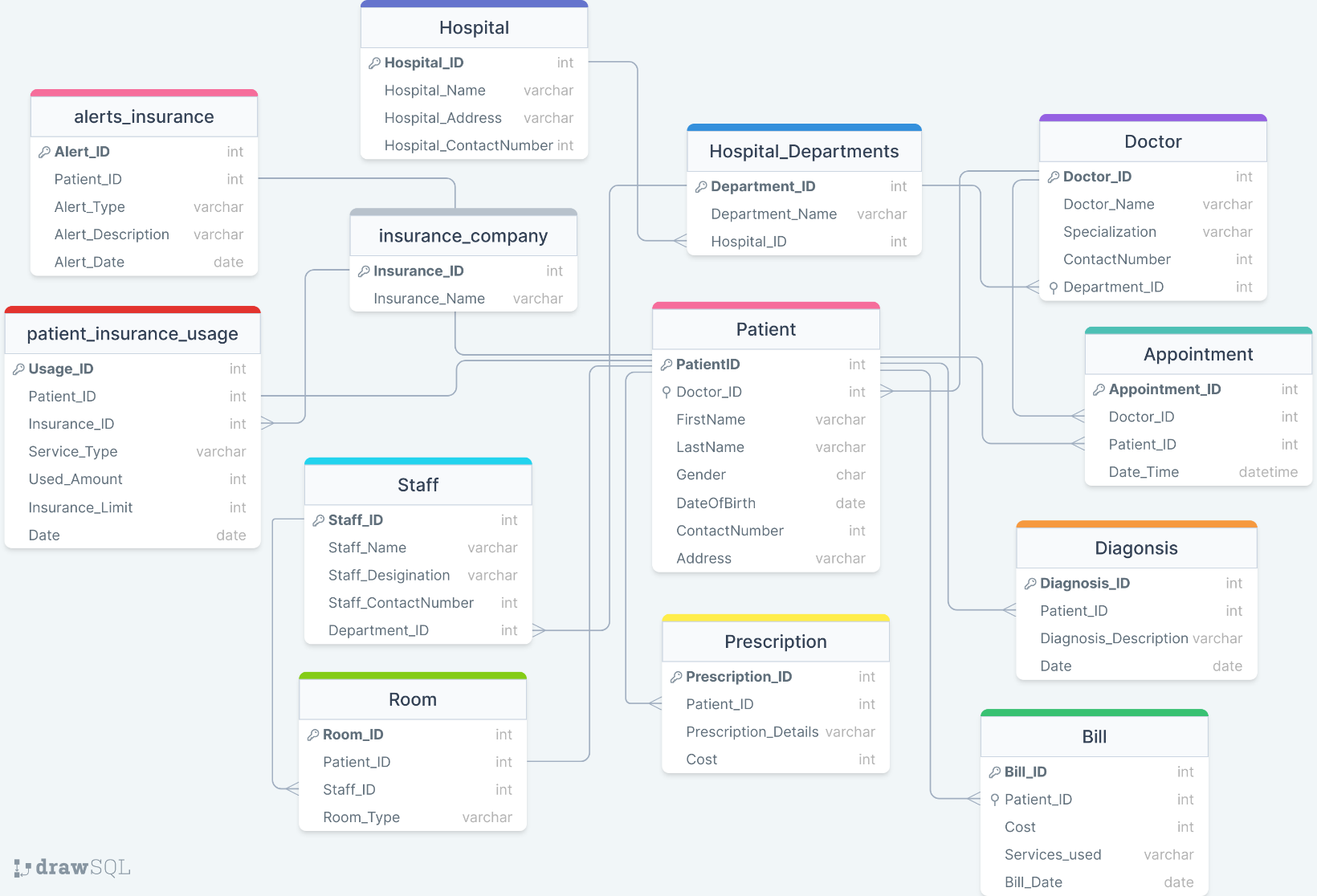
**Milestone Project 1**

**Question 1**

**Create a DB Schema for Hospital Management System.**

****

**Fig: Schema Representation of Hospital Management System**

The figure represents Hospital Management System Database Schema.

The Schema is normalized till BCNF normalization:

* Each table has a primary key that uniquely identifies each record.
* Foreign keys are used to establish relationships between tables.
* Each attributes in table contains atomic value
* All non-key attributes in each table are fully functional dependent on the primary key.
* There are no transitive dependencies within the tables.

**- >Write necessary queries to register new user roles and personas**

**Registering and Assigning Department to a Doctor :**

INSERT INTO `doctor` (`Doctor\_ID`, `Doctor\_Name`, `Specialization`, `ContactNumber`, `Department\_ID`) VALUES ('11014', 'Dr. Rohit Mehra', 'Cardiology', '997879885', '1');

**Registering a new Patient :**

INSERT INTO `patient` (`PatientID`, `Doctor\_ID`, `FirstName`, `LastName`, `Gender`, `Age`, `DateOfBirth`, `ContactNumber`, `Address`) VALUES ('103', '11014', 'Arjun', 'Kapoor', 'Male', '30', '1994-02-21', '997879874', 'Pune');

**Registering and Assigning Department to a Staff :**

INSERT INTO `staff` (`Staff\_ID`, `Staff\_Name`, `Staff\_Desigination`, `Staff\_ContactNumber`, `Department\_ID`) VALUES ('110103', 'Anuj Gupta', 'Clinical Nurse Specialist', '89498183', '4');

**->Write necessary queries to add to the list of diagnosis of the patient tagged by date.**

**Adding Diagnosis Data related to Patient:**

INSERT INTO `diagonsis` (`Diagnosis\_ID`, `Patient\_ID`, `Diagnosis\_Description`, `Date`) VALUES ('12', '103', 'Pneumonia', '2024-01-26');

**Updating Diagnosis Details of Patient:**

UPDATE diagonsis

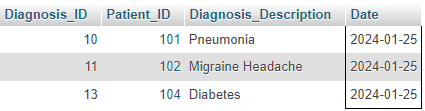
SET diagonsis.Diagnosis\_Description = 'Diabetes'

WHERE Patient\_ID = 104 AND diagonsis.Date = '2024-01-25';

**List of Diagnosis of the Patients tagged by Date:**

SELECT \* FROM diagonsis

WHERE diagonsis.Date="2024-01-25";



Output:

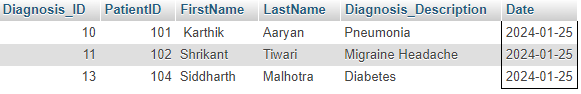
**List of Diagnosis of the Patients Data tagged by Date:**

SELECT d.Diagnosis\_ID,p.PatientID,p.FirstName,p.LastName,d.Diagnosis\_Description,d.Date

FROM patient p

JOIN diagonsis d on p.PatientID=d.Patient\_ID

WHERE d.Date='2024-01-25';



Output:

**->Write necessary queries to fetch required details of a particular patient.**

**Retrieving Patients Data using Patient ID:**

SELECT \* FROM patient WHERE patient.PatientID=102;

Output : 

**Retrieving Patients Data using Patient Name:**

SELECT \* FROM patient WHERE patient.FirstName='Arjun';

Output: 

**Retrieving Patients Diagnosis Data using Patient ID:**

SELECT p.PatientID,p.FirstName,p.LastName,d.Diagnosis\_Description,d.Date FROM patient p

JOIN diagonsis d on p.PatientID=d.Patient\_ID

WHERE p.PatientID=103;

Output:

**Retrieving Patient** **Prescription Data using Patient ID:**

SELECT p.PatientID,p.FirstName,p.LastName,p.Age,pr.Prescription\_Details,pr.Cost FROM patient

p JOIN prescription pr on p.PatientID=pr.Patient\_ID

WHERE p.PatientID=103;

Output:

**Retrieving Patient’s and Doctor’s Data:**

SELECT p.PatientID,p.FirstName,p.LastName,p.Gender,d.Doctor\_Name,d.Specialization

FROM patient p

JOIN doctor d ON p.Doctor\_ID = d.Doctor\_ID

WHERE p.Doctor\_ID='11014';

 Output:

**Retrieving Insurance Data of Patient:**

SELECT p.PatientID,p.FirstName,p.LastName,pi.Service\_Type,pi.Usage\_Amount,pi.Date AS 'Valid\_till' FROM patient p

JOIN patient\_insurance\_usage pi ON p.PatientID=pi.Patient\_ID

WHERE p.PatientID=102;

Output:

**->Write necessary queries to prepare bill for the patient at the end of checkout.**

**Calculating Prescription Cost of the Patient:**

SELECT p.Patient\_ID , GROUP\_CONCAT(p.Prescription\_Details,' ') as 'Prescription',COUNT(p.Cost) as 'Quantity', SUM(p.Cost) as 'Total Cost' FROM prescription p WHERE p.Patient\_ID=103;

 Output:

**Calculating Appointment Charges of Patient:**

SELECT a.Patient\_ID,p.FirstName,p.LastName,COUNT(a.Appointment\_ID) as "No of Appointments",SUM(a.Appointment\_Charges) as 'Total Charges'

FROM appointment a

JOIN patient p ON a.Patient\_ID = p.PatientID

WHERE p.PatientID = '103'

LIMIT 25;

Output:

**Calculating Room Charges of Patient:**

SELECT p.PatientID,p.FirstName,p.LastName,GROUP\_CONCAT(r.Room\_Type," ") as 'Room Types' , SUM(r.Room\_Cost) as 'Total Room Cost'

FROM room r

JOIN patient p ON r.Patient\_ID=p.PatientID

WHERE p.PatientID='102';



Output:

**Calculating Overall Bill of a Patient:**

SELECT patient.PatientID,patient.FirstName,patient.LastName,patient.Gender,patient.Age,

Total\_Appointment\_Charges + Total\_Prescription\_Charges + Total\_Ward\_Charges AS Total\_Bill

FROM

(

SELECT SUM(Appointment\_Charges) AS Total\_Appointment\_Charges FROM appointment

WHERE appointment.Patient\_ID='103') AS Total\_Appointment\_Charges,

(

SELECT SUM(prescription.Cost) AS Total\_Prescription\_Charges FROM prescription

WHERE prescription.Patient\_ID='103') AS Total\_Prescription\_Charges,

(

SELECT SUM(room.Room\_Cost) AS Total\_Ward\_Charges FROM room

WHERE room.Patient\_ID='103') AS Total\_Ward\_Charges ,

patient WHERE patient.PatientID='103';

Output:

**-> Write necessary queries to fetch and show data from various related tables (Joins)**

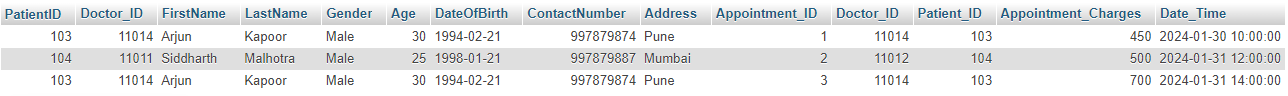
**Retriving data using INNER JOIN between Patient and Appointment Table:**

SELECT p.\*, a.\*

FROM patient p

INNER JOIN appointment a ON p.PatientID = a.Patient\_ID

LIMIT 0, 25;

Output:

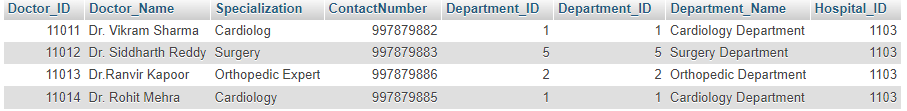
**Retriving data using LEFT JOIN between Doctor and Department Table:**

SELECT d.\*,hd.\*

FROM doctor d

LEFT JOIN hospital\_departments hd ON d.Department\_ID=hd.Department\_ID

LIMIT 0, 25;

Output: 

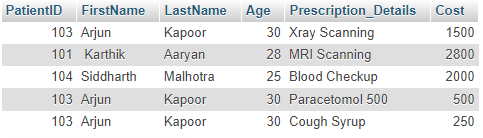
**Retriving data using JOIN between Patient and Prescription Table:**

SELECT p.PatientID,p.FirstName,p.LastName,p.Age,pr.Prescription\_Details,pr.Cost

FROM patient p

JOIN prescription pr on p.PatientID=pr.Patient\_ID

LIMIT 0, 25;



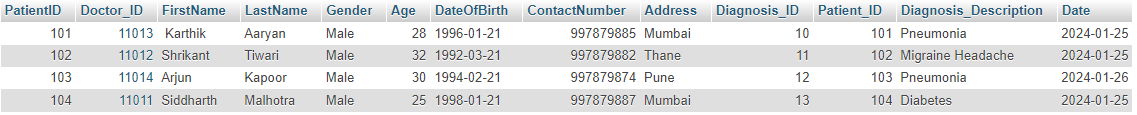
Output:

**Retriving data using RIGHT JOIN between Patient and Diagnosis Table:**

SELECT \*

FROM patient p

RIGHT JOIN diagonsis d on p.PatientID=d.Patient\_ID;

Output: 

**Retriving data using FULL JOIN between Patient and Prescription Table:**

SELECT \* FROM patient

RIGHT JOIN prescription ON patient.PatientID = prescription.Patient\_ID

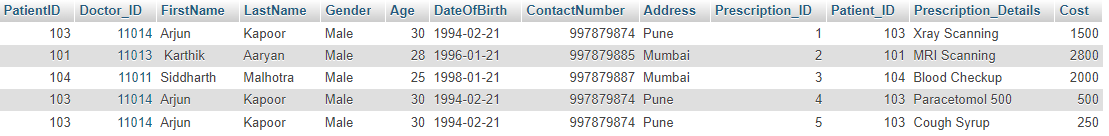
UNION

SELECT \* FROM patient

LEFT JOIN prescription ON patient.PatientID = prescription.Patient\_ID

WHERE patient.PatientID IS NULL

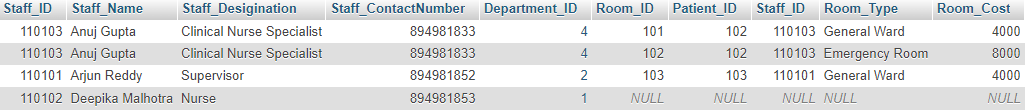
LIMIT 0, 25;

Output:

**Retriving data using LEFT JOIN between Staff and Room Table:**

SELECT \* FROM staff

LEFT JOIN room ON staff.Staff\_ID=room.Staff\_ID;

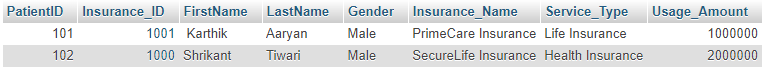
Output:

**Retriving data using Multiple JOIN between Patient , Insurance and Patient\_Insurance Table:**

SELECT p.PatientID,i.Insurance\_ID,p.FirstName,p.LastName,p.Gender,i.Insurance\_Name,pi.Service\_Type,pi.Usage\_Amount FROM patient\_insurance\_usage pi

JOIN patient p ON p.PatientID = pi.Patient\_ID

JOIN insurance\_company i ON i.Insurance\_ID = pi.Insurance\_ID;

Output: 

**Retriving data using Multiple JOIN between Patient , Doctor ,Prescription and Room Table:**

SELECT p.PatientID,d.Doctor\_ID,p.FirstName,p.LastName,d.Doctor\_Name,pr.Prescription\_Details,r.Room\_Type

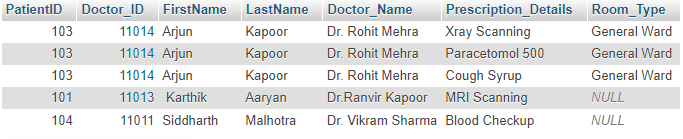
FROM patient p

JOIN doctor d ON d.Doctor\_ID = p.Doctor\_ID

INNER JOIN prescription pr ON pr.Patient\_ID = p.PatientID

LEFT JOIN room r ON r.Patient\_ID = p.PatientID

LIMIT 0, 25;

Output: 

**-> Optimize repeated read operations using views/materialized views**

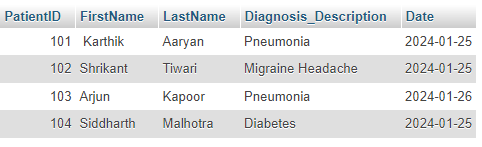
**Created a view named “PatientDiagnosis” which will help in retrieving Patients and Diagnosis Data:**

CREATE VIEW PatientDiagnosis AS

SELECT p.PatientID, p.FirstName, p.LastName, d.Diagnosis\_Description, d.Date

FROM patient p

JOIN diagonsis d ON p.PatientID = d.Patient\_ID;

Output:

**Created a view named “patientprescriptionview” which will help in retrieving Patients and Prescription Data:**

CREATE VIEW PatientPrescriptionView AS

SELECT

p.PatientID,

p.Doctor\_ID,

p.FirstName,

p.LastName,

p.Gender,

p.Age,

p.DateOfBirth,

p.ContactNumber,

p.Address,

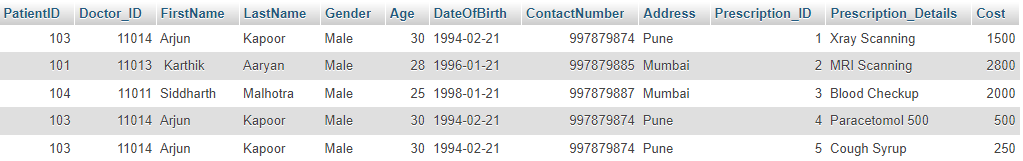
pr.Prescription\_ID,

pr.Prescription\_Details,

pr.Cost

FROM Patient p

INNER JOIN Prescription pr ON p.PatientID = pr.Patient\_ID;

Output:

**Created a View named “PatientDiagnosisSummary” . This view stores summarized data of patients.**

CREATE VIEW PatientSummaryView AS

SELECT

p.PatientID,

p.FirstName,

p.LastName,

p.Gender,

p.Age,

p.Address,

doc.Doctor\_Name,

pr.Prescription\_Details,

d.Diagnosis\_Description,

r.Room\_Type,

pi.Service\_Type,

pi.Usage\_Amount

FROM

patient p

LEFT JOIN

prescription pr ON p.PatientID = pr.Patient\_ID

LEFT JOIN

diagonsis d ON p.PatientID = d.Patient\_ID

LEFT JOIN

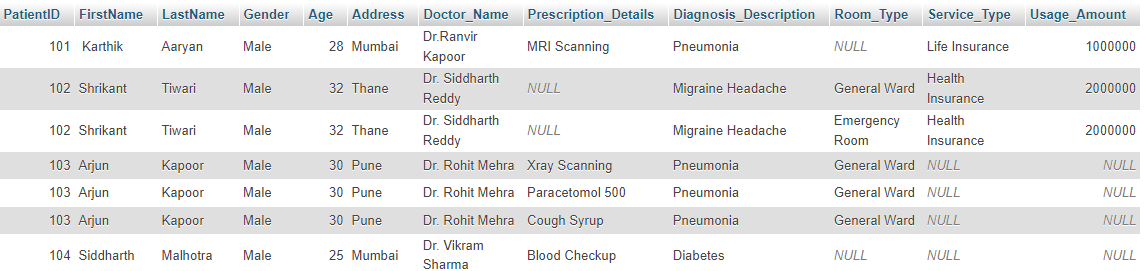
doctor doc on p.Doctor\_ID=doc.Doctor\_ID

LEFT JOIN

room r ON r.Patient\_ID=p.PatientID

LEFT JOIN

patient\_insurance\_usage pi ON pi.Patient\_ID=p.PatientID

**Output:**

**->Optimize read operations using indexing wherever required. (Create index on at least 1 table)**

**Created a index in Patient Table on Patient\_ID column:**

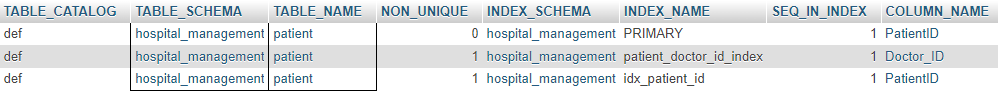
CREATE INDEX idx\_patient\_id ON patient(PatientID);

Verifying the created index :

SELECT \* FROM INFORMATION\_SCHEMA.STATISTICS

WHERE TABLE\_SCHEMA = 'hospital\_management' AND TABLE\_NAME = 'patient';

**Output:**



**Created a index in Prescription Table on Prescription\_ID column:**

CREATE INDEX idx\_ prescription\_id ON prescription(Prescription\_ID);

Verifying the created the index:

SELECT \* FROM INFORMATION\_SCHEMA.STATISTICS

WHERE TABLE\_SCHEMA = 'hospital\_management' AND TABLE\_NAME = 'prescription';

**Output**:



**->Try optimizing bill generation using stored procedures.**

**Created a stored procedures which will calculate Patients Total Bill . Patient\_ID is used as an input parameter.**

Query:

DELIMITER //

CREATE PROCEDURE GenerateBillWithDetails (IN p\_PatientID INT)

BEGIN

DECLARE totalCost DECIMAL(10, 2);

SELECT

Patient.PatientID, Patient.FirstName, Patient.LastName, Patient.Gender,

Patient.Age, Patient.DateOfBirth, Patient.ContactNumber, Patient.Address

INTO

@PatientID, @FirstName, @LastName, @Gender,@Age, @DateOfBirth,@ContactNumber,

@Address

FROM Patient

WHERE PatientID = p\_PatientID;

SELECT IFNULL(SUM(Cost), 0) INTO totalCost

FROM Prescription

WHERE Patient\_ID = p\_PatientID;

SELECT IFNULL(Room\_Cost, 0) INTO totalCost

FROM Room

WHERE Patient\_ID = p\_PatientID;

SELECT

@PatientID AS PatientID,

@FirstName AS FirstName,

@LastName AS LastName,

@Gender AS Gender,

@Age AS Age,

@DateOfBirth AS DateOfBirth,

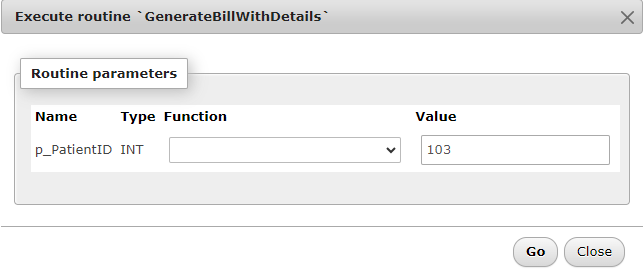
@ContactNumber AS ContactNumber,

@Address AS Address,

totalCost AS TotalCost;

END //

DELIMITER ;

**Input:** 

**Output:** 

**- >Add necessary triggers to indicate when patients medical insurance limit has expired.**

**Created a trigger to check for insurance limit expiration after each insert in patient\_insurance\_usage table:**

**Query:**

DELIMITER //

CREATE TRIGGER trg\_insurance\_limit\_expiration

AFTER INSERT ON patient\_insurance\_usage

FOR EACH ROW

BEGIN

DECLARE remaining\_limit DECIMAL(10, 2);

SET remaining\_limit = NEW.Insurance\_Limit - NEW.Used\_Amount;

IF remaining\_limit <= 0 THEN

-- Insert alert into alerts\_insurance table

INSERT INTO alerts\_insurance (Patient\_ID, Alert\_Type, Alert\_Description, Alert\_Date)

VALUES (NEW.Patient\_ID, 'Insurance Limit Exceeded', CONCAT('Insurance limit exceeded for Service Type: ', NEW.Service\_Type), NOW());

END IF;

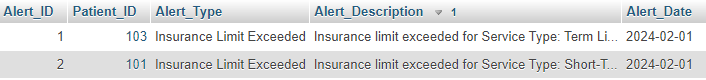
END //

DELIMITER ;

**Verifying Trigger:**

INSERT INTO `patient\_insurance\_usage` (`Usage\_ID`, `Patient\_ID`, `Insurance\_ID`, `Service\_Type`, `Used\_Amount`, `Insurance\_Limit`, `Date`) VALUES ('4', '101', '1002', 'Short-Term Health Insurance', '100000', '100000', '2024-01-26');

**Output:**

****

**Created a trigger to check for insurance limit expiration after each update in patient\_insurance\_usage table**

**Query:**

DELIMITER //

CREATE TRIGGER trg\_insurance\_limit\_update

AFTER UPDATE ON patient\_insurance\_usage

FOR EACH ROW

BEGIN

DECLARE remaining\_limit DECIMAL(10, 2);

SET remaining\_limit = NEW.Insurance\_Limit - NEW.Used\_Amount;

IF remaining\_limit <= 0 THEN

INSERT INTO alerts\_insurance (Patient\_ID, Alert\_Type, Alert\_Description, Alert\_Date)

VALUES (NEW.Patient\_ID, 'Insurance Limit Exceeded', CONCAT('Insurance limit exceeded for Service Type: ', NEW.Service\_Type), NOW());

END IF;

END //

DELIMITER ;

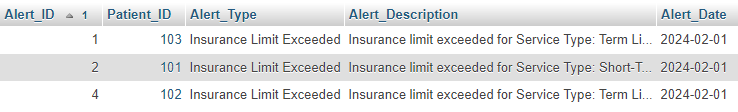
**Verifying Trigger:**

UPDATE patient\_insurance\_usage

SET Used\_Amount = 20000

WHERE Patient\_ID = '102';

**Output:**

****

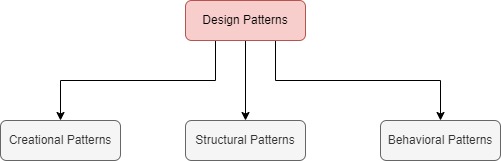
**Question 2**

**Write a report on your understanding of Rendering and Design Patterns. Mention and elaborate where a particular Rendering pattern is applicable and is well suited for which use case.**

**Answer:**

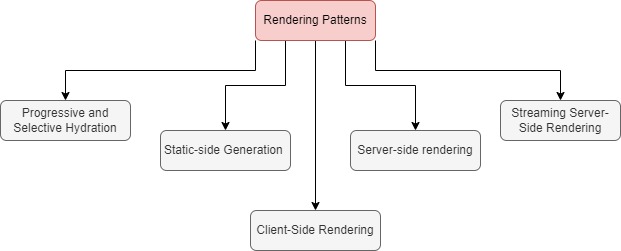
Design patterns are reusable solutions to software design problems. They help developers create flexible, maintainable, and scalable code by using best practices and design principles.

Rendering Pattern is specific to user interface development and focus on efficient rendering of graphical elements on a screen.

Design Patterns is mainly divided into three types:

1. Creational Patterns: Creational patterns works on object creation mechanisms, abstracting the process to make a system independent of how its objects are created, and represented.Creational Patterns contains further more patterns like Singleton , Factory Method , Abstract Factory and many more .
2. Structural Patterns: Structural patterns deal with object composition or the structure of classes. They help ensure that if one part of a system changes, the entire system doesn't need to be rebuilt. Structural Patterns contains the further more patterns like Adapter , Decorator , Facade and many more.
3. Behavioral Patterns: Behavioral patterns focus on communication between objects, defining how objects interact and splits the responsibility. Behavioral Patterns contains further more patterns such as Observer , Strategy , Command and many more.

Design Patterns helps developers in saving time and effort . The have loosely coupled designs that are easier to maintain and scale.

Rendering Patterns contains various types like :

1. Client-Side Rendering(CSR) : In CSR, the web browser loads minimum HTML page containing JavaScript code. This JavaScript code then makes requests to the server to fetch data and renders the content dynamically in the browser. This approach offers fast initial page load.
2. Static Site Generation(SSG) : SSG assembles the entire website into static HTML files during the compile time. Once generated, these HTML files are served to clients without any further processing . It Offers fast load times as there is no need for server-side processing for each request.
3. Server-Side Rendering(SSR): SSR involves generating the HTML for each page on the server for each request. When a user requests a page, the server fetches the data, processes it, and generates the HTML and then fully rendered HTML is then sent to the client's browser, which can display it immediately.
4. Streaming Server-Side Rendering: Streaming Server-Side Rendering combines the benefits of SSR and CSR. In SSSR , the server starts sending the HTML response to the client as soon as it starts rendering, even before the entire page is generated. This allows the client to start displaying content incrementally as it's being received. It requires server and client-side support to handle streaming responses effectively

Overall , design patterns serve as blueprints for designing software systems, guiding developers to create flexible, and maintainable code .On the other hand, Rendering patterns refer to different approaches for generating and displaying web content to users.

As per requirements the required Rendering Pattern is used for the system.

1. Client-Side Rendering (CSR): CSR is used when you need a highly dynamic user experience with frequent data updates and interactions. CSR is used for building Single Page Application(SPA) .The Frameworks like Angular, React, and Vue.js provides tools to build SPAs. CSR is used to develop highly interactive websites.
2. Static Site Generation (SSG): SSG is used when your website's content doesn't change frequently and can be build at initial time. SSG provides excellent performance, security, and simplicity. Additionally, since the content is static, it can be easily cached and distributed via Content Delivery Networks (CDNs)which helps in improving performance and accessibility. SSG is used to develop Content-based websites, blogs, documentation sites.
3. Server-Side Rendering (SSR): SSR is useful when you need to generate HTML dynamically on the server for each request. It's particularly useful for content-heavy websites that need to be indexed by search engines for SEO purposes. SSR ensures that search engine crawlers index the fully rendered data which helps in improving the SEO performance. SSR is used for developing E-Commerce websites, news portals, and multi-page applications with SEO requirements.
4. Streaming Server-Side Rendering (SSSR): SSSR is useful for applications where rendering large pages contents can cause delays in the initial page load. SSSR sends HTML files to the client incrementally as it's being generated on the server. SSSR is used in Large-scale applications with complex UIs, real-time applications, data visualization applications.
5. Progressive Rendering: This technique involves rendering an image gradually, starting with a low-quality approximation and refining it over time to achieve higher quality. It involve focusing on rendering the most critical parts of the scene first and the remaining parts. It is used for developing interactive application, 3D rendering Software
6. Selective Rendering: Selective rendering involves rendering only the parts of the scene that are initially relevant to the viewer. Selective Rendering involves dynamically adjusting the level of detail or quality for different parts based on various factors. It is used in VR and AR application for smooth rendering of objects/models and for maintaining the framerate.