COURSE PROJECT 1: Project Report

Luis Martinez Morales

A. One page Description

The proposed database application is going to help manage and maintain the academic structure of a college. Its key features include tracking departments, courses, instructors, students, and their enrollments. Specifically, this will follow the rules outlined in the article on lovelycoding.org. As per the business rules:

- **Departments:** Each department offers multiple courses and is headed by one instructor.
- **Instructors:** Each instructor belongs to a single department and can teach multiple courses. An instructor can head at most one department.
- **Courses:** Each course is assigned to a department and is taught by one instructor.
- **Students:** Students can enroll in multiple courses, and each course can accommodate many students.

Unique aspects:

One unique aspect is that is important is ensuring that each department has exactly one head instructor. This constraint is a 1:1 relationship.

Another condition that needs to be upheld is restricting each course to a single instructor while still allowing instructors to teach multiple courses.

Finally, it will be important to manage many-to-many relationships between students and courses through the use of enrollments.

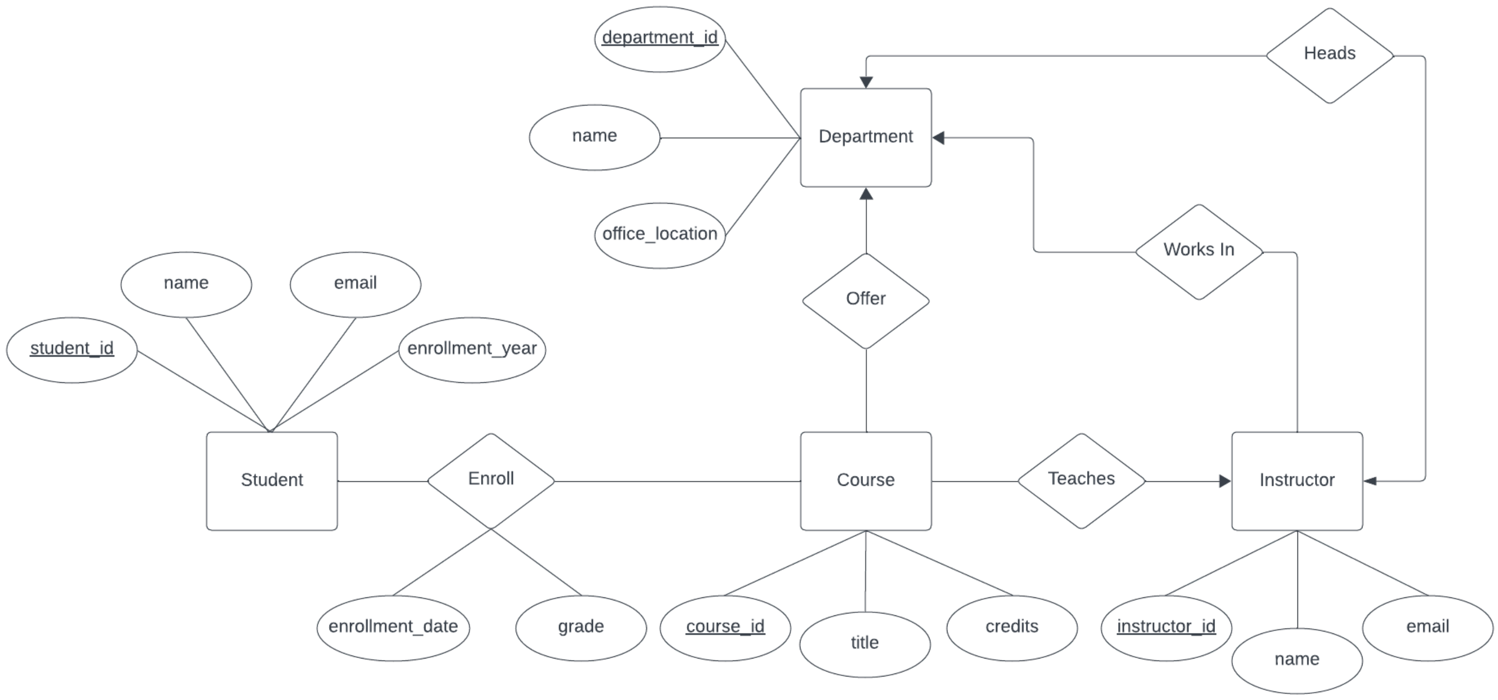
Data source:

- For my data source, I will rely on fabricated data, as this will allow more flexibility and control over the data. Specifically, Python has a library that is useful for data generation.

Challenges:

- While the data will be fabricated, the referential integrity constraints must be upheld. Inserting and deleting entries will be done carefully to maintain these constraints.
- Since the data is fabricated, it is also important to avoid duplicate keys for entities such as courses, instructors, or students. This can be implemented through the use of checks to ensure data integrity.

B. E/R Diagram



There are 5 main entities in this database, each has relevant attributes:

1. **Department:** department_id, name, and office_location
2. **Instructor:** instructor_id, name, and email
    a. Foreign key: department_id
3. **Course:** course_id, title, credits
    a. Foreign keys: instructor_id and department_id
4. **Student:** student_id, name, email, enrollment_year (the year they enrolled at the university)
5. **Enrollment:** enrollment_date, and grade (final grade in the course)
    a. Foreign keys: student_id and course_id

The relationships between the entities are defined based on the description in part A:

- **Works In:** A department can have many instructors, but each instructor belongs to a single department.
- **Heads:** Each department is headed by a single instructor, and an instructor can in turn only head at most one department.
- **Offers:** A department can offer multiple courses, but each course can only belong to a single department.
- **Teaches:** An instructor can teach many courses, but each course is taught by only one instructor.
- **Enrolls:** A student can enroll in many courses, and a course can have many students.

The E/R diagram accurately depicts the college application described in part A. This is because my E/R diagram shows that each department can have many instructors through the Works In relationship. Additionally, each instructor belongs to a single department. Since only a single instructor can Head a single department, the 1:1 constraint is enforced.

Similarly, the Course entity is correctly displayed on the E/R diagram because each course is taught by a single instructor. The Course entity also has important attributes such as title and number of credits to distinguish courses form one another.

In order for there to be a many-to-many relationship between students and courses, the enrollment entity is created. The enrollment entity allows for students to enroll in multiple courses and in turn courses can have multiple students in them. Enrollments have attributes such as enrollment_date which indicate when the student enrolled in that certain class.

Finally, referential integrity is upheld through the correct use of foreign keys. As a result, the relationships are valid and should be consistent with the business rules.

C. Table normalization

This relational database schema that is generated from the E/R Diagram is as follows:

- Department (department_id, name, office_location)
- Instructor (instructor_id, name, email, department_id)
- Course (course_id, title, credits, department_id, instructor_id)
- Student (student_id, name, email, enrollment_year)
- Enrollment (student_id, course_id, enrollment_date, grade)

The first step towards normalization is looking for any BCNF violations:

- **Department:**
    - Functional Dependencies:
        - department_id -> name, office_location
    - *This table does have any BCNF violations since department_id determines all the other attributes.*
- **Instructor:**
    - Functional Dependencies:
        - instructor_id -> name, email, department_id
    - *This table does not have any BCNF violations either.*
- **Course:**
    - Functional Dependencies:
        - course_id -> title, credits, department_id, instructor_id
        - instructor_id -> department_id
    - *This table has a BCNF violation because instructor_id is not a super key.*
- **Student:**
    - Functional Dependencies:
        - Student_id -> name, email, enrollment_year
    - *This table does not have any BCNF violations because student_id determines all of the other attributes.*
- **Enrollment:**
    - Functional Dependencies:
        - (student_id, course_id) -> enrollment_date, grade
    - *This composite key determines all other attributes; therefore, it is already in BCNF.*

The second step of normalization is to decompose:

The **Course** table has a BCNF violation due to the functional dependency instructor_id -> department_id.

1. Original course table:
    a. Attributes: course_id, title, credits, department_id, and instructor_id
2. Decompose into two tables:
    a. Course (course_id, title, credits, instructor_id)
        i. FDs: course_id -> title, credits, instructor_id
    b. Instructor (instructor_id, name, email, department_id)
        i. FDs: instructor_id -> department_id

Final BCNF schema:

- **Department** (department_id, name, office_location)
- **Instructor** (instructor_id, name, email, department_id)
- **CourseInformation** (course_id, title, credits, instructor_id)
- **Student** (student_id, name, email, enrollment_year)
- **Enrollment** (student_id, course_id, enrollment_date, grade)

The table normalization process improves table structure by reducing any redundant functional dependencies. For example, in the initial Course table, instructor_id functionally determined department_id, which was already stored in the Instructor table. This redundancy could lead to inconsistencies and update anomalies. By decomposing the Course table into a new table called CourseInformation and Instructor, this redundancy is eliminated, and it ensure s that department_id is only stored in the Instructor table. As a result, the database is efficient, easier to keep track up, and free of update anomalies.

Please note: While I decomposed Course into CourseInformation and Instructor, I will rename the CourseInformation table back to Course for simplicity and clarity during the coding phase. The shorter name makes it easier to work with and aligns better with the overall design of the database.

Considering Third Normal Form:

A table is in 3NF if is in BCNF and there are no non-prime attributes that depend on other non-prime attributes. Currently the schema is already in BCNF, and there are no transitive dependencies. Since the tables are already in BCNF and there are no transitive dependencies, they are also in 3NF. Therefore, no further normalization is needed for 3NF.

Considering Fourth Normal Form:

Similarly, a table is in 4NF if it is in BCNF and has no non-trivial multi-valued dependencies. In this schema, all the tables are already in BCNF, and there are no multi-valued dependencies. Since the tables are already in BCNF and have no multi-valued dependencies, it is safe to say that they are also in 4NF. As a result, no further normalization is needed for 4NF.

D. Data Collection

In order to populate the database with realistic data, I worked with the "Faker" library on python, and I used the "random" module. It generates synthetic data for all the tables within the database. This data is then saved as CSV files, and I then imported them into PhpMyAdmin using the import feature.

Key features of the Data Generation Process:

1. The Faker library was used to generate fake data that was appropriate for each attribute of the tables. For example, department names were generated using fake.unique.job().
2. In order to ensure that all primary keys are unique, I did sequential numbering while also tracking the generated IDs. Additionally, since I was generating a lot of data, the unique values were exhausted, and I had to eventually append numbers to the end of course titles to keep primary keys unique. The last thing I did in terms of maintaining uniqueness was to keep track of pairs such as student_id and course_id in a set called unique_enrollment_pairs.
3. To preserve referential integrity, I ensured that foreign keys reference valid primary keys in other tables. Another restriction I kept in mind was the fact that each department has a single head instructor. In turn an instructor cannot be in charge of more than one department.
4. When it comes to deciding the range of certain attributes such as enrollment year, credits, and grades I decided to limit it to a realistic range. For example, I made the earliest enrollment year 2020 since that is when I started my undergraduate degree. I also added the 'Q' grade for Q drop because I felt that that is one of the grades that appears often rather than 'I' for incomplete.
5. The final output of the python script is csv files. These csv files are easy to manage and running the python script overwrites the csv files.

Ensuring Data Integrity:

As previously mentioned, there were many steps that I took in order to ensure data integrity. I made unique primary keys, created realistic unique attributes, kept track of unique relationships, and preserved referential integrity. The business rules were something I referenced and kept in the back of my mind throughout this entire process.

References:

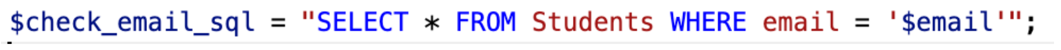
1. Faker documentation:
   a. https://faker.readthedocs.io/en/master/
2. Fake.data_between():
   a. https://faker.readthedocs.io/en/master/providers/faker.providers.date_time.html
3. Python random Module:
   a. https://docs.python.org/3/library/random.html

E. User Interface

The user interface for the college database is built primarily using PHP and HTML. Its design is simple yet straight to the point and it allows users to interact with the database without needing know anything about SQL or how to write it.

1. **The homepage/landing page is index.php**
    a. The main page is the main navigation hub, and it provides access to the rest of the website
2. **The user is able to view data if they choose to through the first two available options**
    a. This includes view_students.php and view_courses.php, these links primarily use the SELECT feature to retrieve the table information of students and courses.
3. **The user also has the ability to input data through the third and fourth option**
    a. The forms which are featured on add_student.php and enroll_student.php go through add_student_process.php and enroll_student_process.php which includes a check to ensure that the data the user has entered is valid. While the database already has check in place that don't allow invalid data such as duplicate keys to be added, I thought it would be better to stop that from even happening and instead letting the user know if they made a mistake.
4. **Lastly the user has the ability to view data insights through the first option**
    a. Here, the course_statistics.php provides helpful information with the help of the Join and Group by SQL query in order to calculate the number of students enrolled in each course.

Here is how each of those functions are implement in SQL:

- **view_students.php:**
    - the SQL code for this is quite simple and it is:
        - SELECT * FROM Students;
    - The query retrieves all of the entries in the Students table and displays them in a table format for the user.
- **view_courses.php:**
    - Similarly, the SQL code for this query is:
        - SELECT * FROM Courses;
    - The query retrieves all of the entries in the Courses table and displays them in a table format for the user.
- **add_student_process.php:**
    - The check implemented in SQL code in order to see if a user with the same email has already been registered is as follows:
        - 

```
$check_email_sql = "SELECT * FROM Students WHERE email = '$email'";
```

    - The point of this query is to block the user from adding a student to the Students table who is already registered.
    - The SQL code used to insert a valid entry to the Students table is:
        - INSERT INTO Students (name, email, enrollment_year)
          VALUES ('$name', '$email', $enrollment_year);

- **enroll_student.php:**
  - The user is allowed to enroll a student that already exists in a course that already exists in the college database.
  - For that reason this page uses both a simple SQL query:
    - SELECT * FROM Students;
  - And a while loop:

    -
      ```
      while($row = $result->fetch_assoc()) {
          echo "<option value='{$row['student_id']}'>{$row['name']}</option>";
      }
      ```
  - The end result being a dropdown with every Student already in the database. The same process is done to retrieve the Courses in the database and generate a dropdown.
- **enroll_student_process.php:**
  - Once the user has the appropriate values to insert into the Enrollments this SQL query is ran:
    - INSERT INTO Enrollments (student_id, course_id, enrollment_date) VALUES ($student_id, $course_id, NOW());
  - *Note: NOW() returns the current date and time. Since the enrollment date only retrieves the date, only the today's date will be used as the enrollment date. This makes it easy for the user since they don't have to waste time entering the date manually. The downside being that the current day is the day that they registered for the course which mimics real life.*

References:

1. Selecting attributes from the table and then displaying them:
   a. https://www.w3schools.com/php/php_mysql_select.asp
2. NOW() functionality:
   a. https://www.w3schools.com/sql/func_mysql_now.asp

F. Project Source Code

https://github.com/LuisM-M/DatabaseProject1.git

G. Discussion

I took the undergraduate database class about a year ago, where we had a similar project using PHP and MySQL. Because of that, I was somewhat familiar with the code and tools like phpMyAdmin. However, I had no prior experience using the Python connector or hosting a project on a platform like Heroku.

At the beginning of the project, I knew I had to create a Heroku account as soon as possible because there would be a delay between applying for student credits and getting approved. I applied within the first week or two of the project being assigned and started working on choosing a database project. I decided to go with a college-related database since I felt I understood that domain well. I created my E/R diagram using Lucidchart and proceeded with normalization. Normalization wasn't too difficult, and it made a lot more sense after completing the first homework.

The first major hurdle I faced was generating data without duplicate keys. I spent a few days trying different methods and implementing checks using if statements. The real test for duplicate keys came when I tried importing the CSV files into phpMyAdmin. If there were duplicates, phpMyAdmin would immediately flag the error and stop the import, letting me know that not all entries were imported because duplicate keys are not allowed. I then had to carefully delete all my tables and regenerate the data from scratch. Since there were referential dependencies, I also had to ensure that I imported data into the tables in the correct order to avoid errors. This process was tedious, but when I finally got tables with no duplicate keys, I was overjoyed.

Two weeks had passed, and I still hadn't heard back from Heroku, so I created a support ticket. They approved my account almost immediately after that. By this point, I had finished writing the Python scripts to generate data and produce CSV files. I had also started working on the user interface locally.

As I mentioned earlier, I had some prior experience with PHP, but I had to relearn it to write functional code for this project. This was a bit of a challenge, but it helped me refresh my skills.

Later on, I ran into another hurdle: setting up Heroku. Heroku works well, but I had to download several applications to get it running. I installed Heroku CLI, MySQL Workbench, and other tools like MySQL, Heroku, and Composer through the terminal. One small issue I encountered was that the latest version of MySQL wasn't compatible with the JAWSDB extension from Heroku, so I had to downgrade my MySQL version to make it work.

After some tinkering, I finally got Heroku to work, and it was fun accessing my database through Heroku and deploying the project. I was able to connect my GitHub repository and deploy the project that way.

However, I faced another challenge: Heroku didn't allow me to insert data into the database. I looked into alternatives and even created a support ticket to resolve the issue. I tried using PostgreSQL as an alternative, but that didn't work either. As a result, I decided to submit the project as a local website instead.

Overall, I learned a lot about databases through this project, and I have several important takeaways. It was a great learning experience, and I feel more confident working with databases, PHP, and deployment platforms like Heroku.