Matthew McCarthy
Benjamin Bricker

CMSC 508 Database Project Proposal

Problem Statement:

The following proposal outlines the construction of a database that will be used for a basic hospital management system. Based on the needs of the hospital, a database will provide an efficient way to store and retrieve information related to their patients, employees, and the treatments provided. For example, a user of the database should be able to search a patient's medical record and see a detailed timeline of any treatments they have received. This will not only make retrieving information easier, but it will also provide the hospital with a seamless way to monitor its overall functioning.

Based on the relatively small size of the hospital, the main goal of this proposal is to address the core features of the database. Once that has been established, additional functionality can be added later. To construct this initial design, the following entities must be recorded in the database:

- Employee
- Doctor
- Nurse
- Patient
- Treatment
- Medicine
- Department
- Room

The *employee* entity will store information common to all employees, while the *doctor* and *nurse* entities will be specializations of *employee*. These specialized entities will keep track of information not common between a doctor or nurse. Using specialization will let the database record what type of employee interacted with a given patient. This is especially useful when trying to determine who treated a given patient.

In addition to recording employee information, patient information will be stored. This will include general information about the patient, but more importantly, it will be used to accurately record the treatments a patient has received, thus providing a medical history for any given individual. For the *treatment* entity, there will be one specialization entity (*medicine*). The purpose behind this design will help determine if a patient underwent a treatment and required some form of medication.

To maintain security, only authorized personnel will have access to the database. This will require each user to have their own login information for the system. For the sake of privacy, the number of individuals that have access to the database should be kept to a minimum. In this regard, only certain employees (e.g. doctors and nurses) and

administrators will be using the database. To keep the focus on the design of the database, the sample web interface (described in more detail in a later section) included for this project demonstrates a basic view that an admin could potentially utilize.

Employees will use the database to update information, such as the treatments provided to a patient on a given date. They will also be able to record new patient information if it's their first visit to the hospital.

Administrators will be able to monitor the entire database, allowing them to extract information regarding the hospital's performance. That information can then be utilized to gain summarized information, such as daily patient counts or average number of employees working on a given day.

Apart from the how the database will be utilized, there are several factors that must be considered to keep the system operational. Based on the guidelines listed so far, the following operations will need to be performed on a regular basis to maintain the database as a whole:

- Record the date a patient is billed for treatments
- Enter the date when an employee administers a given treatment
- New patients must have their information entered into the system
- Record the date when a patient is assigned to a room
- Update when a new patient is assigned to a room after it's available for use
- Update primary care physician for a patient if they were to change doctors
- New treatment IDs will need to be added for each patient that receives a treatment

Lastly, the following are some additional queries that can be conducted using the database (corresponding SQL code associated with each question is included). These will help provide context and give some examples as to the full functionality of the database. Note some of the queries utilize views that are described later in the *Database* section:

1) The names of all employees with a given specialty.

```
SELECT firstName, lastName
FROM employee
WHERE specialty LIKE "<specialty>";
```

2) The names of all doctors that competed their residency at a given hospital.

```
SELECT firstName, lastName
FROM employee e, doctor d
WHERE e.employeeID = d.employeeID AND residency LIKE "<hospital>";
```

3) All doctors that attended a given medical school.

```
SELECT firstName, lastName
FROM employee e, doctor d
```

```
WHERE e.employeeID = d.employeeID AND medschool LIKE "<med_school>";
```

4) A list of treatments that cost over X amount.

```
SELECT treatmentID, treatmentName
FROM treatment
WHERE cost >= <value>;
```

5) A list of all patients treated by a given employee.

```
SELECT ep.patientID, p.firstName, p.lastName
FROM employeePatientTreatment ep, employee e, patient p
WHERE ep.employeeID = e.employeeID AND ep.patientID = p.patientID
AND e.employeeID = <value>;
```

6) Names of all patients for whom a given doctor is their primary care physician.

```
SELECT p.firstName, p.lastName
FROM patient p, employee e, PCP
WHERE e.employeeID = PCP.employeeID AND p.patientID = PCP.patientID
AND e.employeeID = <value>;
```

7) The names of all employees that administered a treatment to a given patient.

```
SELECT ep.employeeID, e.firstName, e.lastName
FROM employeePatientTreatment ep, employee e
WHERE ep.employeeID = e.employeeID AND patientID = <value>;
```

8) The name of all medications administered to a given patient (on a given day)

```
SELECT ep.patientID, m.treatmentID, m.medName
FROM employeePatientTreatment ep, medicine m
WHERE ep.treatmentID = m.treatmentID AND ep.patientID = <value>;
```

9) A list of all treatments administered to a given patient.

```
SELECT treatmentID, treatmentName
FROM employeePatientTreatment
WHERE patientID = <value>;
```

10) All patients treated by a given nurse.

```
SELECT ep.patientID, p.firstName, p.lastName
FROM employeePatientTreatment ep, employee e, patient p
WHERE ep.patientID = p.patientID AND ep.employeeID = e.employeeID
AND e.specialty IN ("nurse");
```

11) All nurses assigned to monitor a given room.

```
SELECT m.employeeID, e.lastName
FROM monitor m, employeeData e
WHERE m.employeeID = e.employeeID AND roomID = <value>;
```

12) The names of all patients assigned to a specific nurse's rooms.

```
SELECT a.patientID, p.firstName, p.lastName
FROM monitor m, assigned a, patientData p
WHERE m.roomID = a.roomID AND a.patientID = p.patientID
AND m.employeeID = <value>
ORDER BY a.patientID;
```

13) A list of rooms a given department oversees.

```
SELECT r.roomID
FROM room r, located l
WHERE r.roomID = l.roomID AND l.deptID = "<deptID>";
```

14) The names of all employees that work in a given department.

```
SELECT e.firstName, e.lastName
FROM employeeData ed, employee e
WHERE ed.employeeID = e.employeeID AND deptID = "<deptID>";
```

15) The names of all employees that have been given a raise.

```
SELECT e.firstName, e.lastName
FROM employee e, salary_log s
WHERE e.employeeID = s.employeeID AND s.oldSalary < s.newSalary;
```

16) The average amount billed at the hospital.

```
SELECT avg(cost) as "Avg Cost"
FROM patientBill;
```

17) All employees with a salary greater than X amount.

```
SELECT firstName, lastName
FROM employee
WHERE salary >= <value>;
```

18) All patients that received a given treatment (on a given day).

```
SELECT p.firstName, p.lastName
FROM employeePatientTreatment ep, patient p
WHERE ep.patientID = p.patientID
AND ep.treatmentName = "<name>"
AND (ep.adminDate BETWEEN "<datetime1>" AND "<datetime2>");
```

19) The average amount billed to a patient for all treatments they have received.

```
SELECT patientID, avg(cost)
FROM patientBill
GROUP BY patientID;
```

20) The names of all doctors that have prescribed a given medication.

```
SELECT e.firstName, e.lastName
FROM employee e, employeePatientTreatment ep, medicine m
```

```
WHERE ep.employeeID = e.employeeID
AND ep.treatmentID = m.treatmentID AND m.medName = "<name>";
```

Conceptual Design:

The following section highlights the conceptual design of the database. This includes the ER diagram along with various integrity constraints. It's important to note that all attributes recorded for any given entity will not allow for null or negative values. Each attribute should follow the format described in the tables below.

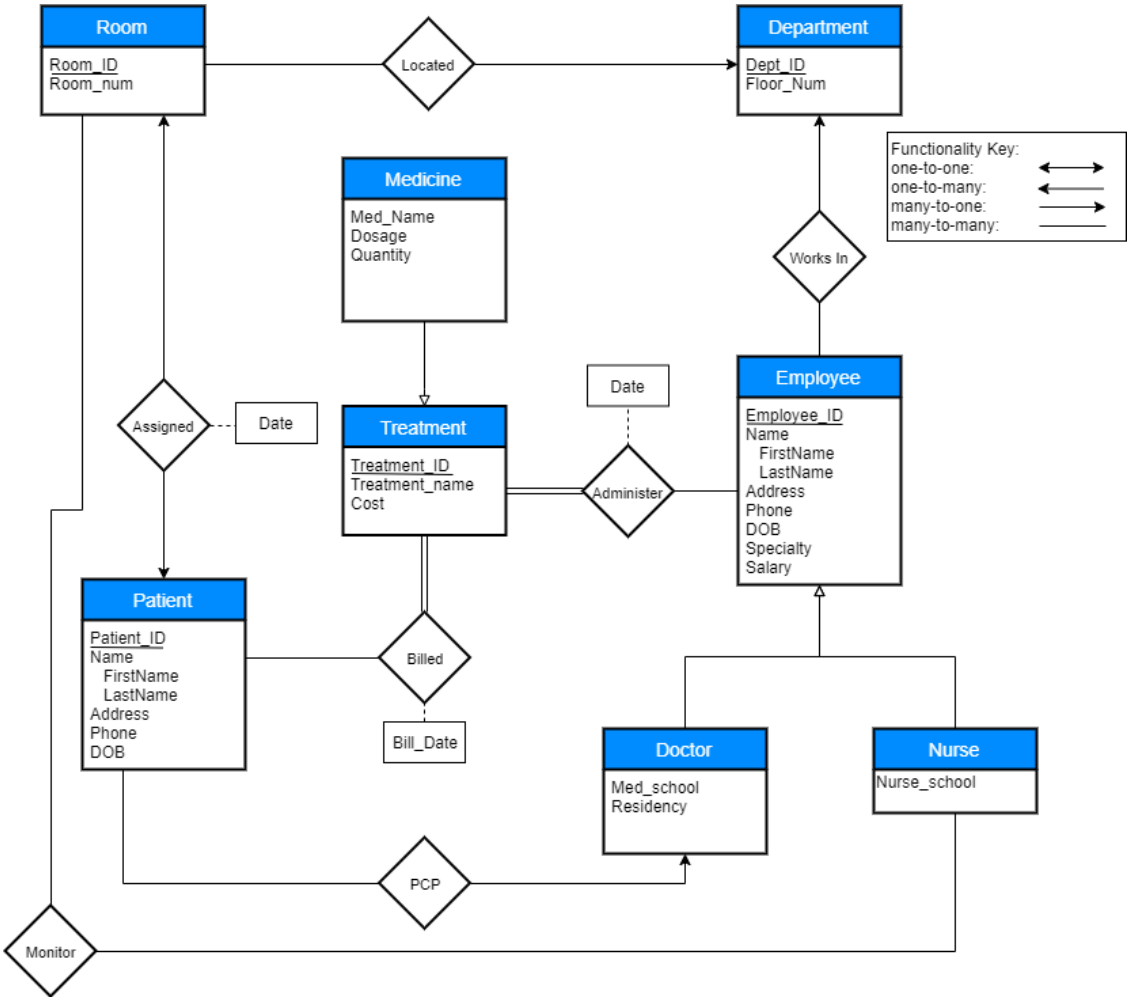

Figure 1: ER Diagram for Hospital

Table 1: Domain Constraints of Entities

Entity	Attribute	Domain
Employee	EmployeeID (PK)	Unique numeric value in the range of 000001-999999
	FirstName	String value
	LastName	String value
	Address	Variable length alphanumeric value. Contains a street num, street name, city, state, postal code
	Phone	10-digit numeric value (includes area code)
	DOB	Numeric value in the format Month/Day/Year (MM/DD/YYYY)
	Specialty	String value representing specialization
	Salary	Numeric value (must be whole #)
Doctor	EmployeeID (PK)	Unique numeric value in the range of 000001-999999
	MedSchool	String value. Name of school
	Residency	String value. Name of hospital
Nurse	EmployeeID (PK)	Unique numeric value in the range of 000001-999999
	NurseSchool	String value. Name of school
Patient	PatientID (PK)	Unique numeric value in the range of 000001-999999
	FirstName	String value
	LastName	String value
	Address	Variable length alphanumeric value. Contains a street num, street name, city, state, postal code
	Phone	10-digit numeric value (includes area code)
	DOB	Numeric value in the format Month/Day/Year (MM/DD/YYYY)
Treatment	TreatmentID (PK)	Unique alphanumeric value
	TreatmentName	String value representing treatment name
	Cost	Numeric value
Medicine	TreatmentID (PK)	Unique alphanumeric value
	MedName	String value representing name of medication
	Dosage	Alphanumeric value representing dosage + weight/volume (e.g. 250mg or 10mL)
	Quantity	Numeric value for total amount prescribed
Room	RoomID (PK)	Unique 3-digit numeric value. First digit will represent the floor and the subsequent two digits are the room identifier
	RoomNum	Numerical value (00-99)
Department	DeptID (PK)	Unique string identifying department
	FloorNum	Numeric value between 1-100

Table 2: Domain Constraints for Relationships

Relationship	Attribute	Domain
Administer	EmployeeID (PK)	Unique numeric value in the range of 000001-999999
	TreatmentID (PK)	Unique alphanumeric value
	Date	Date-time value (MM/DD/YYYY Time)
WorksIn	EmployeeID (PK)	Unique numeric value in the range of 000001-999999
	DeptID	Unique string identifying department
Located	RoomID (PK)	Unique 3-digit numeric value. First digit will represent the floor and the subsequent two digits are the room identifier
	DeptID	Unique string identifying department
Assigned	PatientID (PK)	Unique numeric value in the range of 000001-999999
	RoomID	Unique 3-digit numeric value. First digit will represent the floor and the subsequent two digits are the room identifier
	Date	Date-time value (MM/DD/YYYY Time)
Billed	PatientID (PK)	Unique numeric value in the range of 000001-999999
	TreatmentID (PK)	Unique alphanumeric value
	BillDate	Date-time value (MM/DD/YYYY Time)
PCP	PatientID (PK)	Unique numeric value in the range of 000001-999999
	EmployeeID	Unique numeric value in the range of 000001-999999
Monitor	RoomID (PK)	Unique 3-digit numeric value. First digit will represent the floor and the subsequent two digits are the room identifier
	EmployeeID (PK)	Unique numeric value in the range of 000001-999999

Table 3: Relationship Functionality

Relationship	Functionality	Justification
Administer	Many-to-Many	Many employees can perform a single treatment and many treatments can be done by one employee. A treatment cannot exist without an employee to perform it.
WorksIn	Many-to-One	Many employees can work in a single department
Located	Many-to-One	There are many rooms located in a single department
Assigned	One-to-One	A room is assigned only one patient
Billed	Many-to-Many	Many patients can be billed for many treatments
PCP	Many-to-One	A primary care physician will have many different patients, but patients will only have one PCP
Monitor	Many-to-Many	Many nurses can monitor many different rooms

Additional constraints to consider:

- The specialty attribute of an employee should designate the specific role. For example, if a nurse does not have a specialty then the attribute should hold

the title of the employee (e.g. nurse). This will prevent null values from being needed.

- An employee or patient will only have a single phone number. The database will not account for a person having more than one phone.
- The address of a patient or employee will represent their primary address. The database will not include other properties an individual might own.
- Every treatment performed will contain a unique ID. The purpose behind this is so that unique records can be maintained when similar treatments are performed for various patients.

Logical Design:

This section converts the ER diagram into a relational schema. Any functional dependencies for a given relation are defined at this stage of the design, along with the highest normal form achieved.

Employee (EmployeeID, FirstName, LastName, Address, Phone, DOB, Specialty, Salary)
FDs: EmployeeID → FirstName, LastName, Address, Phone, DOB, Specialty, Salary
Normal Form: BCNF

Doctor (EmployeeID, MedSchool, Residency)
FDs: EmployeeID → MedSchool, Residency
Normal Form: BCNF

Nurse (EmployeeID, NurseSchool)
FDs: EmployeeID → NurseSchool
Normal Form: BCNF

Patient (PatientID, FirstName, LastName, Address, Phone, DOB)
FDs: PatientID → FirstName, LastName, Address, Phone, DOB
Normal Form: BCNF

Treatment (TreatmentID, TreatmentName, Cost)
FDs: TreatmentID → TreatmentName, Cost
Normal Form: BCNF

Medicine (TreatmentID, MedName, Dosage, Quantity)
FDs: TreatmentID → MedName, Dosage, Quantity
Normal Form: BCNF

Room (RoomID, RoomNum)
FDs: RoomID → RoomNum
Normal Form: BCNF

Department (DeptID, FloorNum)
FDs: DeptID → FloorNum
Normal Form: BCNF

Administer (EmployeeID, TreatmentID, Date)
FDs: EmployeeID, TreatmentID → Date
Normal Form: BCNF

Works_In (EmployeeID, DeptID)
FDs: EmployeeID → DeptID
Normal Form: BCNF

Located (RoomID, DeptID)
FDs: RoomID → DeptID
Normal Form: BCNF

Assigned (PatientID, RoomID, Date)
FDs: PatientID → RoomID, Date
Normal Form: BCNF

Billed (PatientID, TreatmentID, BillDate)
FDs: PatientID, TreatmentID → BillDate
Normal Form: BCNF

PCP (PatientID, EmployeeID)
FDs: PatientID → EmployeeID
Normal Form: BCNF

Monitor (RoomID, EmployeeID)
FDs: None
Normal Form: BCNF

The decomposition of the ER diagram into a relational schema required two important steps. First, entities were converted into a relation (the name of the relation is the entity name) containing all attributes listed in the diagram. Second, the relationships between entities were converted into a relation based on the cardinality constraints. The following rules were used:

- One-to-one relationships: Primary key of the relation could be the primary key from either entity. This can be chosen arbitrarily
- One-to-many relationships: Primary key of the relation is the primary key of the many side entity.
- Many-to-many relationships: Primary key of the relation is the primary key from both participating entities.

After the relations were constructed, functional dependencies were defined. Combined with the functional dependencies and the primary keys, it was possible to determine the normal form for each relation. Given the design, no decomposition was required to obtain either BCNF or 4NF.

Database:

With the design of the database finalized, the primary focus of this section was to demonstrate how to implement everything using MySQL. The following scripts below showcase how the tables were created based on the logical design. These SQL statements also implement the integrity constraints that were defined during the design process (e.g. primary keys, foreign keys, and non-null attributes).

Table 4: SQL Scripts to Create Database Tables

CREATE TABLE employee (
  employeeID int(6) unsigned zerofill,
  firstName varchar(20) NOT NULL,
  lastName varchar(25) NOT NULL,
  address varchar(100) NOT NULL,
  phone varchar(15) NOT NULL,
  DOB date NOT NULL,
  specialty varchar(50) NOT NULL,
  salary decimal(10,0) NOT NULL,
  PRIMARY KEY (employeeID)
);
	CREATE TABLE patient (
	  patientID int(6) unsigned zerofill,
	  firstName varchar(20) NOT NULL,
	  lastName varchar(25) NOT NULL,
	  address varchar(100) NOT NULL,
	  phone varchar(15) NOT NULL,
	  DOB date NOT NULL,
	  PRIMARY KEY (patientID)
	);

CREATE TABLE doctor (
  employeeID int(6) unsigned zerofill,
  medSchool varchar(40) NOT NULL,
  residency varchar(40) NOT NULL,
  PRIMARY KEY (employeeID),
  FOREIGN KEY (employeeID)
  REFERENCES employee (employeeID)
);
	CREATE TABLE nurse (
	  employeeID int(6) unsigned zerofill,
	  nurseSchool varchar(40) NOT NULL,
	  PRIMARY KEY (employeeID),
	  FOREIGN KEY (employeeID)
	  REFERENCES employee (employeeID)
	);

CREATE TABLE treatment (
  treatmentID char(6),
  treatmentName varchar(40) NOT NULL,
  cost decimal(10,2) NOT NULL,
  PRIMARY KEY (treatmentID)
);
	CREATE TABLE medicine (
	  treatmentID char(6),
	  medName varchar(40) NOT NULL,
	  dosage varchar(15) NOT NULL,
	  quantity decimal(3,0) NOT NULL,
	  PRIMARY KEY (treatmentID),
	  FOREIGN KEY (treatmentID)
	  REFERENCES treatment (treatmentID)
	);

CREATE TABLE administer (
  employeeID int(6) unsigned zerofill,
  treatmentID char(6) NOT NULL,
  adminDate datetime NOT NULL,
  PRIMARY KEY (employeeID, treatmentID),
  FOREIGN KEY (employeeID)
  REFERENCES employee (employeeID),
	CREATE TABLE billed (
	  patientID int(6) unsigned zerofill,
	  treatmentID char(6) NOT NULL,
	  billDate datetime NOT NULL,
	  PRIMARY KEY (patientID, treatmentID),
	  FOREIGN KEY (patientID)
	  REFERENCES patient (patientID),

```
  FOREIGN KEY (treatmentID)
  REFERENCES treatment (treatmentID)
);
```

```
  FOREIGN KEY (treatmentID)
  REFERENCES treatment (treatmentID)
);
```

```
CREATE TABLE department (
  deptID varchar(20) NOT NULL,
  floorNum decimal(3,0) NOT NULL,
  PRIMARY KEY (deptID)
);
```

```
CREATE TABLE worksIn (
  employeeID int(6) unsigned zerofill,
  deptID varchar(20) NOT NULL,
  PRIMARY KEY (employeeID),
  FOREIGN KEY (employeeID)
  REFERENCES employee (employeeID),
  FOREIGN KEY (deptID)
  REFERENCES department (deptID)
);
```

```
CREATE TABLE room (
  roomID decimal(3,0) NOT NULL,
  roomNum decimal(2,0) NOT NULL,
  PRIMARY KEY (roomID)
);
```

```
CREATE TABLE monitor (
  roomID decimal(3,0),
  employeeID int(6) unsigned zerofill,
  PRIMARY KEY (roomID, employeeID),
  FOREIGN KEY (roomID)
  REFERENCES room (roomID),
  FOREIGN KEY (employeeID)
  REFERENCES employee (employeeID)
);
```

```
CREATE TABLE located (
  roomID decimal(3,0),
  deptID varchar(20) NOT NULL,
  PRIMARY KEY (roomID),
  FOREIGN KEY (roomID)
  REFERENCES room (roomID),
  FOREIGN KEY (deptID)
  REFERENCES department (deptID)
);
```

```
CREATE TABLE assigned (
  patientID int(6) unsigned zerofill,
  roomID decimal(3,0) NOT NULL,
  assignDate datetime NOT NULL,
  PRIMARY KEY (patientID),
  FOREIGN KEY (patientID)
  REFERENCES patient (patientID),
  FOREIGN KEY (roomID)
  REFERENCES room (roomID)
);
```

```
CREATE TABLE PCP (
  patientID int(6) unsigned zerofill,
  employeeID int(6) unsigned zerofill,
  PRIMARY KEY (patientID),
  FOREIGN KEY (patientID) REFERENCES patient (patientID),
  FOREIGN KEY (employeeID) REFERENCES employee (employeeID)
);
```

The tables below highlight some of the more advanced features that can be utilized. Because each relation was designed to conform to BCNF, extracting certain information from the database could require several joins between tables. For example, if an admin wanted to obtain a list of the patients an employee has performed treatments on, they would need to join five tables (employee, treatment, administer, patient, and billed). To simplify this process, a view was constructed to display this information.

In addition, a few sample views were created to limit the amount of data displayed, such as personal employee or patient information. These views were incorporated so admins could grant privileges to other users for a limited view, instead of allowing them access to the original tables. Since there are no restrictions on the number of views that can be created, an admin for the database would have the ability to create even more views than the ones shown in *Table 5*.

Table 5: Sample SQL Views Created for Data Security and Simplification

`CREATE VIEW employeeData AS`
`SELECT e.employeeID, e.lastName,`
`  e.specialty, w.deptID`
`FROM employee e, worksIn w`
`WHERE e.employeeID = w.employeeID;`
	`CREATE VIEW patientBill AS`
	`SELECT p.patientID, b.treatmentID,`
	`  t.treatmentName, t.cost,`
	`  b.billDate`
	`FROM patient p, billed b, treatment t`
	`WHERE p.patientID = b.patientID`
	`AND b.treatmentID = t.treatmentID;`

`CREATE VIEW patientData AS`
`SELECT patientID, firstName, lastName`
`FROM patient;`
	`CREATE VIEW employeePatientTreatment AS`
	`SELECT e.employeeID, p.patientID,`
	`  a.treatmentID, t.treatmentName,`
	`  a.adminDate`
	`FROM employee e, administer a,`
	`  treatment t, patient p, billed b`
	`WHERE e.employeeID = a.employeeID`
	`AND a.treatmentID = t.treatmentID`
	`AND t.treatmentID = b.treatmentID`
	`AND b.patientID = p.patientID;`

Apart from views, other more advanced features that were created involved the use of MySQL triggers. These triggers could be used to maintain a log of changes to a table or keep a consistent formatting structure on new information being added. *Table 6* showcases some of these situations, with some triggers created for recording logs and others to format inserted information.

The triggers created in *Table 6* represent just some of the advanced features that can be utilized. There are many more possibilities to help streamline the database, which will be dependent on the needs of the client hospital. The goal with this section was to provide a few sample triggers to demonstrate how they can be applied to a hospital database. Note, since MySQL does not support triggers that operate on multiple types of modifications, the following triggers only format text on inserted data. Additional triggers would need to be created to also format updated information.

Table 6: Sample SQL Triggers for Maintaining Logs and Formatting Input

CREATE TABLE salary_log (
  employeeID int(6) UNSIGNED ZEROFILL,
  oldSalary decimal(10, 0),
  newSalary decimal(10, 0),
  changeDate datetime,
  FOREIGN KEY (employeeID)
  REFERENCES employee(employeeID)
);
	DELIMITER //
	CREATE TRIGGER log_salary
	AFTER UPDATE on employee
	FOR EACH ROW
	BEGIN
	  IF OLD.salary != NEW.salary THEN
	  INSERT INTO salary_log
	  VALUES (OLD.employeeID,
	  OLD.salary,
	  NEW.salary,
	  sysdate());
	  END IF;
	END//
	DELIMITER ;

CREATE TABLE employee_log (
  employeeID int(6) UNSIGNED ZEROFILL,
  oldDept varchar(20) NOT NULL,
  newDept varchar(20) NOT NULL,
  changeDate datetime NOT NULL,
  FOREIGN KEY (employeeID)
  REFERENCES employee(employeeID)
);
	DELIMITER //
	CREATE TRIGGER log_employee
	AFTER UPDATE on worksIn
	FOR EACH ROW
	BEGIN
	  IF OLD.deptID != NEW.deptID THEN
	  INSERT INTO employee_log
	  VALUES (OLD.employeeID,
	  OLD.deptID,
	  NEW.deptID,
	  sysdate());
	  END IF;
	END//
	DELIMITER ;

```
CREATE TABLE pcp_log (
  patientID int(6) UNSIGNED ZEROFILL,
  oldPCP int(6) UNSIGNED ZEROFILL,
  newPCP int (6) UNSIGNED ZEROFILL,
  changeDate datetime NOT NULL,
  FOREIGN KEY (patientID)
  REFERENCES patient(patientID),
  FOREIGN KEY (oldPCP)
  REFERENCES employee(employeeID),
  FOREIGN KEY (newPCP)
  REFERENCES employee(employeeID)
);
```

```
DELIMITER //
CREATE TRIGGER log_pcp
AFTER UPDATE on PCP
FOR EACH ROW
BEGIN
  IF OLD.employeeID != NEW.employeeID THEN
    INSERT INTO pcp_log
    VALUES (OLD.patientID,
            OLD.employeeID,
            NEW.employeeID,
            sysdate());
  END IF;
END//
DELIMITER ;
```

```
DELIMITER //
CREATE TRIGGER employee_insert
BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
  SET
  NEW.firstName = upper(NEW.firstName),
  NEW.lastName = upper(NEW.lastName),
  NEW.address = upper(NEW.address),
  NEW.specialty = upper(NEW.specialty);
END//
```

```
DELIMITER //
CREATE TRIGGER treatment_insert
BEFORE INSERT ON treatment
FOR EACH ROW
BEGIN
  SET
  NEW.treatmentID = upper(NEW.treatmentID),
  NEW.treatmentName = upper(NEW.treatmentName);
END//
```

```
DELIMITER //
CREATE TRIGGER administer_insert
BEFORE INSERT ON administer
FOR EACH ROW
BEGIN
  SET
  NEW.treatmentID = upper(NEW.treatmentID);
END//
```

```
DELIMITER //
CREATE TRIGGER department_insert
BEFORE INSERT ON department
FOR EACH ROW
BEGIN
  SET NEW.deptID = upper(NEW.deptID);
END//
```

```
DELIMITER //
CREATE TRIGGER patient_insert
BEFORE INSERT ON patient
FOR EACH ROW
BEGIN
  SET
  NEW.firstName = upper(NEW.firstName),
  NEW.lastName = upper(NEW.lastName),
  NEW.address = upper(NEW.address);
END//
```

```
DELIMITER //
CREATE TRIGGER located_insert
BEFORE INSERT ON located
FOR EACH ROW
BEGIN
  SET NEW.deptID = upper(NEW.deptID);
END//
```

```
DELIMITER //
CREATE TRIGGER doctor_insert
```

```
DELIMITER //
CREATE TRIGGER billed_insert
```

```
BEFORE INSERT ON doctor
FOR EACH ROW
BEGIN
  SET
  NEW.medSchool = upper(NEW.medSchool),
  NEW.residency = upper(NEW.residency);
END//
```

```
BEFORE INSERT ON billed
FOR EACH ROW
BEGIN
  SET
  NEW.treatmentID = upper(NEW.treatmentID);
END//
```

```
DELIMITER //
CREATE TRIGGER medicine_insert
BEFORE INSERT ON medicine
FOR EACH ROW
BEGIN
  SET
  NEW.treatmentID = upper(NEW.treatmentID),
  NEW.medName = upper(NEW.medName);
END//
```

```
DELIMITER //
CREATE TRIGGER nurse_insert
BEFORE INSERT ON nurse
FOR EACH ROW
BEGIN
  SET NEW.nurseSchool = upper(NEW.nurseSchool);
END//
```

```
DELIMITER //
CREATE TRIGGER worksin_insert
BEFORE INSERT ON worksIn
FOR EACH ROW
BEGIN
  SET NEW.deptID = upper(NEW.deptID);
END//
```

Web Interface:

To fully tie everything discussed in the project proposal together, a basic web interface was constructed. The purpose of this web design was to provide a simple interface for MySQL database administration. Implementation was done by using phpMyAdmin with Cloud SQL on App Engine. For more details on how this was accomplished, see the source code at the following GitHub page: https://github.com/mjsmccarthy/CMSC508-Project.

To view the sample hospital database, which was designed using the SQL statements described previously, use the following link: https://cmsc508-project-258320.appspot.com/

When at the login page, there are two possible user accounts. The first user, which functions as an admin, has full control over the database. This means they can update and make changes to the database whenever necessary. They also can modify permissions for other users on the database. The second user has a much more limited role and can only view data, they are unable to make any type of modifications. To use either account, see the login information below:

Table 7: Account Login Information for Web Interface

Role	**Username**	**Password**
Root/Admin	root	cmsc508_project
Client	client	cmsc508_proejct

Once logged into the system, the sample database can be viewed by clicking on the database labeled as “project” on the left-hand side menu. At this point, SQL queries can be written for the various tables in the database and the corresponding results will be shown.

Conclusion:

The goal of this proposal was to outline the construction of a database for a basic hospital management system. Given the needs of the hospital, the database will provide an efficient way to store and retrieve information related to their patients, employees, and treatments. This will not only make retrieving information easier, but it will also provide the hospital with a seamless way to monitor its overall functioning.

In addition to providing the conceptual and logical designs for the database, a fully functional web interface was created. This was accomplished by using phpMyAdmin with Cloud SQL on App Engine. The purpose behind the web interface was to help demo the database, but more importantly, provide a seamless way to perform administration tasks for users using the system.