimated Time: 20-35 minutes

Required File: (written portion, `reflection.pdf`)

Overview: This is the section which covers a small amount of relational algebra (since you were tested more on it in the Midterm). You may find it easier to start with your SQL queries first, or your RA. Schemas are not required in this part, only queries.

Reference Materials: A1, Lectures 1-4, Midterm

Requirements:

- Minimum of 3 non-trivial queries (e.g. no queries simply in the form **`SELECT <x> FROM <y>`**)
- At least 1 group by with aggregation
- At least 3 joins (across a minimum of 2 queries)
- At least 1 update, insert, and/or delete
    - This may be equivalent to said SQL statements elsewhere (e.g. queries or procedural code), but are not required to be; in other words, you can write these independent of other sections
- Appropriate projection/extended projection use
- Computed attributes should be renamed appropriately
- Part of your grade will come from overall demonstration of relational algebra in the context of your schemas; obviously minimal effort will be ineligible for full credit; it is difficult to formally define "obviously minimal", but refer to A1 and the midterm for examples of what we're looking for
- Above each query, briefly describe what it is computing; we will use this to grade for correctness based on what the query is supposed to compute; lack of descriptions will result in deductions, since we have no idea otherwise of what the query is intended to do.

Part H. SQL Queries

Estimated Time: 30-45 minutes

Required File: `queries.sql`

Overview: This file will include all of your SQL **`SELECT`** queries, some of which you will include in your Python program. We will grade these independently of your Python program working.

Reference Materials: Lectures 5-8, A2-A6

Requirements:

- Your queries should not have any syntax errors when **`queries.sql`** is sourced in MySQL 8.0.
- Use consistent SQL conventions demonstrated in lectures and homework sets (uppercase keywords, no blank lines within a query, proper indentation, etc.)
- Include a short comment above each query describing what it retrieves (in English, not repeating the SQL statements; we need to understand what your intention was for the queries when grading them for correctness)
- At least 2 of your SQL queries should be equivalent to 2 of your RA expressions, and these should be clearly indicated.
- The number of queries written will depend on how many queries you write in your application; projects with more queries or queries which are more advanced will be eligible for "above and beyond" points described at the end of this document.
- For queries in **`app.py`** which are parameterized (e.g. **`WHERE animal_type LIKE '%s'`**), you can replace the parameter with an example value in this file (since SQL queries don't have a notion of parameters)

Part I. Procedural SQL

Estimated Time: 25-40 minutes

Required File: `setup-routines.sql`

Overview: This file will include the setup code for defining your procedural SQL, all of which should be used in your Python application (UDFs may be called in a SQL query).

Reference Materials: Lectures 9, 10, 12, A5

Requirements:

- At least 1 UDF function
- At least 1 procedure
- At least 1 trigger
- We will load these with **`mysql> source setup-routines.sql;`** for full credit, this should not run into any errors or warnings.
- These should be chosen appropriately; remember that a UDF function returns a single value, not a set of values. If you have an un-parameterized procedure that does not modify any data, then it is probably better defined as a **`VIEW`**.
- For each, briefly comment what the function/procedure/trigger does. We will be using these descriptions when grading to check for correctness given the intended behavior, and a lack of documentation will result in deductions.
- Partner projects must have at least one additional function, procedure, or trigger

Part J. Command-Line Python Application

Estimated Time: 1.5-3 hours

Required File(s): `app.py` (ok if you have a separate `.py` file for client and admin programs)

Overview: This will be the command-line program you implement to simulate your application. We have provided a starting template which you may start with, and an example from lecture which you may reference (but not copy from). This year, we have provided a video going through password management and an example Python/MySQL application using an animal adoption agency database/client application. Unlike an exam, you may ask questions about this and get support in OH/1-1's during the Final Project.

Reference Materials: Lecture 19 and provided files (`app-template.py`, `lecture-demo.py`)

Requirements:
- Must be Python 3.0+, we will be running it!
- A start menu, listing options for users
- Functionality for a user to log in (e.g. users from the *users_table* from Part E, you may extend that table to include an **`is_admin`** or **`role`** attribute)
- Functionality to use 3 select queries that make sense with your application
- Functionality to call at least one procedure from Part I to modify your database
- No output should indicate MySQL usage; use the mysql library appropriately!
- Separation of concerns between MySQL and Python (discussed in Lecture 19)
- Part of your grade will be proper use of Python and program decomposition
    - E.g. use a **`main`** function, a **`show_options()`** function, and encapsulate your queries into functions based on your command-line support
- Use docstrings to document your functions
- You may use external Python libraries, but will have limited support from staff for material outside of CS 121
- You are also expected to follow standard testing and debugging of your program, don't forget to test edge cases (e.g. when no results are found)

Part K. README

Estimated Time: 10-20 minutes

Required File: `README.md` or `README.txt`

Overview: README for staff to follow steps on using your application

Requirements:
- Indicate where your data comes from
- Include instructions for loading your data from the command-line in MySQL
- Include instruction for running your Python program (e.g. if command-line arguments are supported, indicate what those are)
- If any files are written to the user's system, you should also indicate this

- You may guinea pig your README with a friend (without showing your source code) to ensure that it is clear enough to get started without looking into your project
- If there are any unfinished features, indicate this in your README so that we know some incomplete functionality is intended to prevent possible deductions (assuming you follow the rest of the requirements)

Part L. Rest of Reflection

Estimated Time: 15-20 minutes

Required File: `reflection.pdf`

Overview: Remaining reflection questions in Reflection document. Requirements outlined in the provided template.

Testing Your Project

When grading your projects, staff will download your code from CodePost and perform the following statements. To ensure you don't get deductions for unclear instructions on testing your database and application, we strongly encourage you to test the following as well.

```
$ cd your-files
$ mysql
mysql> source setup.sql;
mysql> source load-data.sql;
mysql> source setup-passwords.sql;
mysql> source setup-routines.sql;
mysql> source grant-permissions.sql;
mysql> source queries.sql; -- this is good to confirm no syntax errors
mysql> quit;
$ python3 app.py # we will add command-line arguments if supported/in README
… the fun part!
```

"Above and beyond" Points

Given the creative nature of the project, we will have a small portion of points available for projects which go "above and beyond" in one or more sections described above. These points will not be part of the 100 total points for the project, but may shift your overall grade up for points lost elsewhere. Some examples of "above and beyond" points include:

- Additional procedural SQL

- Dataset mashups (using datasets from different sources that are related to one another somehow, e.g. statistics for dates or locations along with your domain-specific data)
- Extra features integrated that utilize additional Python libraries or functionality (e.g. data visualization, machine learning, NLP, etc.)
- Larger database schemas (5+ tables) which are appropriately represented in ER, DDL, etc.
- Thoughtful written reflections discussing your design process, trade-offs you ran into, limitations of your schema/application, etc.
- Above-and-beyond attention to usability of your Python application (e.g. supporting option flags, saving user preferences, etc.)
- Particularly creative and "out-of-the-box" ideas for integrating MySQL in a Python application
- Anything else that is clearly more than required for a particular section defined above.

Have fun!

Appendix: Password Management Walkthrough

One component of the Final Project gives you practice exploring a very important design pattern used in relational database schema design: how to manage user authentication without storing passwords in an insecure way.

A template archive is provided with this portion, so that you can fill in the answers between the required annotations. Review Lecture 19 material for an overview motivating password hashing and salts.

Overview of Password Hashing and Salts

It is unfortunately all too common for database schemas to include a severe design defect: storing user passwords in plain text in the database. This is a catastrophic flaw for several reasons. First, once the database is compromised, the attacker can use users' passwords to access the system through normal channels. Second, most users use the same passwords for many different systems; once a user's password is known, that password can be used to compromise other systems that the user has access to. Your Final Project most likely stores some notion of user information, and this section will walk you through how to implement secure password management (fortunately, we get most of the hashing functionality available in MySQL!)

As discussed in Lecture 19, a simple solution is to apply a cryptographically secure hash function to passwords. Instead of storing the password itself, a *hash* of the password is stored. Authentication simply involves applying the hash to the entered password, and then comparing the hash values. Unfortunately, this approach is susceptible to *dictionary attacks*; users are very predictable, and by computing the hash codes of the most common passwords (e.g. "abc123", "password", and "123456" are in the top 10 most common passwords), an attacker with access to the database can simply compare the stored hash codes to a dictionary of hash codes computed from common passwords. This won't compromise all user accounts, but almost always it will compromise at least a few accounts.

To prevent dictionary attacks, one can prepend a *salt* value to the password before applying the hash function; if the salt is large enough then the potential for dictionary attacks is eliminated. Of course, there is still the issue of users picking bad passwords, but several techniques can prevent brute-force attacks, such as imposing a time delay on authentication failure, and enforcing strict policies on password choices such as expiring passwords after a period of time, and disallowing users from using any common password.

For this section, you will implement a secure password storage mechanism using both hashing and salting. Note that such functionality is normally implemented in the application sitting on top of the database, or even in the web client. You, however, will be implementing these operations as stored routines in the database, simply to avoid the complexity of having to interface with the database from another programming language.

All of your work for this problem will be in the `setup-passwords.sql` file, which we've started for you. Additionally, a helper function called **`make_salt`** is provided for you at the top of the file, which is able to generate a random salt of up to 20 characters. You can use it like this:

```
SELECT make_salt(10) AS salt;
```

First, we've provided with you a table table definition **`user_info`** that will store the data for this password mechanism. There are three string values the table stores:

- Usernames will be up to 20 characters long
- Salt values will always be 8 characters
- The hashed value of the password, called **`password_hash`**

We will be using the SHA-2 function to generate the cryptographic hash. (MySQL also has the MD5 and SHA-1 hash functions, but both of these are now considered too weak to be secure.) You can try out the SHA-2 hash function in MySQL like this:

```
SELECT SHA2('letmein', 256);
```

This built-in function can generate hashes that are 228 bits, 256 bits, 384 bits, or 512 bits. We will use 256-bit hashes. Note that the hash is returned as a sequence of hexadecimal characters, so you will need to choose a suitable string length to store the hexadecimal version of the 256-bit hash. As a result, the **`password_hash`** column is a fixed-size column that is exactly the right size, no larger and no smaller.

Alright, with all that said, we'll leave the rest of the work to you! Finish the two procedures in the provided code; do not change the **`-- [Problem 1x]`** labels.

1a) Finish the stored procedure **`sp_add_user(new_username, password)`**. This procedure is very simple:

- Generate a new salt.
- Add a new record to the **`user_info`** table with the username, salt, and salted password.

Make sure that you prepend the salt to the password *before* hashing the string. You can use the MySQL **`CONCAT(s1, s2, ...)`** function for this.

> Don't worry about handling the situation where the username is already taken; assume that the application has already verified whether the username is available, though you can improve this procedure to support that for an added challenge.

1b) Finish the function (not a procedure) called **`authenticate(username, password)`**, which returns a **`TINYINT`** value of **`1`** (true) or **`0`** (false) based on whether a valid username and password have been provided. The function should return **`1`** iff:

- The username actually appears in the **`user_info`** table, and
- When the specified password is salted and hashed, the resulting hash matches the hash stored in the database.

If the username is not present, or the hashed password doesn't match the value stored in the database, authentication fails and **`0`** (false) should be returned.

Note that we don't distinguish between an invalid username and an invalid password; again, this is to keep attackers from identifying valid usernames from outside the system.

1c) Of course, you'll need some users for us to test with! Add at least 2 users using your **`sp_add_user`** procedure **at the bottom of this file** so that when we load it, the database will be set up with users that your application will likely be interfacing with. As mentioned above, you may adjust the schema of the **`user_info`** table to support a fourth attribute to represent the user's status, depending on the design of your application.

1d) Optional. Create a stored procedure **`sp_change_password(username, new_password)`**. This procedure is virtually identical to the previous procedure, except that an existing user record will be updated, rather than adding a new record. Make sure to generate a new salt value in this function!

Testing Your Code

Once you have completed these operations, make sure you test them to verify that they are working properly (these are also included in the provided `pw-tester.sql` file). For example:

```
CALL sp_add_user('alex', 'hello');
CALL sp_add_user('bowie', 'goodbye');

SELECT authenticate('chaka', 'hello');     -- Should return 0 (false)
SELECT authenticate('alex', 'goodbye');    -- Should return 0 (false)
SELECT authenticate('alex', 'hello');      -- Should return 1 (true)
SELECT authenticate('alex', 'HELLO');      -- Should return 0 (false)
SELECT authenticate('bowie', 'goodbye');   -- Should return 1 (true)

-- provided tests for optional 1d
CALL sp_change_password('alex', 'greetings');

SELECT authenticate('alex', 'hello');      -- Should return 0 (false)
SELECT authenticate('alex', 'greetings');  -- Should return 1 (true)
SELECT authenticate('bowie', 'greetings'); -- Should return 0 (false)
```