Hospital Appointments Manager

DSCI 551: Final Project Report
Distributed Database System

GitHub Repository
Video
Spring 2024

Tomine Bergseth
Omar Alkhadra

Table of Contents

Introduction

We developed a distributed relational database system designed for a hospital to manage its appointment data. For our project, we chose to use MySQL software. A relational database is well-suited for hospital data as both structure and data integrity are valuable components when managing sensitive data. Our data is stored in one of two databases based on the hash value of the DepartmentID to distribute data evenly. Each database has six tables with functionalities to add, delete, modify, and retrieve data. We have developed our own synthetic dataset, which has data for three departments worth of data - Cardiology, Pediatrics, and Radiology. For our project's two user interfaces, we have a command line UI for database managers and a web-based application designed for hospital receptionists. In the sections to follow, we will describe our initial proposed planned implementation, and detail our final implementation including our ER model, the architecture design, functionalities, tech stacks, data integrity, and hashing logic.

Initial Implementation Idea: Project Proposal

In our planned implementation section in the project proposal, we wrote: "For our project topic, we want to create a distributed database system designed for a hospital. Specifically, we want the database to manage appointment data. Consequently, our distributed database system will include data for appointments, departments, employees, and patients data. Due to the nature of this data, we will store the data in SQL databases; relational databases will help make the data easy to access and utilize for end users. Further, we will work in Python for all of our code and use the SQLAlchemy library for our SQL database.

We plan to have four databases, one for each of the following: Department, Appointments, Employees, and Patients. To add data, we will create separate functions to add data to each database. Additionally, for our employee and appointment databases, we will create hashing functions to partition the data given the large data size. For our employee database, we will partition by whether an employee is medical or non-medical staff. Secondly, for our appointment database, we will hash by department. Departments will list departments within the hospital and the employees that work within it. We plan on doing three departments, namely cardiology, radiology, and pediatrics. Appointments will contain appointment data as well as record the receptionist, patient, and doctor associated with it. Similarly, the employee database will contain information about the hospital employees and the patient database will contain information about patients. Data types will consist of strings (names, departments, etc.) and integers (IDs, dates, times, etc.).

Since medical data is confidential and sensitive, we decided to create synthetic data to assess and test our database stores. We will begin with enough data to set up the infrastructure of the databases. Afterward, we will test the addition, deletion, and modification of data through the user interfaces.

To demonstrate how we plan to structure our SQL databases, we have created the following tables:

Department Database:

Department name (Primary Key)	Number of employees	Total number of rooms/beds	Number of scheduled rooms/beds

Employee Database (Medical):

Employee ID (Primary Key)	Last name	First name	License #	License Status	Department (Cardiology, Radiology, Pediatrics)	Specialty

Employee Database (Non-Medical):

Employee ID (Primary Key)	Last name	First name	Department (Receptionist, Security, Cleaning)

Patient Database:

Patient ID (Primary Key)	Last name	First name	Scheduling State (scheduled, recently visited, dormant)	Insurance (Y/N)	Past procedures	Additional Notes

Appointment Database (Cardiology):

Appointment # (Primary Key)	Receptionist	Patient Name	Doctor Name	Department	Appointment Time	Notes

Appointment Database (Radiology)

Appointment # (Primary Key)	Receptionist	Patient Name	Doctor Name	Department	Appointment Time	Notes

Appointment Database (Pediatrics):

Appointment # (Primary Key)	Receptionist	Patient Name	Doctor Name	Department	Appointment Time	Notes

There will be two User Interfaces (UI): one for the database managers, and one for the hospital receptionists. We plan on making our UI for database managers a Command Line Interface and then as per the requirements the web-based application for hospital receptionists will be a web-based application. For both UIs, we will have an employee ID login to ensure the security of our database and confidential information. We will then create a function to modify, add, delete, and filter/retrieve data through the command line. The database managers will have access to modifying all data across our distributed database system.

On the other hand, our web application for receptionists will allow the receptionist to view all the data; however, they are only able to add, modify, and delete appointments and patient data. We will also create a filtered retrieval query for the web-based application so they can view departmental or employee data as well.

In terms of the more technical aspects of the web application, we plan on using Streamlit as our framework and make a multiple page application. There will be one page per category of data, where under the patient and appointment databases we will design a user-friendly menu option for the receptionist to modify, add, or delete appointments or patients.[1]"

Final Implementation

While the main idea and many technical details remain the same from the proposal, we have made several changes to optimize our distributed database system as we have learned about relational databases and data distribution throughout the semester. The remainder of our report will detail our final implementation of the project. We decided to not include code screenshots in our report and instead do the optional video.

ER Model

Since the proposal, we have made some slight changes to our tables. To best demonstrate the structure of our different tables and attributes, we made an ER model for our project.

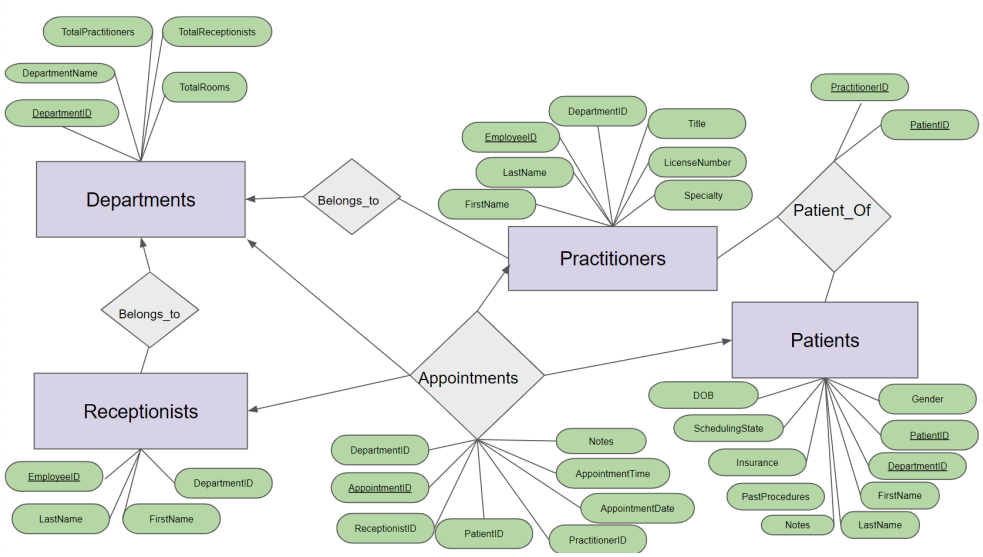

Figure 1.0 Project ER Model

Looking at Figure 1, we see six tables in total. Four of these tables - Departments, Receptionists, Practitioners, and Patients - are depicted as entities in our model, while Appointments and Patient_Of are depicted as relationships. The appointments table has several unique attributes; however, as it also serves as a relationship between the four entities, we decided to depict it as a

[1] Proposal: Hospital Appointments Manager. Spring 2024. Bergseth, Tomine, Alkhadra, Omar, & Patel, Mira.

relationship rather than an entity in the model. The two additional “belongs_to” relationships are for demonstrative relational purposes in the ER model only and are not actual tables in our project. For each table, our primary keys are underlined. Note that for patients and Patient_Of, we have a composite primary key. Relationship types are denoted according to lecture materials, where an arrow indicates that an entity and/or relationship is the “one” in a relationship and a simple line without an arrow indicates that the entity and/or relationship is the “many”. For example, the relationship between departments and appointments is one-to-many, given there can only be one department per appointment; however, there can be many appointments per department. Patient_Of is our only table that is purely an association table for the many-to-many relationship between Patients and Practitioners. Finally, we have also used SQLAlchemy’s relationship feature in our code as an efficient way to retrieve data from relationships. Additional constraints, such as foreign keys can be found in the data integrity section.

Functionalities

For all of our tables except Patient_Of, we have add, delete, modify, and get functions. The table below shows all the available functions by table:

Departments	Patients	Practitioners	Receptionists	Appointments	Patient_Of
Add department	Add patient	Add practitioner	Add receptionist	Add appointment	Get patients of
Delete department	Delete patient	Delete practitioner	Delete receptionist	Delete appointment	Get practitioners for
Modify department	Modify patient	Modify practitioner	Modify receptionist	Modify appointment	—
Get department	Get patient	Get practitioner	Get receptionist	Get appointment	—

As we have used SQLAlchemy for our database code, all tables are associated with a class where all functions are defined as class methods. The paragraphs below describe the parameters and user input for each type of function available.

All of our add functions work to insert data into the correct table within a SQLAlchemy class. The function parameters for all of our add functions is an input dictionary, which is provided and named as json_dict in the command line code and provided through the receptionist filling in the appropriate fields in the web-based application. In addition to the input dictionary, after verifying that the input includes all required attributes, the add functions also take the session object for the appropriate database as a parameter. Note that the session object parameters are not provided by the user. Rather, they are created based on the hashing logic described later in the report.

For all modify functions except for department modification, the functions take both session objects, a filter attribute dictionary, and a new values dictionary as parameters. The modification functions operate in bulk depending on the attributes specified in the filter dictionary. For example, modifying an attribute such as changing a job title from “Doctor” to “Medical Doctor” would happen in bulk across both databases if no other filtering requirements are provided, whereas if you wish to update for only a single practitioner, you could filter by their Employee

ID. Department modification is an exception to this as we have restricted it to only modify a single department at a time. It takes the appropriate session object, the ID number of the department to modify, and a new values dictionary with the attributes and corresponding values to insert.

We discuss certain limitations on updating the DepartmentID for some tables in the data integrity section; however, if the DepartmentID is modified for the practitioners or receptionist tables, the code will call the hash function on the new DepartmentID value and check if the hash value matches the old hash value. If it does not match and the modified row now belongs in the other database, the code automatically deletes the row from the current database and adds it to the correct one.

Delete functions are designed similarly to our modification functions. They can also operate in bulk depending on the filter requirements provided by the user and take both session objects as well as a filter dictionary where the user would specify the attributes and corresponding values for the rows they wish to delete as parameters. Again, delete_department is an exception as a department is very unlikely to be deleted and should only be deleted with great care. Delete_department takes the appropriate session object and the id of the department to be deleted as parameters.

Our get functions retrieve data according to filter requirements provided by the user and work the same for all tables except for Patient_Of. To retrieve data as needed from both databases, these functions take both session objects and an optional filtering dictionary. If no filtering dictionary is provided, the get functions default to retrieving all the data in a table. When a filtering dictionary with the attributes and values the user wishes to retrieve is provided, the function returns only the filtered data. Where there is a foreign key referencing the ID attribute in the departments, practitioners, or patients tables, the name attributes - i.e. the department name, first, and last names of people - are also joined into the get result. Additionally, all of the get functions return a total count of either all rows in the table or the rows that meet the filtering requirements.

For the get functions for Patient_Of, the parameters are both session objects, the id of the patient in get_practitioners_for or the practitioner in get_patients_of, and an optional list of attribute names. The latter is a list rather than JSON in the command line and allows the user to simply view for instance only the first and last names in the relation rather than all attributes if desirable.

While the database managers have access to all functionalities through the Command Line UI, we have limited functionality available to receptionists in the web application. Receptionists have access to all get functions; however, they can only add and modify data in the appointments and patients tables and they can only delete data from the appointments table. These differences are due to the practical nature of a receptionist's job (i.e., they should not be responsible for the add/modification/deletion of employee data, receptionist data, or department data). The usability of the available functions for the receptionists resembles those for the database managers, as explained above.

Architecture Design: Flow Diagram

Our flow diagrams explain the architecture design of our database system. First, the user will provide input either through the command line in the form of JSON objects or buttons in the web application interface. For the command line, the code then checks and verifies the length of the input, and based on the length, it will enter different if statements to appropriately extract the arguments. Across the command line function calls, user input varies in length between three and four, with different logic implemented for differences between functions within those lengths for extracting arguments such as the JSON string. For the web-based application, this is more automatic as the user navigates to the correct table and function tab to insert data through buttons.

After extracting arguments and setting the correct variables, the code checks if "DepartmentID" is in the input. This will be the case for all add statements as well as modify_department and delete_department, and is verified by the add function calls later in the code. For add functions, modify_department, and delete_department, once the Department ID is extracted, the code calls our hash function, which determines which database to store the input in. It will then create the appropriate engine based on the hash result. However, as delete, modify, and get functions to work in bulk across both databases, the Department ID may or may not be included in the input for these functions. Therefore, in those cases, the code creates engines for both databases.

Following the creation of the appropriate engine(s), the command line code will enter a series of if statements to check for a match to function calls within the six classes available: Department, Appointment, Reception, Practitioner, Patient, and Patient_Of. For the web application, this is already determined by the table and function tab the user is on. The code will then try to call the function specified with the extracted parameters. To do so, it first verifies that certain input requirements are met. This is most significant for add functions, where it will verify that all the required input attribute names are included as keys in the JSON object provided in the command line or the input in the web application. The function is then executed, and the command line and web application will print a statement that indicates whether it was successful. If unsuccessful, it also prints an appropriate error message. If the function call is a get function, it will print the filtered data requested by the user.

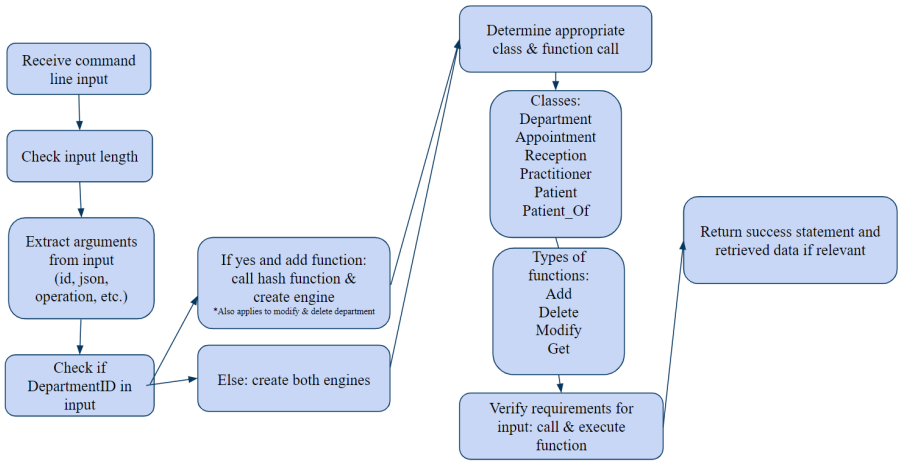

Figure 2.0 Command Line UI Flowchart

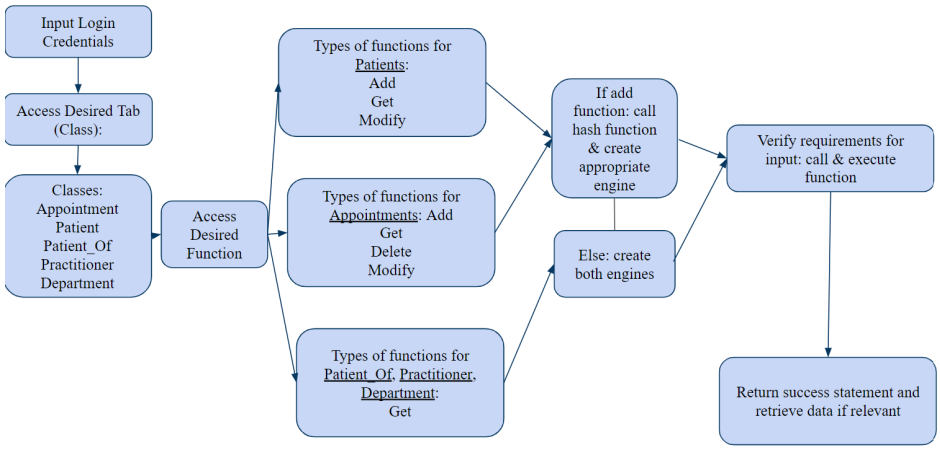

Figure 3.0 Web Based Application FlowChart

Tech Stacks

The two tech stacks in Figure 4.0 demonstrate the technologies and packages used for both the Command Line UI and the Web Application. These have remained the same since our planned implementation, where we have used SQLAlchemy for our database code and Streamlit for our web application code. We added that we would use MySQL as our database system in the mid-progress report. While we have used MySQL locally, an organization with web access to MySQL could use our code and share the data across users so it would work well across hospital departments and reception users.

Figure 4.0a Command Line UI Tech Stack

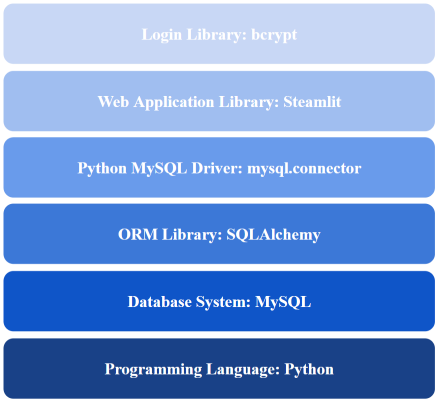

Figure 4.0b Web Application Tech Stack

Data Integrity: Additional Constraints

One of our focuses for our project has been to ensure data integrity in our database system. To most efficiently demonstrate additional constraints in our logic, we made another ER model, which can be found in Figure 5.0 below, where we used borders to color code different additional constraints as best we could. Foreign keys correspond to pink borders, unique constraints correspond to red borders, number of digit constraints correspond to orange borders, not null constraints correspond to royal blue borders, teal borders correspond to features that have a

default value and are automatically updated through event listeners, and forest green corresponds to auto increment.

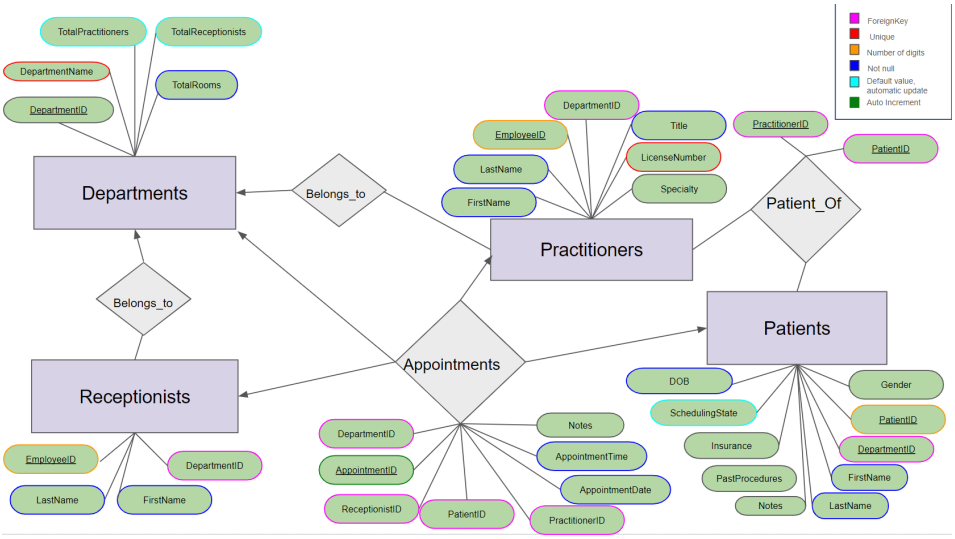

Figure 5.0 ER Model, Additional Constraints

To further explain the model, our only auto-incremented value is the "AppointmentID" for the Appointments table. Across the tables, we wanted our primary key to have an intuitive value. For departments, a hospital may have around 10-20 departments. Therefore an incrementally increasing digit primary key is logical, intuitive, and easy to remember. For our sample data, we had three departments - Cardiology, Pediatrics, and Radiology - with IDs 1-3. For Patients, we have a four-digit PatientID as the primary key, where we intend for this to be the last four digits of someone's Social Security Number (SSN). For this to be consistent, we also have a constraint that checks that the ID is four digits; however, should a hospital wish to use the full SSN instead, this constraint could easily be adapted. For Practitioners and Receptionists, we have a 6-digit EmployeeID. It is common practice to have an employee ID associated with each employee, which also makes these primary keys intuitive. In our sample data, we start each practitioner employee ID with 1 and each receptionist employeeID with 2; however, since our proposal, we split these into different tables so our logic no longer depends on this differentiation in EmployeeIDs. That being said, since the focus of our distributed database system is appointment data and we expect this to have the largest scale, it is less intuitive to have to remember what the last ID digit of an appointment was or have to check every time before you add an appointment which is why we decided to make it auto-incremented. It is also noteworthy that for appointments we have an additional composite unique constraint not shown in the model above that includes PractitionerID, AppointmentDate, and AppointmentTime to make sure that there are no duplicate appointments and practitioners are not double-booked.

Our foreign keys, denoted by pink borders, are all defined with a cascade on update and on delete. DepartmentID is a foreign key referencing the DepartmentID in our departments' table in our receptionists, appointments, patients, and practitioners tables. Additionally, in the appointments table, PatientID references the PatientID in patients, PractitionerID references the EmployeeID in practitioners, and the ReceptionistID references the EmployeeID in receptionists. Beyond the foreign and primary keys, we added not null and unique constraints for appropriate

attributes and we have default values for attributes that are updated through event listeners automatically rather than by the user. Please note that this also applies to the entire Patient_Of table although we could not fit it in the model above.

Event Listeners & Limitations On Modifications

Another component of our data integrity design is our use of event listeners to automatically update certain attributes. For departments, we have event listeners that update the TotalReceptionists and TotalPractitioners columns when a practitioner and receptionist is inserted, updated, or deleted. In our patient's table, we do the same for SchedulingState by department, where unscheduled is the default value and means that the given patient has not been scheduled for any appointments yet in that department. The event listeners listen to inserts, updates, and deletions in the appointments table and update the value to scheduled once there is an appointment scheduled for the patient in the given department. Finally, our entire Patient_Of table is automatically updated through event listeners that listen for inserts, updates, and deletions to appointments. Once a PatientID and PractitionerID are added to the same appointment, they will be added to the Patient_Of table. The Patient_Of table is our only table that doesn't include the DepartmentID which we are hashing on; however, because of the use of event listeners, the Patient_Of relation will be added to the same database that the appointment row is added to, which will always be hashed by DepartmentID.

In addition to our use of event listeners and constraints to ensure data integrity, there are also certain limitations on values that can be modified as well as the abilities of a receptionist in the web application. All automatically updated values mentioned above cannot be modified manually as that would interfere with the logic of event listeners. Further, DepartmentID cannot be modified in the departments, patients, and appointments tables and AppointmentID cannot be modified in the appointments table. The reason we prevent modifying the AppointmentID is that it is auto-incremented. For the DepartmentID in departments, this is a very expensive modification as it would potentially require moving all the data belonging to a department to another database. Further, it is a very unlikely and unnecessary operation as the departments in a hospital likely will range between 10-20 so it is easy to delete and add a new department with appropriate ID if needed, although we do not think this would ever be the case. The reason we prevent modification of DepartmentID for patients is that we have a composite primary key for this table that requires a patient to be added once for each department. Therefore, this should be an addition, not a modification. Finally, we prevent modification of DepartmentID for appointments, as each ID that is referenced as a foreign key in the appointment will be associated with the department. As the receptionist, practitioner, and patient all belong to that department, it is not a value that should be modified and modification would cause foreign key violations. All of these attributes would need to be updated for the DepartmentID to be updated. While unlikely, an appointment should be deleted and added with all updated values if a department change is needed. The receptionists who add the appointment data are also hired by the department and would likely only add appointments for the department where they work.

Hashing Function & Data Distribution Logic

Another component we have put a lot of thought into throughout the semester is our hashing function and data distribution logic. Our hashing function is simple: DepartmentID % 2. In our

planned implementation section, we mentioned that we were going to hash by department. While we ended up hashing on DepartmentID, we considered several options. Initially, in our proposal, while not specified, we intended to hash on department name; however, we realized that to normalize our tables and avoid redundant attributes, hashing on DepartmentID, the primary key of departments which is referenced as a foreign key in most of our tables, is more efficient.

After our demo presentation, we were asked to explain our hashing logic more in our final report with the question of whether hashing on something such as PractitionerID would have been more efficient. While we understand the question, as we have considered different options including PractitionerID, there are two main reasons why we landed on DepartmentID. Firstly, due to the nature of our hospital data and tables, DepartmentID is an intuitive way to partition the data. As most tables reference the DepartmentID as a foreign key, hashing on DepartmentID distributes the data in a way such that we do not risk violating any data constraints where a referenced row may exist but does not exist in the current database. We found that while you can reference other databases in a table in MySQL, it is not common practice to do so and does not work well if you are referencing a table both in the current database as well as the other one simultaneously as either one will always throw an error with a foreign key violation.

The reason why PractitionerID was brought up in the question, was because across all departments in a hospital some may have fewer practitioners than others. Hashing on PractitionerID may therefore provide a more guaranteed even distribution of appointment data; however, this would cause a lot of foreign key violation issues. Further, across all departments of a hospital, if there are 10-20 departments, the number of practitioners in each department would even out across all departments, leading to evenly distributed data. It is also noteworthy that while our sample data only includes three departments - Cardiology, Pediatrics, and Radiology - another reason behind our hashing logic is that it is easily scalable and will still work well when adding more departments. Arguably, the more it is scaled out, the more even the distribution of practitioners and appointments is likely to be.

Finally, as part of our hashing logic, we wanted to briefly explain the composite primary key in the patient's table. To hash on DepartmentID and make sure that the patient exists within the database the appointment would be created in, we found this to be the best solution. When trying to hash patients on a different attribute than DepartmentID, this caused several issues concerning foreign key violations. While the composite key may lead to some cases where a patient exists in multiple rows in a database, it ensures it exists in the right one, and as opposed to solutions such as having all patients in both databases, there will be no unnecessary duplicates. In the latter solution, a patient would always have existed in both databases for comparison. Finally, as a receptionist would be hired by a department at a hospital, and would likely only work with the data for that department, adding a patient once per department would work well in the real world.

Data Security

Due to the sensitive nature of hospital data, our project has made use of security measures for both the data managers and the receptionists. Access to the web-based interface is restricted to a select number of users (receptionists) who can log in using their unique username and password. For our implementation we have only two users, Omar and Tomine, however, this is easily

extensible to a real-life scenario, as the credentials can be updated within the UI code. The basis for creating logins is explored more thoroughly in our implementation video.

For the command line UI, the user - i.e. the database managers would have access to the code to interact with the command line. They either need to update the code with their MySQL login information at the top of the code in the global engine URL dictionary or there is a login feature that can be used. At the top of main in the hospital_db.py file, lines 1256-1260 can be uncommented to call the login function which allows the database manager to type in their MySQL username and password through the command line instead when calling a function; however, as the nature of the command line logic recalls the script each time a new function is called, this would require retyping in the username and password each time which can be time-consuming and is the reason why it is commented out and optional only. To use it, the global engine URLs definition at the top of the script should also be commented out.

Learning Outcomes

As both members of our team were new to relational database systems and web application development, we have learned a lot throughout this semester. Early in the semester, there were times when we made an implementation decision only to learn something new about that area of database management in class a couple of weeks later and come to realize that we needed to change it to make it more efficient. This challenge particularly showed up in attributes and our tables to make sure they are efficient, normalized, avoid redundancy, and ensure data integrity. Another main challenge we have worked through was what we should hash on as detailed in the hash logic section. While we believe our solution ensures both even distribution of data and data integrity, this is something we spent a lot of time discussing and had some trial and error with along the way.

Despite these challenges, overall we have learned a lot about SQL, SQLAlchemy, web-app development, relational database design, data distribution logic, and data integrity throughout this semester which we believe will be very helpful as we continue on our paths in data science.

Individual Contributions

All members of the group were responsible for the brainstorming and formulation of the project idea. Frequent discussions were made on progress and future steps during weekly meetings throughout the semester. Tomine was the team lead and was responsible for the database code, logic, structure, and code for the command line UI. Omar was responsible for the development, design, and code of the web-based application. All code can be found in our GitHub repository; Tomine's code can be found in hospital_db.py and Omar's code can be found in UI.py.

We also would like to note that, we initially had a third group member, Mira Patel, who helped in meetings during the early stages of the project. However, she decided to drop the class.

Conclusion

To summarize, this project sees the development of a distributed relational database system designed for a hospital to manage its appointment data. Our project uses MySQL, SQLAlchemy,

and Streamlit as the pillars of its structure. We designed two applications: a command line UI for the database managers using sys and JSON and a web-based application for the hospital receptionists using Streamlit. Within the database, we have a total of six tables with add, delete, modify, and retrieve functions. To ensure even distribution, we hash the data on DepartmentID. Finally, due to the nature of our project, we placed the necessary emphasis on data integrity and security throughout the structure of our databases.

Future Scope

For future development of this project, a few interesting improvements could be made. Firstly, we could converge the structure of our implementation to all be within a web-based application. This would mean both database managers and receptionists would use the web-based application UI; however, based on privileges assigned to each role, the UI would look different to each role when they log in. A possible extension such as this could include an interface for practitioners, where they would be able to see a calendar of all their appointments and perhaps fill out patient and appointment reports within the system. To do so, we would have to add additional tables and consider expanding our hashing function to add more databases if necessary.

Another future development could involve implementing file imports for large amounts of data. This development would aid in the initiation of the database to real-world hospitals. As many hospitals already have data, such as existing employment records, it would be more efficient for them to upload these existing files rather than input records one by one. Implementing a feature where data induction can be handled by reading CSV and/or JSON files would allow for the proper initialization of the appointment manager database in the real world.