HOSPITAL DATABASE MANAGEMENT SYSTEM

CS5200 – DATABASE MANAGEMENT SYSTEMS

Authors: Sravani Namburu - namburu.sr@northeastern.edu

Rahul Chettri - Chettri.ra@northeastern.edu

Kyle Domingos - domingos.k@northeastern.edu

INTRODUCTION

In today's world, hospitals generate a vast amount of data every day. Efficient management and organization of this data is crucial to the success of the hospital. In this project, we have developed a Hospital Database Management System using MySQL and Python to provide an easy-to-use and efficient solution for managing hospital data. We have implemented a command line interface that offers a menu-driven output, making it simpler for users to interact with the system. Our primary focus is on designing a system that is scalable and can be extended in the future to accommodate new features and functionality. The project follows an iterative development process, starting with requirements gathering, database design, implementation, and testing.

This report provides a detailed description of the database's architecture, the design choices made, and an evaluation of the system's performance. The report also includes a user manual and instructions for installation of the required software.

README

Hardware requirements:

- Intel Core i5 or higher or OS X 10.9 or higher
- RAM: 4GB or higher
- Storage: 100GB or higher

Software requirements:

- Operating System: Windows 10 or Mac OS X or higher
- Database Management System: MySQL 8.0 or other
- Programming Language: Python 3.8 or higher
- Python libraries: pymysql to connect with the database.

Installation Instructions:

To install and set up the Hospital Database Management System, please follow these steps:

1. Download the software required from the official website.
2. Install MySQL 8.0 or higher on your computer.
3. Create a new database and import the database schema.
4. Install Python 3.8 or higher on your computer.
5. Install the required Python packages.

6. Run the application using the command “python hospital.py”.

Commands Available:

The available commands once running the application are:

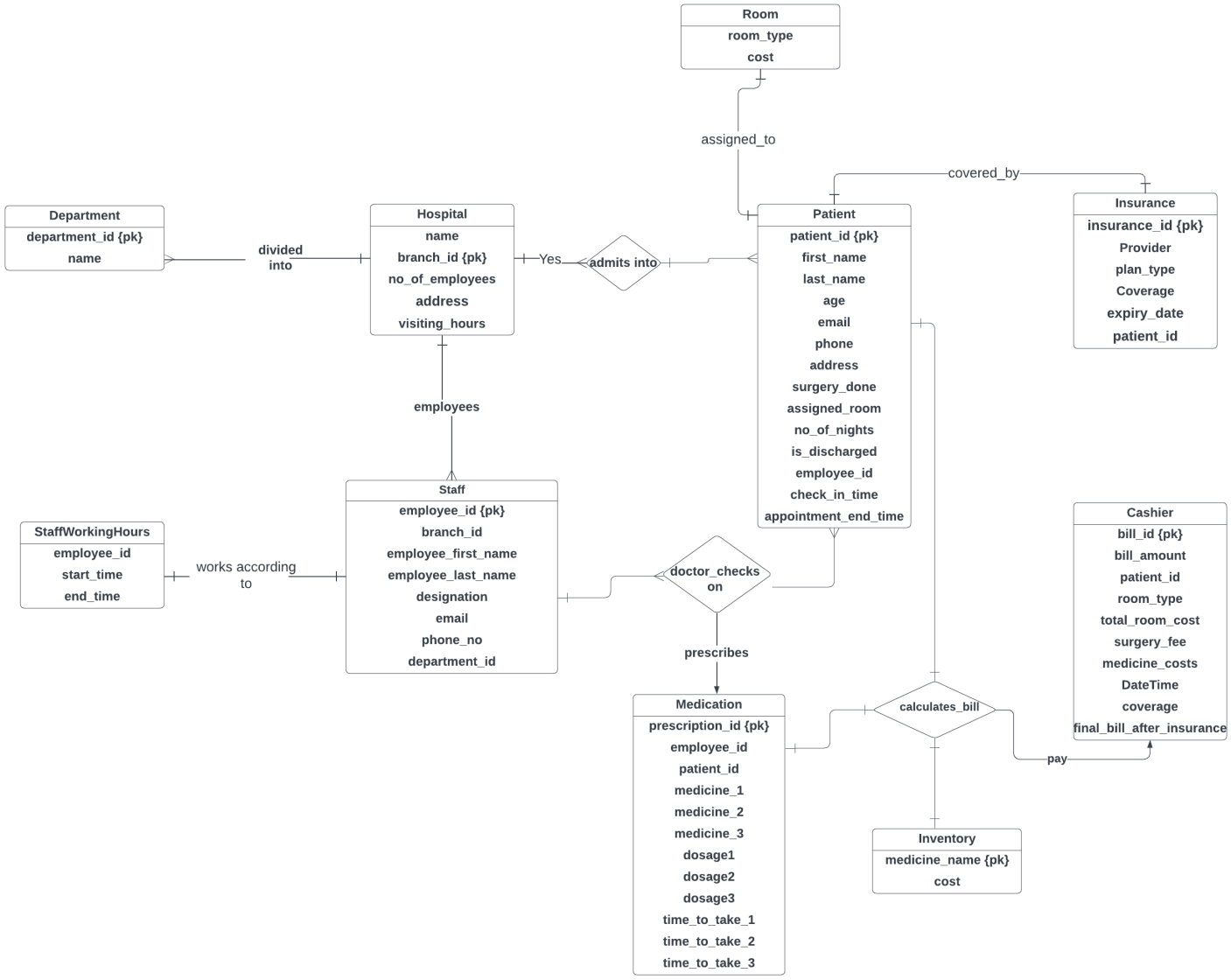
- new_department
- new_employee
- generate_bill
- pay_bill
- admit_patient
- generate_prescription
- insert_update_medicine
- insert_update_insurance
- insert_hospital_data
- insert_room_data
- quit

Database Schema:

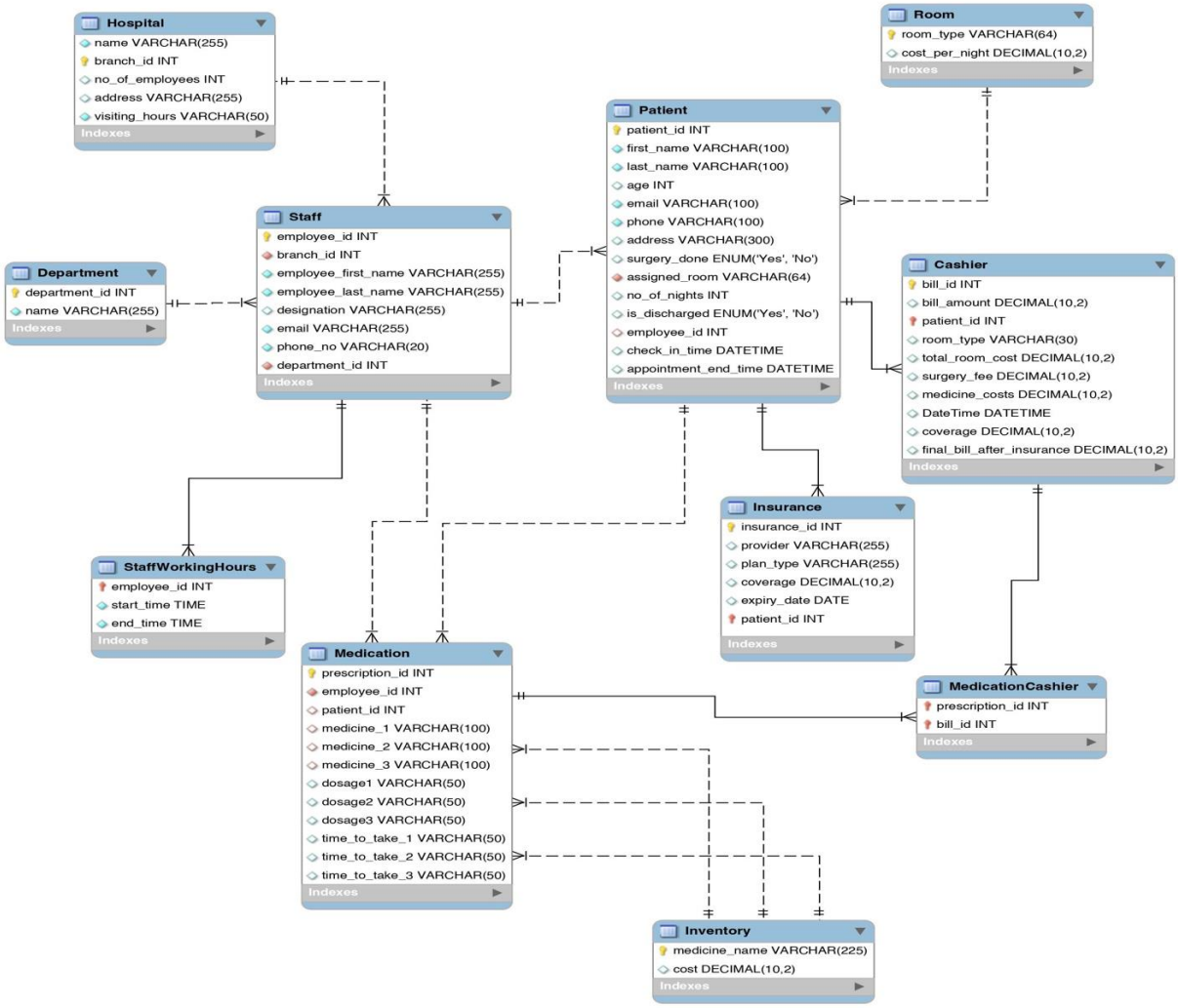
The Hospital Database Management System uses the following database schema:

- Hospital table (name, branch_id, no_of_employees, address, visiting_hours)
- Department table (department_id, name)
- Staff table (employee_id, branch_id, employee_first_name, employee_last_name, designation, email, phone_no, department_id)
- StaffWorkingHours table (employee_id, start_time, end_time)
- Room (room_type, cost_per_night)
- Patient (patient_id, first_name, last_name, age, email, phone, address, surgery_done, assigned_room, no_of_nights, is_discharged, employee_id, check_in_time, appointment_end_time)
- Insurance (insurance_id, provider, plan_type, coverage, expiry_date, patient_id)
- Inventory (medicine_name, cost)
- Medication (prescription_id, employee_id, patient_id, medicin_1, medicine_2, medicine_3, dosage1, dosage2, dosage3, time_to_take_1, time_to_take_2, time_to_take_3)
- Cashier (bill_id, bill_amount, patient_id, room_type, total_room_cost, surgery_fee, medicine_costs, DateTime, coverage, final_bill_after_insurance)
- MedicationCashier (prescription_id, bill_id)

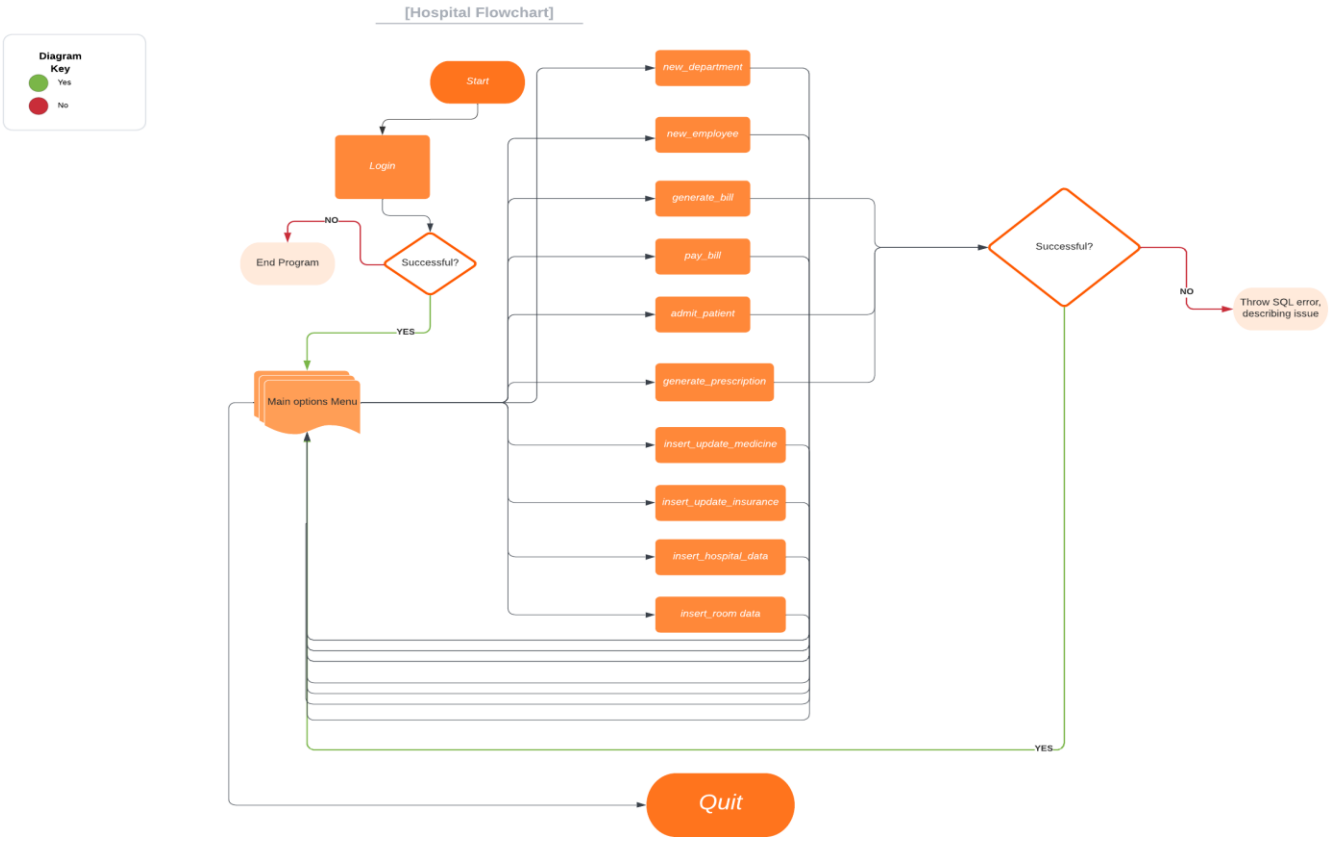
Conceptual design as a UML:



Logical design obtained by reverse engineering the schema



Userflow and features:



The Hospital Database Management System comes with the following procedures and triggers implemented as the key features of our hospital database management system:

admit_patient

- **Purpose:**
  - The AdmitPatient stored procedure is designed to admit a patient into the hospital system by inserting their information into the Patient table and assigning an available doctor.
- **Input Parameters:**
  - p_first_name (VARCHAR (50)): Patient's first name
  - p_last_name (VARCHAR(50)): Patient's last name
  - p_age (INT): Patient's age
  - p_email (VARCHAR(100)): Patient's email address
  - p_phone (VARCHAR(20)): Patient's phone number
  - p_address (VARCHAR(255)): Patient's address
  - p_surgery_done (ENUM('Yes', 'No')): Surgery status of the patient
  - p_assigned_room (VARCHAR(64)): Room assigned to the patient
  - p_no_of_nights (INT): Number of nights the patient will stay
  - p_is_discharged (ENUM('Yes', 'No')): Discharge status of the patient

- **Process:**
  - Finds an available doctor within their working hours.
  - If an available doctor is found, inserts the patient record into the Patient table and assigns the available doctor to the patient.
  - If no doctor is available, throws an error with the message "No doctors are available at the moment. Please try again later."
- **Error Handling:**
  - If no doctor is available, the procedure throws a custom SQL error with the message "No doctors are available at the moment. Please try again later."
- **Usage Example:**
  - CALL admit_patient ('John', 'Doe', 35, 'john.doe@example.com', '+1-555-123-4567', '123 Main St, Anytown, USA', 'No', 'General ward', 3, 'No');
  - This example call to the stored procedure admits a patient with the provided information and assigns an available doctor.

generate_prescription

- **Purpose:**
  - The **generate_prescription** stored procedure is designed to create a new prescription for a patient, parsing and storing the given medicine names, dosages, and times to take.
- **Input Parameters:**
  - p_patient_id (INT): Patient ID
  - p_medicines (TEXT): Comma-separated list of medicine names
  - p_dosages (TEXT): Comma-separated list of dosages
  - p_times_to_take (TEXT): Comma-separated list of times to take each medicine
- **Process:**
  - Parses the medicine names, dosages, and times to take from the comma-separated input parameters.
  - Retrieves the employee_id (doctor) assigned to the current medication holder.
  - Validates the employee_id, and raises an error if not found.
  - Inserts a new record into the Medication table with the parsed values and the retrieved employee_id and given patient_id.
- **Usage Example:**
  - CALL generate_prescription(2011, 'Lisinopril,Furosemide,Metformin', '500 mg,25 mg,20 mg', 'once daily,twice daily,once daily');
  - This example call to the stored procedure creates a new prescription for the specified patient with the provided information on the medicines.

generate_bill

- **Declare local variables:**
  Store temporary values like room type, total room cost, surgery fee, medicine costs, bill amount, insurance coverage, and final bill after insurance

- **Retrieve patient's billing details:**
    - **Create a CTE named `CostDetails` to:**
    - Join the Patient and Room tables on the assigned room type.
    - Calculate the total room cost (cost per night multiplied by the number of nights)
    - Determine the surgery fee (1000 if surgery was performed, 0 otherwise)
    - **Create another CTE named `MedicineCosts` to:**
    - Join the Medication and Inventory tables on medicine names.
    - Calculate the total medicine cost by summing up the costs of all medicines prescribed to the patient.
    - Combine the results of both CTEs and join with the Patient and Insurance tables.
    - Retrieve room type, total room cost, surgery fee, medicine costs, and insurance coverage for the given patient ID.
- **Error handling:**
    - If the patient does not exist in the Patient table, raise a custom error with SQLSTATE '45000' and the message "Patient not found in Patient table."
    - If the patient does not exist in the Medication table, raise a custom error with SQLSTATE '45000' and the message "Patient not found in Medication table."
- **Calculate final bill after insurance:**
    - If the patient has no insurance coverage:
    - Set the final bill equal to the total bill amount.
    - Set insurance coverage to 0.
    - If the patient has insurance coverage:
    - Deduct the insurance coverage from the total bill amount to calculate the final bill after insurance.
    - If the final bill after insurance is negative, set it to 0.
    - Update the remaining insurance coverage in the Insurance table by subtracting the total bill amount.
- **Insert the calculated bill into the Cashier table:**
    - Add a new record with the patient ID, room type, bill amount, current date and time, surgery fee, total room cost, medicine costs, insurance coverage, and final bill after insurance.

The **`generate_bill`** procedure provides a comprehensive and automated solution for calculating a patient's bill, factoring in various cost components, applying insurance coverage, and inserting the final bill into the Cashier table. This helps streamline the billing process and ensures accuracy and consistency in the calculation of patient bills.

PayBill

- **Purpose:**
    - The **PayBill** stored procedure is designed to clear the patient’s account indicating that the bill was paid by updating the final_bill_after_insurance in the Cashier table.

- **Input Parameters:**
  - p_patient_id (INT): Patient_id of the patient, who wants to pay the bill.
- **Usage Example:**
  - CALL PayBill(2003);
  - This example call to the stored procedure clears the account of the patient associated with the patient_id of 2003 to indicate that the patient has paid the bill.

InsertOrUpdateInsurance

- **Purpose:**
  - The **InsertOrUpdateInsurance** stored procedure is designed to either insert a new insurance record or update an existing insurance record in the Insurance table, based on the given parameters.
- **Input Parameters:**
  - p_insurance_id (INT): Insurance ID
  - p_provider (VARCHAR(255)): Insurance provider name
  - p_plan_type (VARCHAR(255)): Insurance plan type
  - p_coverage (DECIMAL(10, 2)): Insurance coverage amount
  - p_expiry_date (DATE): Insurance expiry date
  - p_patient_id (INT): Patient ID
- **Process:**
  - Checks if the insurance_id and patient_id combination already exists in the Insurance table.
  - If the combination exists, updates the existing insurance record with the given values, retaining the highest coverage value.
  - If the combination doesn't exist, checks if the patient_id already exists with a different insurance_id.
  - If the patient_id exists with a different insurance_id, raises an error.
  - if the patient_id doesn't exist, inserts a new record with the given insurance and patient information.
- **Usage Example:**
  - CALL **InsertOrUpdateInsurance**(60007, 'Provider Name', 'Plan Type', 8000.00, '2024-04-18', 2000);
  - This example call to the stored procedure either inserts or updates an Insurance record with the provided values.
  - SELECT * FROM insurance;
  - This query displays all records in the Insurance table after the stored procedure call.

AddOrUpdateMedicineInInventory

- **Purpose:**
  - The **AddOrUpdateMedicineInInventory** stored procedure is designed to either insert a new medicine record or update an existing medicine record in the Inventory table, based on the given parameters.
- **Input Parameters:**

    - p_medicine_name (VARCHAR(255)): Medicine name
    - p_cost (DECIMAL(10, 2)): Cost of the medicine
- **Process:**
    - a. Checks if the medicine already exists in the Inventory table.
    - b. If the medicine exists, updates the cost of the existing medicine record.
    - c. If the medicine doesn't exist, inserts a new record with the given medicine name and cost.
- **Usage Example:**
    - CALL AddOrUpdateMedicineInInventory('Advil', 20.99);
    - This example call to the stored procedure either inserts or updates an Inventory record with the provided values.
    - SELECT * FROM Inventory;
    - This query displays all records in the Inventory table after the stored procedure call.

InsertHospitalData

- **Purpose:**
    - The **InsertHospitalData** stored procedure is designed to insert a new hospital record into the Hospital table, using the given parameters.
- **Input Parameters:**
    - p_name (VARCHAR(255)): Hospital name
    - p_branch_id (INT): Hospital branch ID
    - p_no_of_employees (INT): Number of employees in the hospital
    - p_address (VARCHAR(255)): Hospital address
    - p_visiting_hours (VARCHAR(255)): Visiting hours for the hospital
- **Process**: a. Inserts a new record into the Hospital table with the given values.
- **Usage Example:**
    - CALL InsertHospitalData('SRK Hospitals', 16001, 15, 'Bennington Street, Boston, MA', '11:00:00 - 18:00:00');
    - This example call to the stored procedure inserts a new Hospital record with the provided values.
    - CALL InsertHospitalData('SRK Hospitals', 16002, 15, 'Alverton St, Virginia', '12:00:00 - 18:00:00');
    - This example call inserts another Hospital record with the provided values.

InsertRoomData

- **Purpose:**
    - The **InsertRoomData** store procedure is designed to insert any new type of room into the Room table, using the given parameters.
- **Input Parameters:**
    - P_room_type (VARCHAR (255)): Type of the room
    - P_cost_per_night (DECIMAL (10,2)): Cost of the room per night.

new_employee

- **Purpose:**
  - The **new_employee** stored procedure is designed to insert a new employee record into the Staff table, using the given parameters.
- **Input Parameters:**
  - branch_id_p (INT): Branch id of the hospital into which the new employee record is being inserted into.
  - first_name_p (VARCHAR (50)): First name of the employee
  - last_name_p (VARCHAR (50)): Last name of the employee
  - designation_p (VARCHAR (10)): Designation of the employee, i.e., whether the employee is a doctor or a nurse.
  - email_p (VARCAHR (100)): Email address of the employee
  - phone_p (VARCHAR(15)): Contact number of the employee
  - department_id (INT): Indicates which department the employee belongs to.
- **Usage Example:**
  - CALL new_employee (16003, 'Priya', 'Sam', 'Nurse', 'priya.sa@srk.org','(563)634-4562', 106);
  - This example call to the stored procedure inserts a new Employee record with the name 'Priya Sam' under the hospital represented by the branch_id 16003.

Increase_employee_count trigger

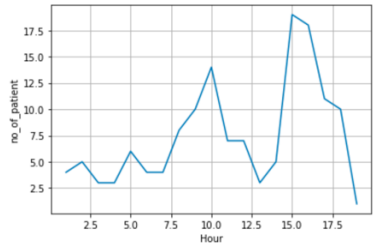
- **Purpose:**
  - The purpose of the increase_employee_count trigger is to update the number of employees associated with each hospital in the Hospital table whenever a new employee record is inserted. This trigger ensures that the employee count for each hospital stays up to date.

Update_discharge_status trigger

- **Purpose:**
  - The update_discharge_status trigger is designed to update the discharge status of a patient in the 'Patient' table when the patient pays their bill. This trigger ensures that the patient's discharge status is accurately reflected in the database after they have settled their account.

As part of the bonus part, we came up with a visualization based on the random data we created for the patient as below:

```
from matplotlib.ticker import StrMethodFormatter
plt.plot(busytime['Hour'],busytime['patient_id'])
plt.xlabel('Hour')
plt.ylabel('no_of_patient')
plt.grid()
plt.show()
```



Lessons learned:

- Creating a hospital database management system project using MySQL and Python made us have a good exposure to Procedures, triggers, and other miscellaneous concepts.
- Implementing our own hospital database gave us better appreciation of the complexities and issues facing the current health care system.
- When starting this project, we found it quickly turned into something increasingly complex as we attempted to add functionality.
- Gained the hospital domain insights which enabled improvement in data quality, interoperability, and data governance.

Limitations:

The Hospital Database Management System has the following limitations:

- Limited to walk-in scheduling of appointments for patients.
- Few features for administration and Human Resources.
- Limited features for nurses and other employees.

Future Scope:

In the future, we plan to add the following features to the Hospital Database Management System:

- Create a user interface web application by incorporating front end technologies, python web framework like Django or java web framework like Spring Boot using MySQL for backend.
- Online appointment scheduling for patients.
- Automatic billing and payment processing.
- Tracking of inventory for pharmacies
- Common side effects of drugs, and their interactions with other prescribed medication
- Scheduling for non-walk-in patients, and future appointments.
- Fleshed out features for more staff members, including nurses and administrative members.