Trainee Database Construction Project

For CMSC 461-01, Spring 2020 Semester at UMBC

Completed Project Report

Prepared by: Dylan Zach

Last Updated: May 10, 2020
Creation Date: March 5, 2020

Table of Contents

1. INTRODUCTION.....3
    a. Purpose of This Document.....3
2. REQUIREMENTS OF THE PROJECT.....4
    a. Background of the Project.....4
    b. High-Level Functionality Requirements.....4
    c. Requirements for Each Stage.....5
3. LOGICAL DESIGN.....7
    a. Reiteration of Data Requirements.....7
    b. Comments on the Process.....7
    c. E-R Diagram of DB system.....8
    d. Explanation of the Diagram.....9
4. CONCEPTUAL DESGIN.....11
    a. Overview of the Transition to MySQL.....11
    b. Table Overviews – Primary Sets.....11
    c. Table Overviews – Challenges.....13
    d. Table Overviews – Certificates.....15
    e. Table Overviews – Counselling.....16
    f. Table Overviews – Other.....17
5. PHYSICAL DESIGN.....18
    a. Circumstances of Physical Design.....18
    b. Overview of the Triggers.....18
    c. Notes about LoadAll Query.....19
6. PROTOTYPE, DEVELOPMENT, AND TESTING.....21
    a. Overview of the UI methods.....21
    b. High-level Requirements.....21
    c. The User Interface Designs.....22
    d. Possible Improvements to the UI.....27

1. Introduction

1.a. Purpose of This Document

The purpose of this document is to outline the entire process and methodology behind the creator's implementation of the requirements described herein. It is divided into sections based on the procedural workflow of the project, and at each section a conceptual view of the state of the project is both presented and defended.

This document also contains any relevant documentation that might be required in order to properly use the implementation once it is finally completed. This documentation might describe the final distinctions between the tables and data contained within the dataset, and how they interact with each other when called via a(n) SQL query.

It is in the creator's best interest that the layout of the workflow described in this document is both comprehensive and accurate, so that the resulting database product is accessible for the purposes of CertRUs's business solutions.

2. Requirements of the Project

2.a. Background of the Project

The original background and purpose for this project is to create a database relating to the facilities of CertRUs's certification business – namely, the certification of specific skills for its trainees, and the architecture that revolves around this process. It should also provide a frontend for interaction from a user in order to manipulate and process data relating to the CertRUs facilities.

As an example, the CertRUs group might have a number of trainees who receive counselling in a skill from a skill master. The project would require the identification of the data contained within this interaction, and the storage of it in a way that is both comprehensive and able to be manipulated using queries.

2.b. High-Level Functionality Requirements

In this section, we will explain the requirements for implementation of the CertRUs requirements from a general perspective. What applications must be used in order to implement the project functionality itself, and for what reasons? How does the system as a whole work?

In order to achieve the goal of regulating and handling the information of each entity, our solution involves the use of a SQL database server architecture. This is because in large scale corporate settings, it is the most effective method to promote the intended goals of data access and data management requested by the project requirements. As such, requirements of individual data values will be defined in the preliminary stages in the forms of their database and SQL equivalencies, in order to prepare for the later steps of specific product implementation.

The high-level functionality also refers to how the database should be required to work from a user standpoint. In terms of who shall be using this database application, the scope is outlined in the user requirements to include the trainees and employees of Cert. But instead of interacting directly with the database, they will require an intermediary application to allow for the addition, removal, and manipulation of data, since the users may not be necessarily the most "tech savvy" in handling complicated implementations.

Our implementation will implement a Jupyter notebook that has the ability to interact with the SQL database. This fulfills the requirement of an intermediary abstraction to allow for indirect control over data in the database.

Finally, in addition to the software of the project, this report will detail the methodologies of the design phases to come, in which models of the specific implementation of data and datasets will be notated as per their relevance to the phase of the project. This requirement is meant to fulfill the need for documentation and explanation of our chosen implementation.
To summarize: the ideal, high-level functionality of this project refers to three major parts: The SQL database, the Jupyter notebook frontend, and the associated report and documentation. Each of these fulfills a requirement for one aspect of the ideal structuring of the project and its project requirements. To reinforce this, further descriptions in the report herein should focus on both how each aspect is implemented, as well as how they interact with each other to create a data handling product that is both functional and comprehensive.

2.c. Requirements for Each Stage

The project as a whole has been broken down into multiple design stages – each reflective of the further sections within this report. In addition, each design stage has been delegated to a full chapter in this report; functionally, this also gives an introduction to each section hereafter.

This subsection's purpose is to briefly overview what should be accomplished in terms of the project at each stage, and how it fits in with the requirements of both the project and the ideal functionality of the final project. These requirements are based both on the project requirements, as well as our own decisions for the best implementation of the user requirements.

Phase A: Requirements of the Project (Section 2)
The goal of this phase is to outline the requirements of the project, in order to define our interpretation of the execution of tasks relating to the overall product. This includes definition of applications used, and what functionality they hold in the goal of data maintenance. It is not to go into detail about specific data sets, however – its main purpose is clarification for users and future documentation purposes. If it were not already evident, output of this stage is the current report section and associated documentation.

Phase B: Conceptual Design. (Section 3)
The goal of this phase should be to implement the specific data requirements as outlined in the user requirements. This includes creating an E-R relationship model that is both comprehensive and accurate to the description of how the models will interact in the real world. The major output of this stage should be a comprehensive E-R model describing the requirements for data handling and access, and why it is an ideal model to use.

Phase C: Logical Design

The goal of this phase should be to expand the E-R model in such a way that actual tables and attributes are modelled as schema for a database. Whereas the E-R model is more abstract, the output of this phase should be a well-designed schema model of the different relations, how they interact, and why this is an ideal model for the requirements of the client architecture.

Phase D: Physical Design:
The goal of this phase is to adapt the schema model in order to create an actual database in the MySQL program. Output should be a fully functional MySQL database with the ability to add and remove tuples satisfying the constraints of each relation's attributes, as well as perform queries to perform a number of high-level data retrieval tasks – some of which outlined in a project appendix.

Phase E: Prototype, Development, and Testing
The goal of this phase is to create a Python Jupyter Notebook with SQL scripts that can be used by a user defined within the scope of the project. Such a notebook should be comprehensive and understandable, as well as functional with the same practicability as the database. Output of this stage is a frontend for the database in the form of a Jupyter notebook.

Phase F: User Guide and Documentation
In addition to this report and any other relevant documentation, the goal and output of this phase should be a user's guide understandable by the users of this product – most ideally packaged with the Jupyter notebook from Phase E. This will assist with comprehension and further integration in the corporate space.

3. Logical Design

3.a. Reiteration of Data Requirements

In order to create a representation of the relationships between the data elements in an E-R model, we must first consider the project description of the data requirements. These requirements outline the process through which CertRUs conducts its business in the context of how elements interact with each other. With these constraints in mind, we can create a high level connection between the parts of the database, and isolate what aspects could make for entity sets, which could make for entity attributes, and which could identify a relationship between sets.

The data requirements specify four major categories of entities within the system:

A trainee refers to a person within the database that is able to be counselled (or give counselling) by a skillmaster in a particular skill, and perform challenges in order to show knowledge of a skill. If a trainee gets a good enough skill rating on challenges, they will receive a certificate. Earning an intermediate or advanced certificate upgrades a trainee to a mentor.

A certificate refers to a set of objects which are awarded to persons achieving some score in a set of skills. They may also be based on earning previous certificates.

A challenge is an event which involves four different participants: one trainee, and three mentors. The challenge is meant to test a variety of skills, of which each mentor will grade the trainee on based on their demonstration of knowledge. Each mentor must be a "skillmaster" in all challenge skills in order to evaluate the challenge. (A "skillmaster" is defined as someone who has received a 4 or higher in a skill in their own challenge.) The final score each trainee receives for a skill is the average of the three mentor's responses.

A mentor refers to someone who is able to serve as an evaluator for a trainee's challenge, provided they are a skillmaster in all of the skills tested in the challenge. Otherwise, a mentor seems to be equivalent to a trainee.

3.b. Comments on the Process

My methodology for creating the E-R diagram relied on differentiating which of the elements forms a non 1-1 relationship with another element. In other words, in what cases do multiple tuples have to be created in order to represent a join between two relations?

One problem I had to overcome was what to do on the matter of skillmasters. "Skillmaster" seems to be a property based on multiple datasets, and not something that

can directly be correlated 1-1 with a user relation, based on multiple attempts for different challenges that might each test different skills. Also, the requirements were somewhat ambiguous on whether or not a trainee could give counselling if they themselves were a skillmaster. I assumed that this would be true, since it would make sense since the main differentiator of the mentor set is the ability to give evaluations, not counselling. As such, the terms “skillmaster” and “counselor” are more or less interchangeable in my implementation.

3.c. E-R Diagram of DB System

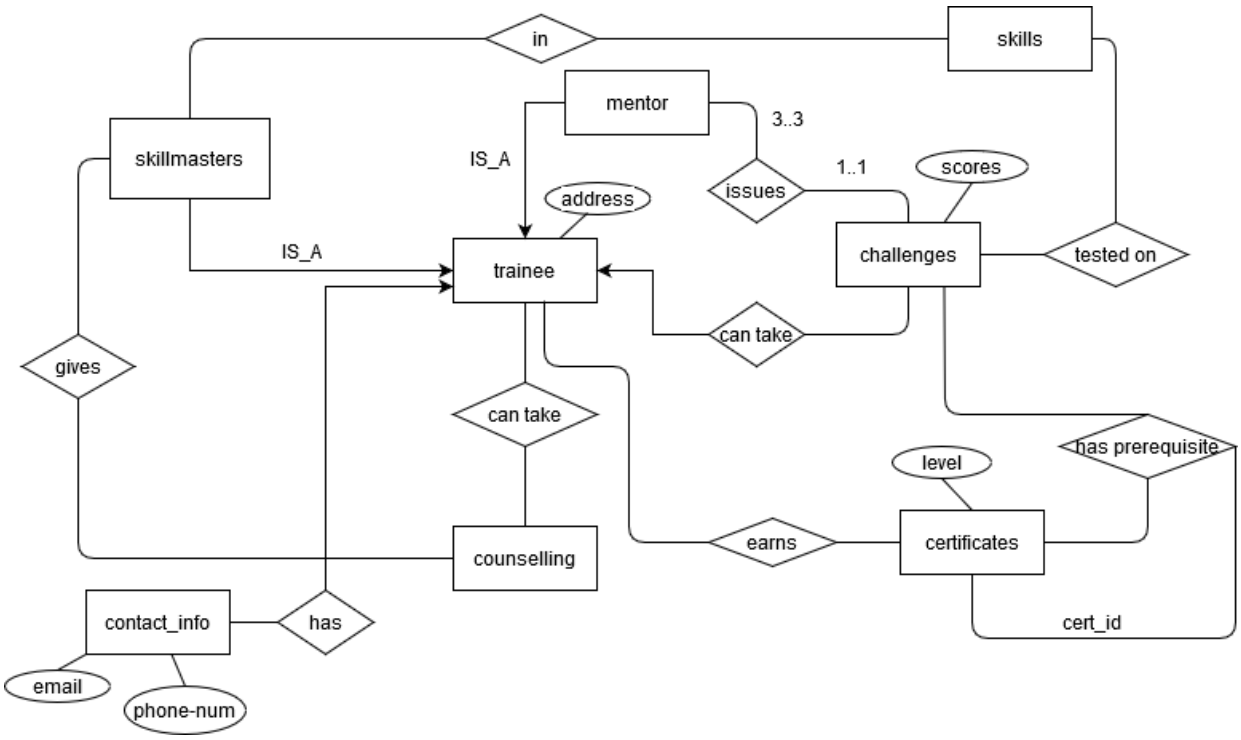

Figure 1: It might not look it, but this diagram is the product of many hours of work.

3.d. Explanation of the Diagram

This diagram leaves out a few of the more specific attributes for each relation, for the benefit of being more general. For example, the primary keys of each relation have not been included, unless they are directly related to the function of similarity between sets. This will most certainly be expanded upon in the next section. (In fact, the whole purpose of the Logical Design step is to buff out the attributes for each relation. It is my hope that I will not be docked points for such a reservation.)

In my diagram, I have included a number of new relations that exist to help satisfy the requirements for each data set. Some of them, such as the "counselling" relation, seem to have been implied to exist from the requirements (seeing as how it was defined as "each counselling session having a specified start and end date", which would be difficult to replicate were it not its own relation). Others were introduced in order to create new mechanisms for the storing of data, and will likely be expanded through query functions in the final product.

SKILLS- Presumably, different skills to be tested should be able to be added and removed as one desires. The skills relation keeps track of the skill name and skill id. Since a skill mostly just revolves around these two attributes (kind of like an enum type), I do not anticipate many more attributes here.

SKILLMASTERS – From what I've seen of the requirements, a counselor and a skillmaster are functionally identical. So, this set could also be called "counsellor".

You might also notice how there is a multiple inheritance lattice around the trainee relation. This is because membership of the skillmasters set and of the mentor set does not necessarily exclude an individual from participation in trainee-related activities. As such, the easiest way to model this is with inheritance of the different tasks each relation might have. Please note that it is therefore possible in this model for a person to be either a skillmaster or a mentor, or both. One thing that is not immediately obvious is that the actions of a mentor are actually dependent on whether or not one is a skillmaster. (ie: someone can be a member of mentor without being a skillmaster, and thus unable to proctor challenges.) However, a further implementation would account for this.

COUNSELLING – Like explained before, "counselling" refers to a connection between a skillmaster and a trainee. The relation is many-many, since although a single counselling is between a single skillmaster and a single trainee, there seems to be no limit on how many skillmasters a trainee can be learning different skills from, and no limit on how many trainees a skillmaster can have. This does not seem to be specified, but it is the rational interpretation.

CONTACT-INFO – As it is the base case for a person in the system, the contact info (except for address, which does not seem to require a non-1 to 1 relationship, and could probably be stored as an attribute for database simplicity) is associated with the

individual trainee. This can include multiple instances of phone numbers and emails. Although I suppose multiple people could share a phone number, such as in the case of a home phone, for the purposes of identification it's fine if the cardinality is 1 to many.

Instructions were ambiguous as to whether or not phone number and email had to be in separate relations. If this is true, then that change will be made.

Some other things to point out include the mentor-challenges relationship. In reality, many mentors can be assigned across many projects. However, for each instance of "challenges", only three mentors can be assigned. This felt more important to notate.

Finally, certificates can be based off of either grades achieved in challenges, or on other certificates (hence, cert_id).

UPDATE (4/8/2020): In line with the current implementation changes, the following footnotes would like to be acknowledged regarding changes on these bases made in the Logical and Physical Design phases:

1) In Figure 1, the composite attribute address has been moved to the contact_info entity set entirely. This was to cut down on redundancy.
2) In Figure 1, a certificate can have prerequisites of skills, not challenges. Though the two are closely interlinked, in the Logical phase it is clear that the prerequisite comes from an instance of a trainee having some required skill obtained.
3) In Figure 1, contact_info should be a weak entity set.

4. Conceptual Design

4.a. Overview of the Transition to MySQL

The Logical Stage is very important for setting the theoretical groundwork for which the eventual database will operate upon. Though it does not describe anything necessarily concrete, the idea of relationships between independent entity sets should be established early on.

Even when a single table becomes normalized through the addition of multiple tables to join and offer foreign-key support, the conglomeration should still hold the same general relationship role within the database as it did within the E-R diagram. In this sense, simplifying things at first and getting more and more complicated with each step ensures that there is little margin for tactless error.

So, in order to transition from a general network to a concrete database with constraints and the like, we need to acknowledge the intermediary step – that is, what exactly are the tables and attributes of our database going to be?

In this Conceptual Design step, I hope to clearly define the purpose of each table and its attributes within the database and the project as a whole, in order to justify my implementation and its adherence to the proper assignment requirements.

4.b. Table Overviews – Primary Sets

First and foremost, my implementation of a database will likely be making strong usage of primary-foreign key relationships between tables and entity sets. The reason for this is that enforcing a strict hierarchy in the data set prevents minor mistakes in data entry (such as a misspelled id name) from having larger effects by catching them in violation of the relationship.

In my implementation, I have isolated three tables that do not represent any of their data values by foreign keys. These are what I would call “primary sets” – sets of which foreign key references from drive the connections of the rest of the database, and the hierarchy therein. In any insertion they should be first, and in any deletion they should come last.

|**TABLE NAME**: skills
|**PRIMARY KEY(S):** skillname
|OTHER ATTRIBUTES:
| skill_id
|
Description of this set: The "skills" table is the table in which primary keys for skills can be added, with each skill having a varchar name skillname, and a numeric id skill_id.

One may immediately notice that although skill_id exists, it is not really a key in any way. The reason for this is twofold. One: having multiple primary keys could let arise the odd situation of having two skills with the same id value, but different names. In this case, I fail to see the usefulness of an id. Two: Because they are referenced in queries so often, it made much more sense to me to be referencing skillnames instead of skill_id in data relations, mostly for the purposes of readability and modifiability, at the cost of slight performance

At any rate, this table mostly exists for the adding and removal of skills from the set. Due to foreign key relations, it should be impossible to do this anywhere else.

|**TABLE NAME**: certificates
|**PRIMARY KEY(S):** cert_acronym
|OTHER ATTRIBUTES:
| cert_title
| classification
|
Description of this set: In this set, one can add and subtract certificates from the dataset. Each certificate has a title (cert_title), a classification that must be in range Beginner, Intermediate, or Advanced – and a designated acronym for the certificate's title (cert_acronym), which is also the primary key.

Here, I decided to make acronym the primary key instead of title in order to conserve database space. There is a threshold for efficiency vs usability, and I think acronyms are a great way to reach that balance, especially with how long some cert_titles could likely get.

|**TABLE NAME**: challenge
|**PRIMARY KEY(S):** challenge_acronym
|OTHER ATTRIBUTES:
| challenge_name
|
Description of this set: This set allows for the adding and removal of challenge names. (See: 4.c. for more info on that.) However, because a challenge can have many skills associated with it, that is represented in a different table. As with many of these primary tables, its main function is to be manually added to and removed from by an administrator.

|**TABLE NAME**: trainee
|**PRIMARY KEY(S):** user_id
|OTHER ATTRIBUTES:
| name
|

Description of this set: A trainee is a name for any person within the dataset. This may sound counterintuitive, but for reasons I will explain later, every person enrolled in the database has their first entry created here in trainee. Each entity has a numeric id user_id and a name, the former of which is the primary key.

The reason why I went with key as primary key was, again, a liberty taken. Names can get very, very long, so to avoid excluding any people's names I made that attribute one of the largest in the database. Plus, having some keys in tuples is alright. I just didn't want a single table of the database to become just a mess of numbers.

4.c. Table Overviews – Challenges

Next, I want to go over my implementation of what was outlined as the "challenges". These are meant to be tests of a trainee's skills that are overseen and graded by a board of 3 mentors – mentors being those who are a special rank above trainee.

|**TABLE NAME**: mentor
|**PRIMARY KEY(S):** <none>
|**FOREIGN KEY(S):** user_id references trainee(user_id)
|OTHER ATTRIBUTES:
| name
|

Description of this set: As described above, a mentor is a special rank that has greater access to relationships compared to just trainee. In my implementation, I wanted to stick as close to the ISA relationship between mentor and trainee in my earlier steps (As in, all mentors are trainees), so it relies on a foreign key to denote a subset of trainees as also being mentors.

Even though its attributes are the same as trainee, at least sitting in a table, this table represents more of an elevated privilege than an actual class of person. The main benefit a mentor gets is, again, the ability to proctor challenges for trainees. Trainees cannot proctor, unless they are also mentors.

There should also exist correct checks to make sure that mentors can be promoted, in this case upon obtaining an Intermediate or Advanced certificate. Note that the user_id never changes, no matter which role a person is in. It's all about existence.

|**TABLE NAME**: challenge_skills
|**PRIMARY KEY(S):** <none>
|**FOREIGN KEY(S):** challenge_acronym references challenge(challenge_acronym),
| skill_name references skills(skill_name)
|OTHER ATTRIBUTES:
| <none>
|

Description of this set: As continued from the "challenge" dataset explanation, this table exists to give a specific skill to a challenge by the name of challenge_acronym. Challenges are defined as consisting of many different skills, so this is a kind of "bridge" between the challenge and skill sets such that it relates to the individual instance of a challenge.

|**TABLE NAME**: challenge_event
|**PRIMARY KEY(S):** challenge_inst_id
|**FOREIGN KEY(S):** challenge_acronym references challenge(challenge_acronym),
| graded_trainee references trainee(user_id)
|OTHER ATTRIBUTES:
| date_held
|

Description of this set: A challenge_event refers to any specific instance or event of a challenge in which a trainee is actively testing for grades in the scores of their challenge. Each challenge_event has its own id, which uniquely identifies its occurrence in the data sets. It also takes the foreign keys challenge_acronym from challenge (in order to signify what the challenge actually is), and graded_trainee from trainee (the person being tested).

|**TABLE NAME**: score
|**PRIMARY KEY(S):** <none>
|**FOREIGN KEY(S):** challenge_inst_id references challenge(challenge_inst_id),
| trainee_id references trainee(user_id),
| mentorN_id references mentor(user_id) /*for N = 1,2,3*/,
| skill_name references skills(skill_name)
|OTHER ATTRIBUTES:
| gradeN /*for N= 1,2,3*/
| grade_avg
|

Description of this set: A score entity represents all of the mentors and their corresponding grades for a single skill in the challenge with id challenge_inst_id. That is, a challenge instance has one corresponding score entity for every skill within the challenge in question.

How it works is that the score of mentor1 is denoted in attribute grade1, etc. Originally, this set was going to operate a lot differently, with each mentor grade for each skill having its own row (3 * skill_count for each challenge_inst_id). However, finding a way

to assign triggers to that was borderline futile, so I took advantage of the 3-mentor absolute in the documentation and made a wider table as a result.

Ideally, the grade_avg spot should be left null. It will be calculated based on inputted gradeNs by a trigger.

4.d. Table Overviews – Certificates

|**TABLE NAME**: has_skill
|**PRIMARY KEY(S):** <none>
|**FOREIGN KEY(S):** user_id references trainee(user_id),
| skill_name references skills(skill_name),
| challenge_inst_id references challenge(challenge_inst_id)
|OTHER ATTRIBUTES:
| grade
|

Description of this set: This set differs from the other tables, in that values are not supposed to be automatically inserted by a database admin. Rather, it bases its values on a trigger. In this case, whenever a new score is added, the score and its skill are added to has_skill under the id user_id = user_id.

This table denotes whether or not a trainee has a certain skill and skill level – something that makes the prereq table run itself faster.

|**TABLE NAME**: earns_cert
|**PRIMARY KEY(S):** <none>
|**FOREIGN KEY(S):** user_id references trainee(user_id),
| cert_acronym references certificates(cert_acronym)
|OTHER ATTRIBUTES:
| <none>
|

Description of this set: This set is the certificate equivalent to has_skill. In that, whenever trainee ‘Alice’ meets the prerequirements for earning a certificate, an entry for that certificate with user_id ‘Alice’ is added to earns_cert.

|**TABLE NAME**: prereq
|**PRIMARY KEY(S):** <none>
|**FOREIGN KEY(S):** cert_acronym references certificates(cert_acronym)
| prereq_acronym references certificates(cert_acronym)
| skill_name references skills(skill_name)
|OTHER ATTRIBUTES:
| grade
|

Description of this set: This table, in short, gives out certificates to trainees based on whether or not their earns_cert and has_skill meet the proper criteria at update-time.

Entries in this set can either consist of a skill_name and grade, or a prereq_acronym – as well as the cert_acronym for which certificate has the prerequisite in question. In other words, both certificate and skill-only prerequisites are represented in this table, but with different non-null attributes.

4.e. Table Overviews – Counselling

Counselling was one of the easier requirements to build tables for, simply because it didn't require a lot of normalization. It also doesn't seem to have any direct effects on the other aspects of the database. So, if nothing else, this section would probably be complete first.

|**TABLE NAME**: skillmaster
|**PRIMARY KEY(S):** <none>
|**FOREIGN KEY(S):** user_id references trainee(user_id),
| skill_name references skills(skill_name)
|OTHER ATTRIBUTES:
| name
|
Description of this set: This set behaves a lot like the mentors table, in that skillmaster ISA trainee. Unlike mentor, however, a skillmaster has multiple entries in the skillmaster table, one for each of the skills they are a "master" (avg. Score of 4) in, added each as a side effect of the triggers that update has_skill.

|**TABLE NAME**: counselling
|**PRIMARY KEY(S):** session_id
|**FOREIGN KEY(S):** trainee_id references trainee(user_id),
| skmaster_id references skillmaster(user_id),
| skill_name references skills(skill_name)
|OTHER ATTRIBUTES:
| date_beginning
| date_ending
|
Description of this set: A skillmaster and a trainee can engage in an event of counselling, where some skill_name registered between the two in the table from date_beginning to date_ending.

The requirements were stating that "either a skillmaster or mentor" can be a counsellor, but I took that to mean that a mentor had to be a skillmaster to counsel in a given skill. So, it doesn't actually matter if you're a mentor; as long as you're a skillmaste in skill, you can be skmaster_id.

... Would it be possible to counsel oneself? I suppose it wasn't expressly prohibited...

4.f. Table Overviews – Other

|**TABLE NAME**: contact_info
|**PRIMARY KEY(S):** <none>
|**FOREIGN KEY(S):** user_id references trainee(user_id),
|OTHER ATTRIBUTES:
| street_adr
| city
| zip_code
| phone_no
| category

Description of this set: Not too much to explain here, I don't think. Each user_id can have many entries of contact_info, and since it doesn't seem to link anywhere else in the table it can probably be filled out with however many null values in a tuple as one wishes. "category" should probably be a helpful identifier (eg: "Home", "Work"), but again, there's more interesting stuff in the database going on than this.

Street_adr, city, and zip_code could be used to form an address entry.

5. Physical Design

5.a. Circumstances of the Physical Design

Even with all of the tables mapped out, there is still a huge mountain to climb, and that is adapting all of these queries to the MySQL database and its SQL command script. This was not easy, for a number of reasons.

First and foremost, some big things about the Conceptual Design did have to change during this stage, most notable being the score table. (These steps have since been edited over their previous counterpart.) One problem was that implementing MySQL's triggers was not always simple, and especially not to the format that one might expect from the SQL implementation logs such as that in the course's textbook. So, that was a fun wild goose chase for hints.

Regardless, however, the basic triggers and tables have been implemented, and especially for the case of inserting rather than removing entries the database works perfectly. However, there are a couple of issues that may be worth revisiting, should they become a possible conflict in the next step.

5.b. Overview of the Triggers

As I explained back in the Conceptual Design, not all of the tables are meant to be modified and inserted into solely by user queries. There is, for the database to be up to the project specifications, a degree of autonomy that needs to take place. For that purpose, we need to set some triggers to be able to detect when certain events happen that would conceptually warrant an update in some table.

For reference on this and other queries, check the three SQL query files that should have been made available along with this section of the report.

>>TRIGGER: GetGradeAvg before insert on score

PURPOSE: This trigger works by updating the (ideally null on insertion) grade_avg for a new insert on the score database. It sets the new grade_avg to the sum of grade1...grade3 divided by 3.
REASON INCLUDED: grade_avg is needed in order to compute an accurate skill entry for a has_skill for user_id.

>>TRIGGER: AddNewSkillset after insert on score

PURPOSE: This trigger processes the newest insert on score database after the previous trigger takes effect. First, it inserts a corresponding tuple into has_skill using some of the new insert's parameters. Then, it decides whether or not the score warrants an addition into the skillmaster table of the trainee_id (new.grade_avg = 4).
REASON INCLUDED: First of all, this gives skillmaster the ability to update on its own, without user intervention. Secondly, it updates the has_skill, which is important because it saves a couple joins in having to get to a user's own skill levels – something which has changed since my initial E-R diagram.

>>TRIGGER: GiveNewCertificates after insert on has_skill
PURPOSE: There is a method to the madness herein. What it basically does it that it scans through each entry of cert_acronym and all of its corresponding prerequisites in prereq to see if these prerequisites are a subset of the has_skill and earns_cert for user_id.

In the end, if user_id is eligible for any certificates, and it has not already been allocated, then add that certificate to user_id's earns_cert list.
REASON INCLUDED: To the best of my knowledge and my coding ability, this was the best way to implement this in a way that managed to agglutinate all of the cert_acronym in prereq in such a way that it could be compared as a set. My implementation requires this, because it is whether or not the sets are truly inclusive that determines whether prerequisites of cert_acronym are truly in user_id's has_skill and earns_cert sets already, to fulfil the prerequisite.
COMMENTS: There is a lot that needs to be fixed here. For one, it doesn't take into account grades very reliably, which can be a big problem. For some reason the clause I put in isn't catching anything. This is, to my knowledge, the only huge logical problem in the database. But this may have to be fixed in the future, should the problem come up.

5.c. Notes about loadAll query

If you take a look at loadAll, you might notice that I took a couple of liberties in the process of making the database functional. These include but are not limited to:

1) Early on I realized a problem with how the requested system works, especially on my implementation. In order for a trainee to learn skills, they have to take tests. But in order to take tests, someone has to be able to host the tests. But in order to host the tests, someone has to already have learned that skill. That's a problem.
    a. I solved this by adding in a bunch of skillmasters and mentors that didn't actually get their scores from taking tests. Let's just assume that these are legacy users, and that they were already in the CertUs's old database.
    b. For reference, a has_skill without an associated challenge id was added as part of this, up to about line 208.

2) There are a couple of queries at the bottom to return data. These should be able to, in order:
    a. Find the skills and grades for user with user_id = '15056'
    b. Select trainees and their skills, including where challenge_inst_id is null.
    c. Join the counselling table entries with the names of the trainee and the skillmaster, respectively.
3) Queries and reports from functional requirements shall be included with the final milestone, as it is not specified it must be done now.

6. Prototype, Development, and Testing

6.a. Overview of the UI methods

It is generally view as a poor design choice to allow a user to be directly interfacing with a database. And seeing as how a CertRUs employee using this isn't supposed to be well-versed in SQL language commands and practices and good data management (the latter of which is imperative to be as untrusting with as possible),

The solution to this is to embark on what may be the final step of initial production – a usable UI that is both functional to use, and restrictive in its possible applications so as to prevent accidental misuse.

6.b. High-level Requirements

According to the requirements of the project, the completed UI service ought to be written in Python as a Jupyter notebook. Such a notebook should be able to use commands and functions in order to perform a set of defined interactions with the database.

This should include the ability to connect and disconnect from the certrus database, tuple manipulation (insertion, modification, and deletion), and the ability to generate reports of data values.

In the next section, I will describe the mechanics of how my implementation of the notebook works, and explain how each of the requirements were fulfilled through my UI design.

6.c. The UI Designs

Because my implementation(s) of the User Interface are written in Jupyter notebook, the current iteration is entirely text-based. This is what I feel is most suitable to the medium. However, expansion to a visual representation – though likely not of high priority – could definitely be possible, thanks to the flexibility of the Python language. In this section, I will walk through the parts of the program in-depth.

== Section One: Imports and Var Declarations

In traditional Python fashion, imports of various functions required for the program runtime are imported at the beginning. To quickly summarize the purpose of each of the imports:

- *mysql.connect:* This is used for the database connection between the notebook and the port on which the MySQL server is running. Basically a requirement of the task at hand.
- *getpass:* This library is used for inputting the user's password. It is generally a bad idea to process this as plaintext, and this seemed to be a secure option for the level the current project is at.
- *xlsxwriter:* I understood the "generate reports" portion of the project as generating an actual file of the pulled SQL requests, which I believed was the most business-practical application of this requirement. Thanks to the magic of Python, we can create actual xlsx documents from SQL queries!
- *re:* When inputting a filename for the previous step's operations, it is possible for the user to input incorrect characters. This library is used in a regex equation to filter out any invalid filename characters.
- *traceback*: Halting the program execution whenever an exception was found proved to be annoying. Traceback allows for exceptions to display error codes without halting execution.

In addition, several variables were declared after this point and before "main". These are variables which I included in this section partially because I am unfamiliar with the way Python handles variable scope. These variables should be "global" in scope, at least in a C-base sense.

These are the "table" variable – which needs to stay the same between functions in case a user wants to reuse the same table over and over again easily (explained in-depth later), and the db_tables and db_inaccess tuples. The latter of these refers to the total name of all of the tables in the database implementation. This is for reference, to verify that SQL queries are referencing tables that actually exist in the database, which we can do because the DB table count itself should not be changing.

== Section Two: Functions

In order to increase code reusability, and to encapsulate certain operations such as database access, my implementation contains several functions, which are generally called to serve the purposes of the original functionality of the project, such as managing tuples and creating reports.

This section will explain the purposes of each of them, and how they work efficiently for their given solutions.

First of all, we should make clear that the three functions for managing tuples (insert_sql, update_sql, and delete_sql) all work in similar ways to submit and receive queries. Here, I will explain these similarities.

Firstly, each of them assume that some data values have been collected from the user beforehand. The main goal of these functions is to repeatedly append to form an SQL query, and then submit that string query to the database system. What data values are taken in depends on which function is being called, and will be explained for each function later on.

Secondly, each of these functions contains a special parsing line with the purpose of differentiating between strings and non-strings in the query. Basically, the function "string.isdigit()" is called, with the goal to encapsulate any strings that contain a non-numeric character in quotes. And, because we are appending to the string directly, we don't have to worry about substituting in with %s or %d for parameters.

Finally, the construction of these sql statement strings occurs in sequential order. First, the variables after the "SELECT " section are appended, then after " FROM ", onwards, to give an example. Again, this differs based on the query. Out of the many options I tried in getting this to work, this was the most consistent effort. It also has the benefit of being expandable. Sure, the option of using a huge if-elif-elif depending on the table type alone was possible, but using for loops to construct statements based on the assumption that the query is already going to be formatted correctly for the table trying to be accessed allows for this greater freedom of coding style. Besides, the try-except block catches anything wrong anyways.

> *-get_paramnames(table)*
> Out of all of the functions, this one was the one that I feel was the most necessary addition that was not stated in the initial requirements. What this function does is take in the name of a table (assuming that table has been verified to be part of the database), and then runs a special SQL query that returns a set of the names of the column headers as a string. This function serves two major purposes. One: In the insert_sql function, it is used to define the names of columns for the table being inserted in the SQL query there (see: col_names). Two: Before most major operations, I thought it was a good idea to print out the

names of the columns in the tables beforehand. This saves the user time in needing to look up exactly what the specific names of the columns and order is, since some operations are dependent on being able to cite the precise attribute name in a query.

-insert_sql(table, parameters, col_names)
This function operates by appending the given parameters into an SQL query with the function of inserting the given “parameters” into the given “table”. This creates a new entry in the table.

-update_sql(table, param_check, param_change, param_newvalues)
This function operates by appending the given parameters into an SQL query that updates all tuples fitting the criteria of “param_check”. Multiple attributes can be checked for, by appending through the list with “AND” statements. Also, all attributes defined in param_change will be changed to the corresponding values in param_newvalues. This means that multiple param_change can be defined, but only if they correspond directly to the orderings of the same number of param_newvalues.

-delete_sql(table, param_check)
This function operates by appending the given parameters into an SQL query that deletes all tuples in the table “table” where “param_check” holds true.

-get_valid_filename(s)
See: re definition in Section 1

-make_report(filename, param_include, col_names, param_where)
This is the most complex function out of all, but its construction of SQL queries is pretty straightforwards. It goes like SELECT param_include FROM col_names WHERE param_where. This does require some user formatting, especially on the part of reducing attribute ambiguity for param_include where col_names spans multiple tables. Yes, col_names can format itself out of multiple tables, which by default are just JOIN’ed. Once the query is constructed and received, an xlsx document of formatted “filename”.xlsx is created, and each column-row space is filled with the corresponding data entry based on the number of attributes in each entry. Then, the file is closed, and returns None should no errors occur.

== Section Three: Login Interface

Upon launching the program, execution skips to this section of the program. The purpose of this section is to take the user's login credentials, and attempt to start a new session using these entries. If it is unsuccessful, then the While loop that the section is contained in will continue indefinitely, unless the user manually exits the program. However, if the user is a valid database login, then a session will be established and a break will be instigated to go to the next section.

Note that from here on the majority of these functions were tested only with root. I presume that there may be some problems depending on whether or not the user trying to use the database has fewer permissions than root (as is likely), but these should probably be caught respectively by the try-except blocks in the actual SQL functions as described above.

```
=== Welcome to the CertRUs Database Utility! ===
Please enter your username
> not_root
Please enter your password
> ........
\\\Error: Username-password combination invalid!
Please enter your username
> root
Please enter your password
> ........
```

Figure 1: The login interface. Notice how the passwords are hidden.

== Section Four: Utility Execution

Assuming that a successful connection has been made between the program and the MySQL certrus database, the major While loop that is essentially the program begins. From here, the user is presented with a variety of input choices.

It should be noted that from here on, each of the first four options has the majority of its technical code residing in the respective function that was outlined earlier in this document. So, in many cases, there is not a lot to say on any specifics.

```
================================================================================
---- Welcome, root
---- What would you like to do?
'A': INSERT New data entries
'B': UPDATE Already existing entries
'C': DELETE From existing entries
'D': Generate a report of a dataset

'Q': To exit the utility
Type 'info' for information on the accessible data tables.
> info
You chose: info
The list of tables is:
certificates, challenge, challenge_event, challenge_skills, contact_info, co
q, score, skillmaster, skills, trainee
If you want to view a table's list of parameters, enter its name.
Otherwise, press ENTER to return.
>

================================================================================
---- Welcome, root
---- What would you like to do?
```

Figure 2: A depiction of the main body of the program as it runs.

First of all, an introductory line of print statements outlines the possible actions that a user can take. It then prompts the user to choose an input. If the user's input does not match any of these queries, then the main body repeats after displaying a quick error line.

Inputs A-D call their respective functions from within their bodies of text. Other than prompting for the inputs that will be used in the functions to generate SQL queries, not much more is of note from a programming perspective. Please read the "Quick Start" guide that should be included with this submission for more information from a user's point of view.

Quickly, however: the function "info" will call get_paramnames after outputting the contents of the db_tables tuple. This is so that a user can see the table list and the columns for each of the tables without making a query directly.

And, "q" breaks out of the loop, as one would expect. After this, the connection with the MySQL database is closed, and the program terminates.

6.d. Possible Improvements to the UI

There are a number of things that could immediately be improved upon relating to the UI, which for some reason or another could not be reached. For the most part, I feel that as a notebook the Python code runs incredibly well.

One thing I would like to be able to change, would this project be going onwards with multiple iterations, would be to migrate the code from Jupyter notebook to a different Python client. Because the majority of the program exists in a large While statement, I feel that the notebook tends to hold back the potential of the UI to function properly. If it existed solely in a terminal interface, for example, I think that the program would be a lot easier to use. In a notebook, it tends to jump around a lot, and can be a bit disorienting to run through multiple lines of code all at once.

Furthermore, the fact that it seems very hard to catch exceptions and errors that occur with inputting SQL statements incorrectly is a bit counter-intuitive to usability. If the user inputs a query even slightly incorrectly, then the whole program terminates suddenly. I don't know if this disrupts connections in some bad way, but if there were a way to mitigate this? It would definitely be worth implementing.

Another thing would be to refine the functionality of the database triggers. Like I said in the previous section, there were some bugs that might prevent the database from being completely functional. The main one I foresee at this point is that certificates may still be earned even if the score to attain them is not quite high enough. However, other than that it works remarkably well.

All in all however, especially given the obstacles encountered throughout production, the database and its client run spectacularly well. I am confident that this product as a whole fulfills the requirements set forth in the original iteration of the requirements in the introduction to this report.