Arogya Mitra

"Bringing together diverse healthcare professionals and users for a unified experience."

Introduction to the Project

1.1Project Overview

- ### Purpose and Objective

  Design and implement a relational database for a health portal that efficiently organizes and manages vast healthcare-related data. The database should support multiple user roles, offer flexible querying for insights, and ensure data security and integrity, especially around sensitive medical information.

1.2 Background and Motivation

Why SQL?

SQL (Structured Query Language) is essential for managing and querying relational databases, enabling efficient data retrieval, manipulation, and storage. It's widely adopted due to its standardization, ease of use, and support for complex data operations.

Applications of SQL in Real-World Projects:

- **Healthcare Portals:** Manage patient records, appointments, and prescriptions securely.
- **E-commerce**: Track inventory, orders, and customer data across multiple tables.
- **Finance:** Handle transaction records, generate reports, and manage account balances.
- **Social Media Platforms:** Organize user profiles, posts, and engagement metrics.
- **Education Systems:** Store student data, grades, and course enrollment details.

Expected Outcomes:

1. **Efficient Data Management:** Streamlined storage and retrieval of healthcare data, including patient records, appointments, and billing, enhancing overall data organization.

2. **Improved User Experience:** Quick access to information for all user types (patients, physicians, etc.), allowing seamless interaction and faster service delivery.
3. **Enhanced Data Security:** Secure handling of sensitive health data with role-based access control, ensuring compliance with privacy regulations.
4. **Scalable Insights and Analytics:** Ability to generate insights and reports for health trends, appointment patterns, and user engagement, supporting informed decision-making.
5. **Operational Efficiency:** Reduced redundancy and automated workflows, like appointment reminders and billing updates, to streamline health services.

1. Database Design

2.1 Requirements Gathering

- ### User Types Analysis:

1 **Patients:** Need to access personal health records, schedule appointments, view prescriptions, and communicate with healthcare providers.

2 **Physicians/Practitioners**: Require the ability to manage patient records, schedule consultations, update treatment plans, and access lab results.

3 **Researchers:** Seek access to anonymized health data for analysis, insights into health trends, and collaboration tools for studies.

4 **Service Providers**: Need to manage service listings, view bookings, and access customer feedback.

5 **Health Content Creators**: Require tools to upload, manage, and publish health-related content.

- ### Functional Requirements:

1. **User Authentication:** Secure login and registration processes for different user roles.
2. **Profile Management:** Ability for users to update personal and professional information.
3. **Appointment Scheduling:** Functionality for patients to book, modify, and cancel appointments.

4. **Record Management:** Physicians can create, update, and view patient medical records.
5. **Subscription Management:** Options for users to choose and manage their subscription plans.
6. **Notification System:** Alerts for appointment reminders, prescription updates, and system messages.
7. **Data Analytics:** Generate reports for health statistics and user engagement metrics.

- ## Non-Functional Requirements:

1. **Performance:** The system should handle multiple concurrent users without significant delays.
2. **Scalability:** Ability to expand the database and system functionalities as user demand grows.
3. **Security:** Implementation of data encryption, access control, and compliance with healthcare regulations (e.g., HIPAA).
4. **Usability:** User-friendly interface with intuitive navigation for all user roles.
5. **Reliability:** High availability and minimal downtime, ensuring users can access the portal consistently.
6. **Maintainability:** The system should be easy to update and maintain with minimal disruption to users.

2.2 Entity-Relationship (ER) Modeling

- ## Creating the ER Diagram:

The ER diagram visually represents the entities involved in the health portal, their attributes, and the relationships between them. Here's a brief description of key components

User

- **Attributes:** user_id (PK), name, email, password, user_type, subscription_type, created_at

Patient

- **Attributes:** patient_id (PK, FK to User), date_of_birth, medical_history

Physician

- **Attributes:** physician_id (PK, FK to User), specialty, license_number

Appointment

- **Attributes:** appointment_id (PK), patient_id (FK), physician_id (FK), date, time, status

MedicalRecord

- **Attributes:** record_id (PK), patient_id (FK), diagnosis, treatment, visit_date

Prescription

- **Attributes:** prescription_id (PK), patient_id (FK), medication_name, dosage, prescribed_by (FK to Physician)

Subscription

- **Attributes:** subscription_id (PK), user_id (FK), plan_type, start_date, end_date

Notification

- **Attributes:** notification_id (PK), user_id (FK), message, timestamp, status

Relationships

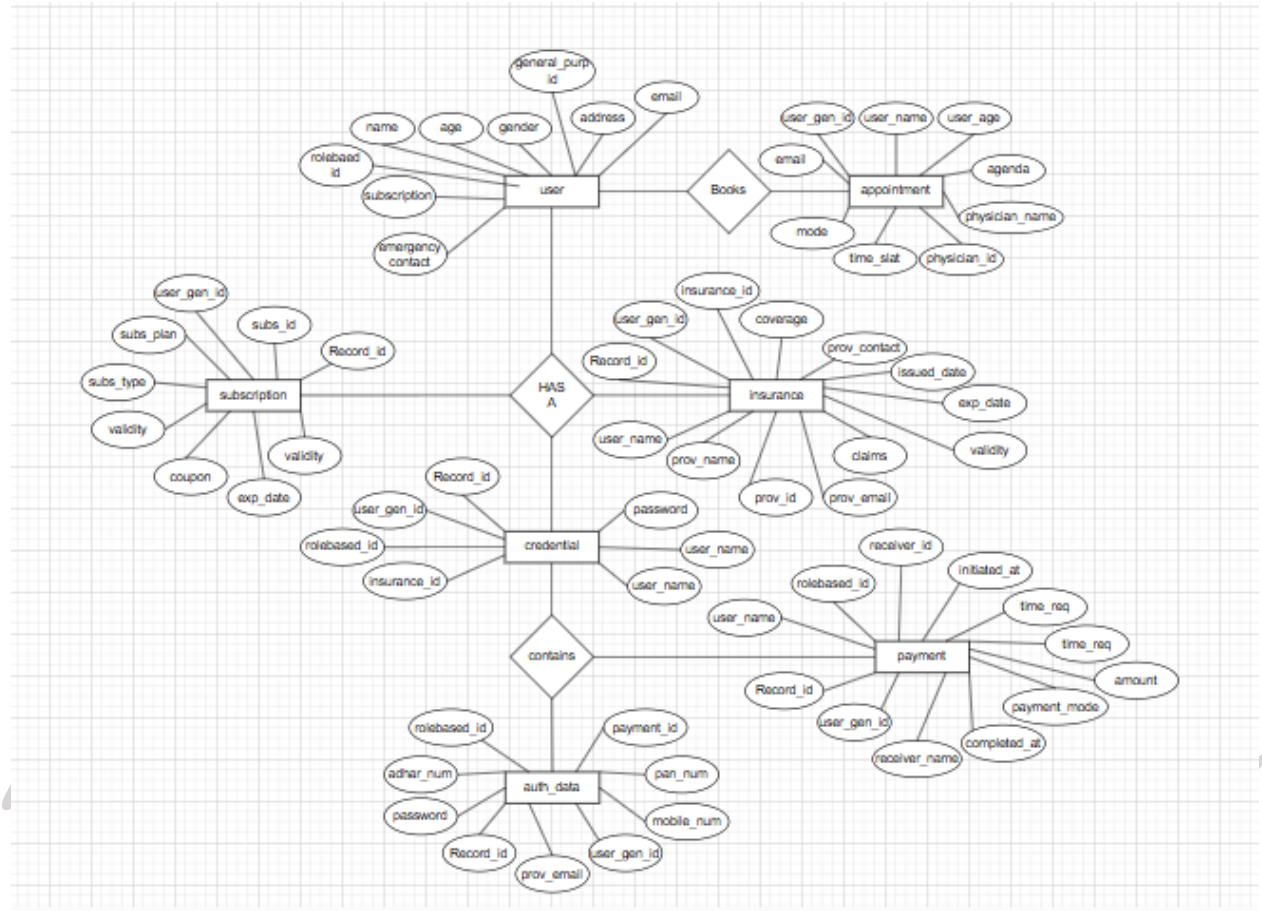
- **User to Patient:** One-to-One (A user can be a single patient)
- **User to Physician:** One-to-One (A user can be a single physician)
- **Patient to Appointment:** One-to-Many (A patient can have multiple appointments)
- **Physician to Appointment:** One-to-Many (A physician can handle multiple appointments)
- **Patient to MedicalRecord:** One-to-Many (A patient can have multiple medical records)
- **Patient to Prescription:** One-to-Many (A patient can have multiple prescriptions)
- **User to Subscription:** One-to-Many (A user can have multiple subscriptions over time)
- **User to Notification:** One-to-Many (A user can receive multiple notifications)

[User] --1:1--> [Patient] | [Patient] --1:N--> [MedicalRecord]

[User] --1:1--> [Physician] | [Patient] --1:N--> [Prescription]

[Patient] --1:N--> [Appointment] | [User] --1:N--> [Subscription]

[Physician] --1:N--> [Appointment] | [User] --1:N--> [Notification]


general_purp id
email
address
gender
age
name
rolebased id
subscription
emergency contact
user
Books
user_gen_id
user_name
user_age
email
agenda
physician_name
mode
time_slot
physician_id
appointment
user_gen_id
subs_id
subs_plan
Record_id
subs_type
validity
coupon
exp_date
validity
subscription
HAS A
insurance_id
user_gen_id
coverage
Record_id
prov_contact
issued_date
exp_date
validity
claims
user_name
prov_name
prov_id
prov_email
insurance
Record_id
user_gen_id
rolebased_id
insurance_id
credential
password
user_name
user_name
receiver_id
rolebased_id
user_name
initiated_at
time_req
time_req
amount
payment_mode
completed_at
receiver_name
Record_id
user_gen_id
payment
contains
rolebased_id
adhar_num
password
Record_id
prov_email
payment_id
pan_num
mobile_num
user_gen_id
auth_data

2.3 Normalization

Normalization is a process in database design that organizes data to minimize redundancy and improve data integrity. By applying normalization, we divide large tables into smaller, related tables and establish relationships between them. This helps ensure data consistency, reduces storage requirements, and enhances query performance.

Key Normal Forms (NF):

1. **1st Normal Form (1NF):**
   - (indivisible) values.
   - Eliminate duplicate columns within the same table.
   - Example: Separate phone numbers into individual rows instead of storing multiple numbers in one column.
   - Ensure each column contains atomic
2. **2nd Normal Form (2NF):**
   - Achieve 1NF and remove partial dependencies.
   - Each non-key attribute must depend on the entire primary key (applicable to tables with composite primary keys).
   - Example: If a table has student_id and course_id as a composite key, remove attributes that depend only on one part (e.g., student_name).
3. **3rd Normal Form (3NF):**
   - Achieve 2NF and remove transitive dependencies.
   - Non-key attributes should depend only on the primary key.
   - Example: In a table with employee_id, department_id, and department_name, remove department_name to a separate table linked by department_id.
4. **Boyce-Codd Normal Form (BCNF):**
   - A stricter version of 3NF, where every determinant is a candidate key.
   - Ensures even more robust data integrity, especially in complex relationships.

Benefits of Normalization:

- Reduces redundancy and improves storage efficiency.
- Maintains data consistency by eliminating anomalies (insertion, update, deletion).
- Optimizes queries by streamlining data structures.

Applying Normalization (1NF, 2NF, 3NF)

Let's apply the normalization process to a sample healthcare database:

Patient_id	Patient_name	Appointment_id	Physician_id	Physician_name	Diagnosis	Treatment
1	John Doe	101	201	Dr. Smith	Flu	Rest and Fluids
1	John Doe	102	202	Dr. Johnson	Cold	Rest and Meds
2	Jane Smith	103	201	Dr. Smith	Allergies	Antihistamines

Step 1: 1st Normal Form (1NF)

1NF Rule: Ensure each column contains atomic values, with no repeating groups or arrays.

Patient_id	Patient_name	Appointment_id	Physician_id	Physician_name	Diagnosis	Treatment
1	John Doe	101	201	Dr. Smith	Flu	Rest and Fluids
1	John Doe	102	202	Dr. Johnson	Cold	Rest and Meds
2	Jane Smith	103	201	Dr. Smith	Allergies	Antihistamines

All data in each column is atomic, and there are no repeating groups or arrays. This satisfies **1NF**.

Step 2: 2nd Normal Form (2NF)

2NF Rule: Achieve 1NF and remove partial dependencies, where non-key attributes depend only on the primary key (i.e., there should be no dependency on part of a composite key).

In this case, the table's primary key is a combination of **Patient_ID** and **Appointment_ID**. The **Physician_Name** depends on **Physician_ID**, not on the composite key **(Patient_ID, Appointment_ID)**. Therefore, we need to split the table.

Normalized Tables (2NF):

Patient table

Patient_id	Patient_name
1	John Doe
1	John Doe
2	Jane Smith

Appointment table

Appointme	Physician_id	Physician_name	Diagnosis	Treatment
101	201	Dr. Smith	Flu	Rest and Fluids
102	202	Dr. Johnson	Cold	Rest and Meds
103	201	Dr. Smith	Allergies	Antihistamines

Physician table

Physician_id	Physician_name
201	Dr. Smith
202	Dr. Johnson
201	Dr. Smith

Now, **Physician_Name** is moved to the **Physicians Table**, and **Appointments Table** only holds data that directly depends on the **Appointment_ID**. This satisfies 2NF.

Step 3: 3rd Normal Form (3NF)

3NF Rule: Achieve 2NF and remove transitive dependencies, where non-key attributes depend on other non-key attributes (i.e., no indirect dependency).

In the **Appointments Table**, Diagnosis and Treatment are related and depend on each other indirectly via the Patient_ID. To eliminate this, we separate the Diagnosis and Treatment information into a new table.

Patient table :

Patient_id	Patient_name
1	John Doe
2	Jane Smith

Appointment Table

Appointme	Patient_id	Physician_id
101	1	201
102	1	202
103	2	201

Physician table

Physician_i	Physician_name
201	Dr. Smith
202	Dr. Johnson

Diagnosis table

Diagnosis_i	Appointment_	Diagnosis	Treatment
1	101	Flu	Rest and Fluid
2	102	Cold	Rest and Med
3	103	Allergies	Antihistamine

3. **Database Implementation**

To create a table in SQL, you can use the CREATE TABLE statement. Here is the general syntax for creating a table:

```
CREATE TABLE table_name (
    column1 datatype [constraint],
    column2 datatype [constraint],
```

column3 datatype [constraint],

...

);

```
create table credential(
record_id int auto_increment primary key,
user_name varchar(50),
general_purp_id varchar(20),
rolebased_id varchar(20),
insurance_id varchar(15),
password varchar(15),
mobile_num varchar(10)
);
```

Tables present in database

Tables_in_arogyamitra
appointment
auth_data
credential
health_records
insurance
order_records
payment
roles
status
store_inventory
subscription
time_slat
user_rec

Constraints on table

1. PRIMARY KEY

- A PRIMARY KEY constraint uniquely identifies each record in a table.
- Each table can have only one PRIMARY KEY.
- A primary key column cannot contain NULL values.
- It automatically creates a unique index on the column(s).

```sql
CREATE TABLE order_records (
    record_id INT AUTO_INCREMENT PRIMARY KEY,
    order_id VARCHAR(25) NOT NULL,
    user_gen_id VARCHAR(20) NOT NULL,
    product_id VARCHAR(20) NOT NULL,
    product_name VARCHAR(100) NOT NULL,
    product_cat VARCHAR(50),
    product_quantity INT NOT NULL,
    offer VARCHAR(50),
    batch_code VARCHAR(20),
    shipping_id VARCHAR(20),
    shipping_status VARCHAR(20) CHECK (shipping_status IN ('pending', 'shipped', 'delivered', 'not available')),
    shipping_address VARCHAR(255),
    payment_id VARCHAR(25),
    amount DECIMAL(10, 2) NOT NULL
);
```

2. FOREIGN KEY

- A FOREIGN KEY constraint ensures that the value in a column (or combination of columns) matches a value in another table's primary key or unique key.
- It is used to enforce referential integrity between two tables.
- It helps maintain consistency between related tables by ensuring the relationship between them.

3. UNIQUE

- The UNIQUE constraint ensures that all values in a column are different from each other.
- Unlike the primary key, a column with a UNIQUE constraint can contain NULL values (depending on the database).
- A table can have multiple unique constraints.

```sql
CREATE TABLE user_rec (
    record_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age TINYINT UNSIGNED NOT NULL,
    gender ENUM('male', 'female') NOT NULL,
    address VARCHAR(150),
    contact CHAR(10) NOT NULL,
    email VARCHAR(100) UNIQUE,
    rolebased_id VARCHAR(20) NOT NULL,
    general_purp_id VARCHAR(20),
    subscription_id VARCHAR(25) NOT NULL,
    emergency_contact CHAR(10) NOT NULL,
    insurance_id VARCHAR(15)
);
```

(adding foreign key with alter command)

```sql
ALTER TABLE health_records
add CONSTRAINT fk_users_gen_purp_id
FOREIGN KEY (user_gen_id) REFERENCES user_rec(general_purp_id)
ON DELETE CASCADE
ON UPDATE CASCADE;
```

First alter command will set unique constraint to that column.

ON DELETE CASCADE

- When a record in the parent table is deleted, all corresponding rows in the child table that reference the deleted row are also deleted automatically.
- Useful when you want to ensure that if a record is deleted from the parent table, no orphaned records remain in the child table.

ON UPDATE CASCADE

- When a record in the parent table is updated, all corresponding rows in the child table are updated automatically to reflect the changes.
- Useful when you want to propagate updates (such as changes to a primary key) through the related tables.

4. NOT NULL

- The NOT NULL constraint ensures that a column cannot have a NULL value.
- This constraint is used to enforce that a column must contain a value when inserting or updating a row.

```
CREATE TABLE insurance (
    record_id INT AUTO_INCREMENT PRIMARY KEY,
    insurance_id VARCHAR(15) NOT NULL UNIQUE,
    user_name VARCHAR(50) NOT NULL,
    user_general_id VARCHAR(20) NOT NULL,
    provider_name VARCHAR(50) NOT NULL,
    provider_contact CHAR(10) ,
    provider_email VARCHAR(50) NOT NULL,
    coverage VARCHAR(299),
    validity VARCHAR(10) CHECK (validity IN ('Yes', 'No')),
    issued_date DATE NOT NULL,
    exp_date DATE,
    claims VARCHAR(100) CHECK (claims IN ('Approved', 'Pending', 'Rejected'))
);
```


5. CHECK

- The CHECK constraint ensures that the value in a column meets a specific condition or set of conditions.
- It can be used to enforce rules such as numeric ranges or specific patterns in data.

```
create table subscription(
record_id int auto_increment primary key,
subscription_id varchar(25) not null,
user_gen_id varchar(20) not null,
sub_plan varchar(20) check(sub_plan in('monthly','quarterly','yearly','life-time')),
sub_type varchar(20) check(sub_type in('free','basic','premium','exclusive')),
validity varchar(10) check(validity in('valid','expired')),
coupon varchar(15),
exp_date date
);
```

6. DEFAULT

- The DEFAULT constraint sets a default value for a column when no value is provided for that column during an insert operation.
- The default value will be applied if the column is omitted in the insert statement.

```
CREATE TABLE store_inventory (
    product_id varchar(20) PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    product_description TEXT,
    product_price DECIMAL(10, 2) NOT NULL,
    quantity INT NOT NULL,
    manufacturer VARCHAR(100),
    category VARCHAR(50),
    availability BOOLEAN DEFAULT TRUE,
    status VARCHAR(20),
    exp_date DATE,
    storage_req VARCHAR(20),
    batch_no VARCHAR(50),
    created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```


8. AUTO_INCREMENT

- An AUTO_INCREMENT (MySQL) is used to automatically generate unique values for a column, typically for primary keys.
- This is commonly used for ID columns where the database will automatically provide a unique identifier for each new row.

```
create table credential(
record_id int auto_increment primary key,
user_name varchar(50),
general_purp_id varchar(20),
rolebased_id varchar(20),
insurance_id varchar(15),
password varchar(15),
mobile_num varchar(10)
);
```

4. Data Insertion and Manipulation

1. Basic INSERT Statement

This syntax is used to insert data into all columns of a table.

```
create table roles(
role_id varchar(15) primary key,
role_name varchar(50),
honour varchar(10)
);

insert into roles
values('dr.1001','physician','PHY'),
('pt.1002','patient','PAT'),
('pt.1003','HealthContentCreator','HCC'),
('res.1004','Researcher','RES'),
('ss.1005','ServiceProvider','SPV'),
('prt.1002','Practitioner','PRC');
```


2. INSERT with Specific Column Names

If you want to insert values into only specific columns, you can specify the columns in the INSERT statement. This is useful when not all columns are required to have values (for example, when there are NULL values or default values).

INSERT INTO table_name (column1, column2, column3, ...)

VALUES (value1, value2, value3, ...);

3. INSERT Multiple Rows

You can insert multiple rows in a single INSERT statement by separating each set of values with commas.

INSERT INTO table_name (column1, column2, column3, ...)

VALUES

(value1, value2, value3, ...),

(value4, value5, value6, ...),

(value7, value8, value9, ...);

```
INSERT INTO credential (user_name, general_purp_id, rolebased_id, insurance_id, password, mobile_num) VALUES
('Rajesh Kumar', 'GEN_RAJA_A001', 'PAT_001A', 'INS_A1234', 'p@ssRajK123', '9876543210'),
('Sneha Sharma', 'GEN_SNEH_B002', 'PHY_002B', 'INS_B2345', 'p@ssSnehaS789', '9823456789'),
('Amit Patel', 'GEN_AMIT_C003', 'PRC_003C', 'INS_C3456', 'p@ssAmitP456', '9871234560'),
('Pooja Reddy', 'GEN_POOJ_D004', 'RES_004D', 'INS_D4567', 'p@ssPoojaR123', '9865432101'),
('Manish Verma', 'GEN_MANI_E005', 'SPV_005E', 'INS_E5678', 'p@ssManishV678', '9843210987'),
('Anjali Gupta', 'GEN_ANJA_F006', 'HCC_006F', 'INS_F6789', 'p@ssAnjaliG890', '9812345678'),
('Vikram Singh', 'GEN_VIKR_G007', 'PAT_007G', 'INS_G7890', 'p@ssVikramS321', '9887654321'),
('Neha Joshi', 'GEN_NEHA_H008', 'PHY_008H', 'INS_H8901', 'p@ssNehaJ098', '9832109876'),
('Suresh Pillai', 'GEN_SURE_I009', 'PRC_009I', 'INS_I9012', 'p@ssSureshP345', '9876543012'),
('Radhika Mehta', 'GEN_RADH_J010', 'RES_010J', 'INS_J0123', 'p@ssRadhikaM876', '9823456712'),
('Akash Rao', 'GEN_AKAS_K011', 'SPV_011K', 'INS_K1234', 'p@ssAkashR543', '9867012345'),
('Priya Sethi', 'GEN_PRIY_L012', 'HCC_012L', 'INS_L2345', 'p@ssPriyaS567', '9845123678'),
('Rahul Deshmukh', 'GEN_RAHU_M013', 'PAT_013M', 'INS_M3456', 'p@ssRahulD234', '9832198746'),
('Nisha Jain', 'GEN_NISH_N014', 'PHY_014N', 'INS_N4567', 'p@ssNishaJ123', '9810223345'),
('Arjun Sharma', 'GEN_ARJU_O015', 'PRC_015O', 'INS_O5678', 'p@ssArjunS678', '9876453102'),
('Kiran Yadav', 'GEN_KIRA_P016', 'RES_016P', 'INS_P6789', 'p@ssKiranY432', '9887765432'),
('Deepak Rao', 'GEN_DEEP_Q017', 'SPV_017Q', 'INS_Q7890', 'p@ssDeepakR890', '9845123789'),
('Swati Kulkarni', 'GEN_SWAT_R018', 'HCC_018R', 'INS_R8901', 'p@ssSwatiK098', '9827012398'),
('Ajay Bansal', 'GEN_AJAY_S019', 'PAT_019S', 'INS_S9012', 'p@ssAjayB567', '9813456789'),
('Mona Agarwal', 'GEN_MONA_T020', 'PHY_020T', 'INS_T0123', 'p@ssMonaA456', '9890123456');
```


Updating a table :

```
UPDATE appointment
SET charges = 200.00
WHERE user_age BETWEEN 25 AND 50
  AND time_slat = 'morning'
  AND mode = 'In-Person';
select * from appointment;
```

record_id	user_gen_id	user_name	user_age	agenda	physician_id	physician_name	mode	time_slat	charges
1	GEN_RAJA_A001	Rajesh Kumar	35	General checkup	PHY_002B	Dr. Sneha Sharma	In-Person	Morning	200.00
2	GEN_SNEH_B002	Sneha Sharma	28	Consultation on skin rash	PHY_002B	Dr. Sneha Sharma	Online	Afternoon	NULL
3	GEN_AMIT_C003	Amit Patel	40	Heart disease consultation	PHY_003C	Dr. Amit Patel	In-Person	Evening	NULL
4	GEN_POOJ_D004	Pooja Reddy	31	Blood test and general consultation	PHY_004D	Dr. Pooja Reddy	Online	Morning	NULL
5	GEN_MANI_E005	Manish Verma	37	Orthopedic consultation	PHY_005E	Dr. Manish Verma	In-Person	Afternoon	NULL
6	GEN_ANJA_F006	Anjali Gupta	29	Eye exam consultation	PHY_006F	Dr. Anjali Gupta	Online	Evening	NULL
7	GEN_VIKR_G007	Vikram Singh	45	Routine health checkup	PHY_007G	Dr. Vikram Singh	In-Person	Morning	200.00
8	GEN_NEHA_H008	Neha Joshi	34	Consultation on back pain	PHY_008H	Dr. Neha Joshi	Online	Afternoon	NULL
9	GEN_SURE_I009	Suresh Pillai	50	Cancer screening	PHY_009I	Dr. Suresh Pillai	In-Person	Evening	NULL
10	GEN_RADH_J010	Radhika Mehta	27	Consultation on allergies	PHY_010J	Dr. Radhika Mehta	Online	Morning	NULL
11	GEN_AKAS_K011	Akash Rao	33	Cardiac consultation	PHY_011K	Dr. Akash Rao	In-Person	Afternoon	NULL
12	GEN_PRIY_L012	Priya Sethi	30	Routine checkup	PHY_012L	Dr. Priya Sethi	Online	Evening	NULL
13	GEN_RAHU_M013	Rahul Deshmukh	39	General health checkup	PHY_013M	Dr. Rahul Deshmukh	In-Person	Morning	200.00
14	GEN_NISH_N014	Nisha Jain	41	Consultation on stress management	PHY_014N	Dr. Nisha Jain	Online	Afternoon	NULL
15	GEN_ARJU_O015	Arjun Sharma	38	Diabetes management consultation	PHY_015O	Dr. Arjun Sharma	In-Person	Evening	NULL
16	GEN_KIRA_P016	Kiran Yadav	32	Routine checkup	PHY_016P	Dr. Kiran Yadav	Online	Morning	NULL
17	GEN_DEEP_Q017	Deepak Rao	36	Orthopedic consultation	PHY_017Q	Dr. Deepak Rao	In-Person	Afternoon	NULL
18	GEN_SWAT_R018	Swati Kulkarni	26	Consultation for skin issues	PHY_018R	Dr. Swati Kulkarni	Online	Evening	NULL
19	GEN_AJAY_S019	Ajay Bansal	42	Routine health checkup	PHY_019S	Dr. Ajay Bansal	In-Person	Morning	200.00
20	GEN_MONA_T020	Mona Agarwal	35	Consultation for menstrual problems	PHY_020T	Dr. Mona Agarwal	Online	Afternoon	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Modifying structure of table :

1. Adding a New Column

To add a new column to an existing table, use the ADD clause.

ALTER TABLE table_name

ADD column_name datatype [constraints];

```
alter table appointment add charges decimal(5,2);
```

2. Modifying an Existing Column

You can change the data type, length, or constraints of an existing column using the MODIFY or ALTER COLUMN clause (depending on the database).

ALTER TABLE table_name

MODIFY column_name new_datatype

3. Renaming a Column

To rename an existing column, use the RENAME COLUMN clause. Not all databases use the same syntax for renaming columns.

ALTER TABLE table_name

RENAME COLUMN old_column_name TO new_column_name;

```
alter table appointment rename column agenda to purpose;
```

Arithmetic Operator

- **Arithmetic Operators:**
  - **Addition (+):** Adds two values.
  - **Subtraction (-):** Subtracts one value from another.
  - **Multiplication (*):** Multiplies two values.
  - **Division (/):** Divides one value by another.

- **Modulo (%):** Returns the remainder of a division operation.

- **Comparison Operators:**
  - **Equal to (==):** Checks if two values are equal.
  - **Not equal to (!= or <>):** Checks if two values are not equal.
  - **Greater than (>):** Checks if one value is greater than another.
  - **Less than (<):** Checks if one value is less than another.
  - **Greater than or equal to (>=):** Checks if one value is greater than or equal to another.
  - **Less than or equal to (<=):** Checks if one value is less than or equal to another.
- **Logical Operators:**
  - **AND (&& or AND):** Returns true if both conditions are true.
  - **OR (|| or OR):** Returns true if at least one condition is true.
  - **NOT (! or NOT):** Reverses the result of the condition.

- **EQUALS (=)**:
  - Checks if two values are equal.
  - Example: `SELECT * FROM table WHERE column1 = 'value';`

```
select * from user_rec
where name = 'rajesh kumar';
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

record_id	name	age	gender	address	contact	email	rolebased_id	general_purp_id	subscription_id	emergency_contact	insurance_id
1	Rajesh Kumar	35	male	123 MG Road, Delhi	9876543210	rajesh.kumar@example.com	PAT_001A	GEN_RAJA_A001	SUB_BASIC_A1	9123456780	INS_A1234
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

- **NOT EQUALS (<> or !=)**:
  - Checks if two values are not equal.
  - Example: `SELECT * FROM table WHERE column1 <> 'value';`

```
select * from user_rec
where  gender <> 'male';
```

record_id	name	age	gender	address	contact	email	rolebased_id	general_purp_id	subscription_id	emergency_contact	insurance_id
2	Sneha Sharma	28	female	45 Residency Rd, Mumbai	9823456789	sneha.sharma@example.com	PHY_002B	GEN_SNEH_B002	SUB_PREMIUM_B2	9129876543	INS_B2345
4	Pooja Reddy	31	female	90 Jubilee Hills, Hyderabad	9865432101	pooja.reddy@example.com	RES_004D	GEN_POOJ_D004	SUB_BASIC_D4	9176543210	INS_D4567
6	Anjali Gupta	29	female	54 Civil Lines, Jaipur	9812345678	anjali.gupta@example.com	HCC_006F	GEN_ANJA_F006	SUB_BASIC_F6	9154321098	INS_F6789
8	Neha Joshi	34	female	85 Park St, Kolkata	9832109876	neha.joshi@example.com	PHY_008H	GEN_NEHA_H008	SUB_PREMIUM_H8	9132109876	INS_H8901
10	Radhika Mehta	27	female	102 Mira Road, Thane	9823456712	radhika.mehta@example.com	RES_010J	GEN_RADH_J010	SUB_BASIC_J1	9119876543	INS_J0123
12	Priya Sethi	30	female	150 Defence Colony, Chandigarh	9845123678	priya.sethi@example.com	HCC_012L	GEN_PRIY_L012	SUB_EXCLUSIVE_L3	9193456782	INS_L2345
14	Nisha Jain	41	female	210 Lavelle Rd, Bengaluru	9810223345	nisha.jain@example.com	PHY_014N	GEN_NISH_N014	SUB_PREMIUM_N5	9171223345	INS_N4567
16	Kiran Yadav	32	female	53 Andheri East, Mumbai	9887765432	kiran.yadav@example.com	RES_016P	GEN_KIRA_P016	SUB_BASIC_P7	9158765432	INS_P6789
18	Swati Kulkarni	26	female	33 Malad, Mumbai	9827012398	swati.kulkarni@example.com	HCC_018R	GEN_SWAT_R018	SUB_BASIC_R9	9129012398	INS_R8901
20	Mona Agarwal	35	female	21 Lajpat Nagar, Delhi	9890123456	mona.agarwal@example.com	PHY_020T	GEN_MONA_T020	SUB_PREMIUM_T2	9189123456	INS_T0123
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

- **BETWEEN ... AND ...**:
    - Checks if a value is within a specified range (inclusive).
    - Example: `SELECT * FROM table WHERE column1 BETWEEN 5 AND 10;`

```
select * from user_rec
where  age between 20 and 30;
```

record_id	name	age	gender	address	contact	email	rolebased_id	general_purp_id	subscription_id	emergency_contact	insurance_id
2	Sneha Sharma	28	female	45 Residency Rd, Mumbai	9823456789	sneha.sharma@example.com	PHY_002B	GEN_SNEH_B002	SUB_PREMIUM_B2	9129876543	INS_B2345
6	Anjali Gupta	29	female	54 Civil Lines, Jaipur	9812345678	anjali.gupta@example.com	HCC_006F	GEN_ANJA_F006	SUB_BASIC_F6	9154321098	INS_F6789
10	Radhika Mehta	27	female	102 Mira Road, Thane	9823456712	radhika.mehta@example.com	RES_010J	GEN_RADH_J010	SUB_BASIC_J1	9119876543	INS_J0123
12	Priya Sethi	30	female	150 Defence Colony, Chandigarh	9845123678	priya.sethi@example.com	HCC_012L	GEN_PRIY_L012	SUB_EXCLUSIVE_L3	9193456782	INS_L2345
18	Swati Kulkarni	26	female	33 Malad, Mumbai	9827012398	swati.kulkarni@example.com	HCC_018R	GEN_SWAT_R018	SUB_BASIC_R9	9129012398	INS_R8901
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Range Operator

- **BETWEEN:**
    - Syntax: `value BETWEEN low AND high`
    - Description: Checks if a value lies within a specified range inclusive of both low and high values.
    - Example: `SELECT * FROM Products WHERE Price BETWEEN 50 AND 100;`
- **NOT BETWEEN:**
    - Syntax: `value NOT BETWEEN low AND high`
    - Description: Checks if a value does not lie within a specified range inclusive of both low and high values.
    - Example: `SELECT * FROM Orders WHERE OrderDate NOT BETWEEN '2024-01-01' AND '2024-06-30';`

```
select * from appointment
where user_age between 25 and 50;
```

record_id	user_gen_id	user_name	user_age	purpose	physician_id	physician_name	mode	time_slat	charges
1	GEN_RAJA_A001	Rajesh Kumar	35	General checkup	PHY_002B	Dr. Sneha Sharma	In-Person	Morning	200.00
2	GEN_SNEH_B002	Sneha Sharma	28	Consultation on skin rash	PHY_002B	Dr. Sneha Sharma	Online	Afternoon	NULL
3	GEN_AMIT_C003	Amit Patel	40	Heart disease consultation	PHY_003C	Dr. Amit Patel	In-Person	Evening	NULL
4	GEN_POOJ_D004	Pooja Reddy	31	Blood test and general consultation	PHY_004D	Dr. Pooja Reddy	Online	Morning	NULL
5	GEN_MANI_E005	Manish Verma	37	Orthopedic consultation	PHY_005E	Dr. Manish Verma	In-Person	Afternoon	NULL
6	GEN_ANJA_F006	Anjali Gupta	29	Eye exam consultation	PHY_006F	Dr. Anjali Gupta	Online	Evening	NULL
7	GEN_VIKR_G007	Vikram Singh	45	Routine health checkup	PHY_007G	Dr. Vikram Singh	In-Person	Morning	200.00
8	GEN_NEHA_H008	Neha Joshi	34	Consultation on back pain	PHY_008H	Dr. Neha Joshi	Online	Afternoon	NULL
9	GEN_SURE_I009	Suresh Pillai	50	Cancer screening	PHY_009I	Dr. Suresh Pillai	In-Person	Evening	NULL
10	GEN_RADH_J010	Radhika Mehta	27	Consultation on allergies	PHY_010J	Dr. Radhika Mehta	Online	Morning	NULL
11	GEN_AKAS_K011	Akash Rao	33	Cardiac consultation	PHY_011K	Dr. Akash Rao	In-Person	Afternoon	NULL
12	GEN_PRIY_L012	Priya Sethi	30	Routine checkup	PHY_012L	Dr. Priya Sethi	Online	Evening	NULL
13	GEN_RAHU_M013	Rahul Deshmukh	39	General health checkup	PHY_013M	Dr. Rahul Deshmukh	In-Person	Morning	200.00
14	GEN_NISH_N014	Nisha Jain	41	Consultation on stress management	PHY_014N	Dr. Nisha Jain	Online	Afternoon	NULL
15	GEN_ARJU_O015	Arjun Sharma	38	Diabetes management consultation	PHY_015O	Dr. Arjun Sharma	In-Person	Evening	NULL
16	GEN_KIRA_P016	Kiran Yadav	32	Routine checkup	PHY_016P	Dr. Kiran Yadav	Online	Morning	NULL
17	GEN_DEEP_Q017	Deepak Rao	36	Orthopedic consultation	PHY_017Q	Dr. Deepak Rao	In-Person	Afternoon	NULL
18	GEN_SWAT_R018	Swati Kulkarni	26	Consultation for skin issues	PHY_018R	Dr. Swati Kulkarni	Online	Evening	NULL
19	GEN_AJAY_S019	Ajay Bansal	42	Routine health checkup	PHY_019S	Dr. Ajay Bansal	In-Person	Morning	200.00
20	GEN_MONA_T020	Mona Agarwal	35	Consultation for menstrual problems	PHY_020T	Dr. Mona Agarwal	Online	Afternoon	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

List Operator

- **IN:**
  - Syntax: `value IN (val1, val2, ..., valN)`
  - Description: Checks if a value matches any value in a specified list.
  - Example: `SELECT * FROM Employees WHERE Department IN ('HR', 'Finance', 'IT');`
- **NOT IN:**
  - Syntax: `value NOT IN (val1, val2, ..., valN)`
  - Description: Checks if a value does not match any value in a specified list.
  - Example: `SELECT * FROM Orders WHERE CustomerID NOT IN ('C001', 'C002', 'C003');`

```
select * from appointment
where mode in('in-person');
```

record_id	user_gen_id	user_name	user_age	purpose	physician_id	physician_name	mode	time_slat	charges
1	GEN_RAJA_A001	Rajesh Kumar	35	General checkup	PHY_002B	Dr. Sneha Sharma	In-Person	Morning	200.00
3	GEN_AMIT_C003	Amit Patel	40	Heart disease consultation	PHY_003C	Dr. Amit Patel	In-Person	Evening	250.00
5	GEN_MANI_E005	Manish Verma	37	Orthopedic consultation	PHY_005E	Dr. Manish Verma	In-Person	Afternoon	250.00
7	GEN_VIKR_G007	Vikram Singh	45	Routine health checkup	PHY_007G	Dr. Vikram Singh	In-Person	Morning	200.00
9	GEN_SURE_I009	Suresh Pillai	50	Cancer screening	PHY_009I	Dr. Suresh Pillai	In-Person	Evening	250.00
11	GEN_AKAS_K011	Akash Rao	33	Cardiac consultation	PHY_011K	Dr. Akash Rao	In-Person	Afternoon	250.00
13	GEN_RAHU_M013	Rahul Deshmukh	39	General health checkup	PHY_013M	Dr. Rahul Deshmukh	In-Person	Morning	200.00
15	GEN_ARJU_O015	Arjun Sharma	38	Diabetes management consultation	PHY_015O	Dr. Arjun Sharma	In-Person	Evening	250.00
17	GEN_DEEP_Q017	Deepak Rao	36	Orthopedic consultation	PHY_017Q	Dr. Deepak Rao	In-Person	Afternoon	250.00
19	GEN_AJAY_S019	Ajay Bansal	42	Routine health checkup	PHY_019S	Dr. Ajay Bansal	In-Person	Morning	200.00
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

5. **Data Retrieval with SQL Queries**

5.1 Basic Queries

Select statements

```
select * from appointment
```

record_id	user_gen_id	user_name	user_age	purpose	physician_id	physician_name	mode	time_slat	charges
1	GEN_RAJA_A001	Rajesh Kumar	35	General checkup	PHY_002B	Dr. Sneha Sharma	In-Person	Morning	200.00
2	GEN_SNEH_B002	Sneha Sharma	28	Consultation on skin rash	PHY_002B	Dr. Sneha Sharma	Online	Afternoon	NULL
3	GEN_AMIT_C003	Amit Patel	40	Heart disease consultation	PHY_003C	Dr. Amit Patel	In-Person	Evening	250.00
4	GEN_POOJ_D004	Pooja Reddy	31	Blood test and general consultation	PHY_004D	Dr. Pooja Reddy	Online	Morning	NULL
5	GEN_MANI_E005	Manish Verma	37	Orthopedic consultation	PHY_005E	Dr. Manish Verma	In-Person	Afternoon	250.00
6	GEN_ANJA_F006	Anjali Gupta	29	Eye exam consultation	PHY_006F	Dr. Anjali Gupta	Online	Evening	NULL
7	GEN_VIKR_G007	Vikram Singh	45	Routine health checkup	PHY_007G	Dr. Vikram Singh	In-Person	Morning	200.00
8	GEN_NEHA_H008	Neha Joshi	34	Consultation on back pain	PHY_008H	Dr. Neha Joshi	Online	Afternoon	NULL
9	GEN_SURE_I009	Suresh Pillai	50	Cancer screening	PHY_009I	Dr. Suresh Pillai	In-Person	Evening	250.00
10	GEN_RADH_J010	Radhika Mehta	27	Consultation on allergies	PHY_010J	Dr. Radhika Mehta	Online	Morning	NULL
11	GEN_AKAS_K011	Akash Rao	33	Cardiac consultation	PHY_011K	Dr. Akash Rao	In-Person	Afternoon	250.00
12	GEN_PRIY_L012	Priya Sethi	30	Routine checkup	PHY_012L	Dr. Priya Sethi	Online	Evening	NULL
13	GEN_RAHU_M013	Rahul Deshmukh	39	General health checkup	PHY_013M	Dr. Rahul Deshmukh	In-Person	Morning	200.00
14	GEN_NISH_N014	Nisha Jain	41	Consultation on stress management	PHY_014N	Dr. Nisha Jain	Online	Afternoon	NULL
15	GEN_ARJU_O015	Arjun Sharma	38	Diabetes management consultation	PHY_015O	Dr. Arjun Sharma	In-Person	Evening	250.00
16	GEN_KIRA_P016	Kiran Yadav	32	Routine checkup	PHY_016P	Dr. Kiran Yadav	Online	Morning	NULL
17	GEN_DEEP_Q017	Deepak Rao	36	Orthopedic consultation	PHY_017Q	Dr. Deepak Rao	In-Person	Afternoon	250.00
18	GEN_SWAT_R018	Swati Kulkarni	26	Consultation for skin issues	PHY_018R	Dr. Swati Kulkarni	Online	Evening	NULL
19	GEN_AJAY_S019	Ajay Bansal	42	Routine health checkup	PHY_019S	Dr. Ajay Bansal	In-Person	Morning	200.00
20	GEN_MONA_T020	Mona Agarwal	35	Consultation for menstrual problems	PHY_020T	Dr. Mona Agarwal	Online	Afternoon	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Filtering data with WHERE clause

```
51 •  select * from appointment
52    where user_age = 28 ;
53
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

record_id	user_gen_id	user_name	user_age	purpose	physician_id	physician_name	mode	time_slat	charges
2	GEN_SNEH_B002	Sneha Sharma	28	Consultation on skin rash	PHY_002B	Dr. Sneha Sharma	Online	Afternoon	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

6.2 Sorting and Limiting Results

ORDER BY , Distinct LIMIT clauses

- **ORDER BY Clause:**
  - **Description:**
    - The ORDER BY clause is used in SQL to sort the result set of a query in ascending or descending order based on one or more columns.

- **Syntax:**
  - `SELECT column1, column2, ... FROM table_name ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;`
- **ASC and DESC:**
  - ASC (ascending) is the default sorting order.
  - DESC (descending) sorts in reverse order.

```
select order_id,amount from order_records
order by amount asc;
```

order_id	amount
ORD016	75.50
ORD005	85.00
ORD003	90.00
ORD019	98.00
ORD018	105.00
ORD007	110.00
ORD001	120.50
ORD009	125.75

- **DISTINCT Keyword:**
  - **Description:**
    - The DISTINCT keyword is used to retrieve unique values from a specified column or combination of columns in a query result set.
  - **Syntax:**
    - `SELECT DISTINCT column1, column2, ... FROM table_name;`
  - **Usage:**
    - Eliminates duplicate rows from the result set.

```
select distinct(order_id),amount from order_records;
```

order_id	amount
ORD001	120.50
ORD002	299.99
ORD003	90.00
ORD004	450.75

Built in Functions

String Functions

- **CONCAT(str1, str2, ...)**:
  - Concatenates two or more strings together.
  - Example: `SELECT CONCAT('Hello', ' ', 'World');` -> `Hello World`

```
select concat(general_purp_id,'    ',name) as user_details from user_rec;
```

	user_details
▶	GEN_RAJA_A001 Rajesh Kumar
	GEN_SNEH_B002 Sneha Sharma
	GEN_AMIT_C003 Amit Patel
	GEN_POOJ_D004 Pooja Reddy
	GEN_MANI_E005 Manish Verma
	GEN_ANJA_F006 Anjali Gupta
	GEN_VIKR_G007 Vikram Singh

- **UPPER(str)**:
    - Converts a string to uppercase.
    - Example: `SELECT UPPER('hello');` -> `HELLO`

```
select upper(name) from user_rec;
```

	upper(name)
▶	RAJESH KUMAR
	SNEHA SHARMA

- **LOWER(str)**:
    - Converts a string to lowercase.
    - Example: `SELECT LOWER('HELLO');` -> `hello`

```
select lower(name) from user_rec;
```

- **LENGTH(str)**:
    - Returns the length of a string.
    - Example: `SELECT LENGTH('Hello World');` -> `11`

```
select length(name) from user_rec;
```

	length(name)
▶	12
	12
	10
	11

Math Operator

ROUND(num, decimal_places):

- Rounds a numeric value to a specified number of decimal places.

```
select round(amount) from order_records limit 5;
```

	round(amount)
▶	121
	300
	90
	451
	85

	amount
▶	120.50
	299.99
	90.00
	450.75
	85.00

FLOOR(num):

- Returns the largest integer less than or equal to a given number.

```
mysql> SELECT FLOOR(5.9);
+------------+
| FLOOR(5.9) |
+------------+
|          5 |
+------------+
1 row in set (0.00 sec)
```

Date Function

CURRENT_DATE:

- Returns the current date (without time).

```
mysql> SELECT CURRENT_DATE;
+--------------+
| CURRENT_DATE |
+--------------+
| 2024-11-13   |
+--------------+
1 row in set (0.00 sec)
```

CURRENT_TIME:

- Returns the current time (without date).

```
mysql> SELECT CURRENT_time;
+--------------+
| CURRENT_time |
+--------------+
| 18:45:09     |
+--------------+
1 row in set (0.00 sec)
```

CURRENT_TIMESTAMP:

- Returns the current date and time.

```
mysql> SELECT CURRENT_timestamp;
+---------------------+
| CURRENT_timestamp   |
+---------------------+
| 2024-11-13 18:45:43 |
+---------------------+
1 row in set (0.00 sec)
```

Aggregate Function

- **COUNT()**:
    - Counts the number of rows that match a specified condition.
    - Example: `SELECT COUNT(*) FROM table_name;`

```
mysql> select count(*) from user_rec;
+----------+
| count(*) |
+----------+
|       20 |
+----------+
1 row in set (0.01 sec)
```

- **SUM()**:
    - Calculates the sum of values in a column.
    - Example: `SELECT SUM(column_name) FROM table_name;`

```
mysql> select sum(product_price) from store_inventory;
+--------------------+
| sum(product_price) |
+--------------------+
|             486.15 |
+--------------------+
1 row in set (0.00 sec)
```

- **AVG()**:
    - Calculates the average of values in a column.
    - Example: `SELECT AVG(column_name) FROM table_name;`

```
mysql> select avg(product_price) from store_inventory;
+--------------------+
| avg(product_price) |
+--------------------+
|          18.005556 |
+--------------------+
1 row in set (0.00 sec)
```

- **MIN()**:
    - Returns the minimum value in a column.
    - Example: `SELECT MIN(column_name) FROM table_name;`

```
mysql> select min(product_price) from store_inventory;
+--------------------+
| min(product_price) |
+--------------------+
|               5.99 |
+--------------------+
1 row in set (0.00 sec)
```

- **MAX()**:
    - Returns the maximum value in a column.
    - Example: `SELECT MAX(column_name) FROM table_name;`

```
mysql> select max(product_price) from store_inventory;
+--------------------+
| max(product_price) |
+--------------------+
|              45.50 |
+--------------------+
1 row in set (0.00 sec)
```

GROUP BY Clause with HAVING

Learn:

- **Description:**
    - The `GROUP BY` clause in SQL is used to group rows that have the same values into summary rows, typically used with aggregate functions like `SUM()`, `COUNT()`, `AVG()`, etc. The `HAVING` clause further filters the grouped rows based on specified conditions.
- **Usage:**
    - **GROUP BY Clause:**
        - Groups rows that have the same values into summary rows.
        - Example: `SELECT column1, SUM(column2) FROM table_name GROUP BY column1;`
    - **HAVING Clause:**
        - Filters groups based on specified conditions after the `GROUP BY` clause.
        - Example: `SELECT column1, SUM(column2) FROM table_name GROUP BY column1 HAVING SUM(column2) > 1000;`
- **Order of Execution:**
    1. **FROM**: Specifies the tables and views from which to retrieve data.
    - **WHERE**: Filters rows before grouping with conditions.
    3. **GROUP BY**: Groups rows based on specified columns.
    4. **HAVING**: Filters groups based on specified conditions.
    5. **SELECT**: Retrieves columns or expressions for output.
    6. **ORDER BY**: Sorts the result set based on specified columns or expressions.

```sql
SELECT
    product_cat,
    COUNT(order_id) AS total_orders,
    SUM(amount) AS total_amount_spent
FROM
    order_records
GROUP BY
    product_cat
HAVING
    SUM(amount) > 1000;
```

	product_cat	total_orders	total_amount_spent
▶	Pharma	10	1216.75
	Face Care	10	3310.62

Complex Queries

1) Sub query

- **Description:**
    1. A subquery, also known as a nested query or inner query, is a query nested within another SQL statement (outer query). It allows you to retrieve data from one or more tables based on a condition or criteria defined in the subquery. The result of the subquery is used by the outer query to further filter or manipulate data.
- **Types of Subqueries:**

1.Single-Row Subquery:

- Returns zero or one row to the outer query

```sql
select product_id,product_name,product_price
from store_inventory
where product_price >= (select avg(Product_price) from store_inventory)
order by product_price asc
limit 5;
```

	product_id	product_name	product_price
▶	FC006	Hydrating Toner	18.50
	PH012	Probiotic 10B CFU	18.99
	FC003	Sunscreen SPF 50	19.99
	PH007	Multivitamins	20.99
	PH014	Fish Oil 1000mg	21.50
*	NULL	NULL	NULL

2. Multiple Row Subquery

A **multiple row subquery** returns multiple rows and is used with operators like `IN`, `ANY`, or `ALL`.

```
select product_id,product_name,amount
from order_records
where product_id in (select product_id from order_records
where order_id in (select order_id from order_records where amount > 250))
order by amount asc
limit 5;
```

	product_id	product_name	amount
▶	FC004	Brightening Mask	260.00
	FC010	Vitamin C Serum	299.99
	FC001	Moisturizing Cream	320.40
	FC003	Sunscreen SPF 50	450.75
	FC007	Anti-Pollution Serum	460.00

2) Joins

- **Description:**
    - Joins in SQL are used to combine rows from two or more tables based on a related column between them. They allow you to retrieve data from multiple tables simultaneously and create a single result set.

- **Types of Joins:**
    - **Inner Join:** Returns only the rows that have matching values in both tables.

```sql
select insurance.user_name,insurance.insurance_id,appointment.physician_id,
    appointment.agenda as purpose,appointment.mode
from insurance inner join appointment
on insurance.user_general_id=appointment.user_gen_id
limit 5;
```

user_name	insurance_id	physician_id	purpose	mode
Rajesh Kumar	INS_A1234	PHY_002B	General checkup	In-Person
Sneha Sharma	INS_B2345	PHY_002B	Consultation on skin rash	Online
Amit Patel	INS_C3456	PHY_003C	Heart disease consultation	In-Person
Pooja Reddy	INS_D4567	PHY_004D	Blood test and general consultation	Online
Manish Verma	INS_E5678	PHY_005E	Orthopedic consultation	In-Person

- **Left Join (or Left Outer Join):** Returns all rows from the left table and the matched rows from the right table. If no match is found, NULL values are returned for the right table columns.

```sql
SELECT
    insurance.insurance_id,
    insurance.user_name,
    appointment.time_slat,
    appointment.mode
FROM
    insurance
LEFT JOIN
    appointment
ON
    insurance.user_general_id = appointment.user_gen_id;
```

insurance_id	user_name	time_slat	mode
INS_A1235	Rajesh Kumar	Morning	In-Person
INS_B2346	Sneha Sharma	Afternoon	Online
INS_C3457	Amit Patel	Evening	In-Person
INS_D4568	Pooja Reddy	Morning	Online
INS_E5679	Manish Verma	Afternoon	In-Person
INS_F6790	Anjali Gupta	Evening	Online
INS_G7891	Vikram Singh	Morning	In-Person
INS_H8902	Neha Joshi	Afternoon	Online
INS_I9013	Suresh Pillai	Evening	In-Person
INS_J0124	Radhika Mehta	Morning	Online
INS_K1235	Akash Rao	Afternoon	In-Person
INS_L2346	Priya Sethi	Evening	Online
INS_M3457	Rahul Deshmukh	Morning	In-Person
INS_N4568	Nisha Jain	Afternoon	Online
INS_O5679	Arjun Sharma	Evening	In-Person
INS_P6789	Kiran Yadav	Morning	Online
INS_Q7890	Deepak Rao	Afternoon	In-Person
INS_R8902	Swati Kulkarni	Evening	Online
INS_S9013	Ajay Bansal	Morning	In-Person
INS_T0124	Mona Agarwal	Afternoon	Online

- **Right Join (or Right Outer Join):** Returns all rows from the right table and the matched rows from the left table. If no match is found, NULL values are returned for the left table columns.

```
SELECT
    insurance.insurance_id,
    insurance.user_name,
    appointment.time_slat,
    appointment.mode
FROM
    insurance
RIGHT JOIN
    appointment
ON
    insurance.user_general_id = appointment.user_gen_id;
```

insurance_id	user_name	time_slat	mode
INS_A1235	Rajesh Kumar	Morning	In-Person
INS_B2346	Sneha Sharma	Afternoon	Online
INS_C3457	Amit Patel	Evening	In-Person
INS_D4568	Pooja Reddy	Morning	Online
INS_E5679	Manish Verma	Afternoon	In-Person
INS_F6790	Anjali Gupta	Evening	Online
INS_G7891	Vikram Singh	Morning	In-Person
INS_H8902	Neha Joshi	Afternoon	Online
INS_I9013	Suresh Pillai	Evening	In-Person
INS_J0124	Radhika Mehta	Morning	Online
INS_K1235	Akash Rao	Afternoon	In-Person
INS_L2346	Priya Sethi	Evening	Online
INS_M3457	Rahul Deshmukh	Morning	In-Person
INS_N4568	Nisha Jain	Afternoon	Online
INS_O5679	Arjun Sharma	Evening	In-Person
INS_P6789	Kiran Yadav	Morning	Online
INS_Q7890	Deepak Rao	Afternoon	In-Person
INS_R8902	Swati Kulkarni	Evening	Online
INS_S9013	Ajay Bansal	Morning	In-Person
INS_T0124	Mona Agarwal	Afternoon	Online

Purpose of Self Join

In this scenario, the goal is to find pairs of orders that:

1. Belong to the same product category (like "Pharma" or "Face Care").
2. Were placed by different users.
3. Are not the same order (i.e., have different order_id values).

```
SELECT
    o1.order_id AS Order1_ID,
    o2.order_id AS Order2_ID,
    o1.user_gen_id AS User1_Gen_ID,
    o2.user_gen_id AS User2_Gen_ID,
    o1.product_id AS Product_ID,
    o1.product_name AS Product_Name,
    o1.product_cat AS Product_Category
FROM
    order_records o1
JOIN
    order_records o2
ON
    o1.product_cat = o2.product_cat
    AND o1.order_id != o2.order_id
    AND o1.user_gen_id != o2.user_gen_id
ORDER BY
    o1.product_cat, o1.order_id
LIMIT 5;
```

Order1_ID	Order2_ID	User1_Gen_ID	User2_Gen_ID	Product_ID	Product_Name	Product_Category
ORD002	ORD008	GEN_SNEH_B002	GEN_NEHA_H008	FC010	Vitamin C Serum	Face Care
ORD002	ORD012	GEN_SNEH_B002	GEN_PRIY_L012	FC010	Vitamin C Serum	Face Care
ORD002	ORD004	GEN_SNEH_B002	GEN_POOJ_D004	FC010	Vitamin C Serum	Face Care
ORD002	ORD006	GEN_SNEH_B002	GEN_ANJA_F006	FC010	Vitamin C Serum	Face Care
ORD002	ORD010	GEN_SNEH_B002	GEN_RADH_J010	FC010	Vitamin C Serum	Face Care

Benefits of Using SELF JOIN

- **Comparison**: Helps in comparing rows from the same dataset (e.g., finding potential conflicts or similarities).
- **Grouping**: Useful to group and filter data based on conditions involving relationships within the same table (like products of the same category but different users).
- **Analysis**: Allows deeper analysis within a single dataset without needing another table.

Views in SQL

- **Definition:**
  - Views in SQL are virtual tables generated from a SELECT query's result set. They do not store data themselves but present data from one or more underlying tables or other views.
  - Views allow users to access and manipulate data stored in the underlying tables through a simplified, customized interface.
- **Purpose:**
  - Views provide a way to:
    - Simplify complex queries by abstracting underlying table structures.
    - Restrict access to sensitive data by exposing only necessary columns to users.
    - Aggregate data from multiple tables into a single virtual table for easier reporting or analysis.
- **Types of Views:**
  - **Simple Views:** Based on a single table and may contain all rows or a subset based on a condition.
  - **Complex Views:** Derived from multiple tables using joins, aggregations, or subqueries.

- **Simple View:**
  - A Simple View is based on a single table and shows all or selected columns of that table. It may also filter rows based on a condition.
  - It provides a virtual representation of a table with limited complexity.

```
CREATE VIEW Pending_Orders AS
SELECT
    order_id,
    user_gen_id,
    product_name,
    product_cat,
    product_quantity,
    shipping_status,
    shipping_address,
    amount
FROM
    order_records
WHERE
    shipping_status = 'pending';
```

order_id	user_gen_id	product_name	product_cat	product_quantity	shipping_status	shipping_address	amount
ORD001	GEN_RAJA_A001	Paracetamol 500mg	Pharma	2	pending	123 MG Road, Delhi	120.50
ORD004	GEN_POOJ_D004	Sunscreen SPF 50	Face Care	2	pending	90 Jubilee Hills, Hyderabad	450.75
ORD007	GEN_VIKR_G007	Cough Syrup	Pharma	2	pending	120 Cantonment Area, Pune	110.00
ORD010	GEN_RADH_J010	Anti-Aging Serum	Face Care	1	pending	102 Mira Road, Thane	599.50
ORD013	GEN_RAHU_M013	Cetirizine 10mg	Pharma	3	pending	35 Sharanpur Road, Nashik	135.00
ORD016	GEN_KIRA_P016	Lip Balm	Face Care	3	pending	53 Andheri East, Mumbai	75.50
ORD019	GEN_AJAY_S019	Magnesium 300mg	Pharma	1	pending	98 CG Road, Ahmedabad	98.00

- **Complex View:**
  - A Complex View involves multiple tables joined together, or it might include aggregations, subqueries, or functions.
  - Complex Views can be used to retrieve and present data from different tables in a consolidated manner.

```
CREATE VIEW User_Category_Spendings AS
SELECT
    u.name,
    u.email,
    o.product_cat,
    SUM(o.amount) AS total_spent,
    COUNT(o.order_id) AS total_orders
FROM
    order_records o
JOIN
    user_rec u
ON
    o.user_gen_id = u.general_purp_id
GROUP BY
    u.name,
    u.email,
    o.product_cat
ORDER BY
    u.name,
    o.product_cat;
```

```
SELECT * FROM User_Category_Spending
limit 8;
```

name	email	product_cat	total_spent	total_orders
Ajay Bansal	ajay.bansal@example.com	Pharma	98.00	1
Akash Rao	akash.rao@example.com	Pharma	150.00	1
Amit Patel	amit.patel@example.com	Pharma	90.00	1
Anjali Gupta	anjali.gupta@example.com	Face Care	320.40	1
Arjun Sharma	arjun.sharma@example.com	Pharma	220.00	1
Deepak Rao	deepak.rao@example.com	Pharma	190.00	1
Kiran Yadav	kiran.yadav@example.com	Face Care	75.50	1
Manish Verma	manish.verma@example.com	Pharma	85.00	1

1. Check Constraints in Views

A Check Constraint ensures that data being modified through the view meets specific conditions.

```
CREATE VIEW Expensive_Product AS
SELECT product_id, product_name, product_price
FROM store_inventory
WHERE product_price >= 10
WITH CHECK OPTION;

select * from expensive_product;
```

product_id	product_name	product_price
FC001	Moisturizing Cream	25.99
FC002	Anti-Aging Serum	45.50
FC003	Sunscreen SPF 50	19.99
FC004	Brightening Mask	22.99
FC005	Face Cleanser	15.00
FC006	Hydrating Toner	18.50
FC007	Anti-Pollution Serum	40.00
FC008	Night Cream	28.99

2. Unique Constraints in Views

Unique constraints ensure that a column or a combination of columns has unique values.

```
CREATE VIEW Unique_User_Emails AS
SELECT general_purp_id, name, email
FROM user_rec;

select * from Unique_User_Emails limit 8;
```

general_purp_id	name	email
GEN_RAJA_A001	Rajesh Kumar	rajesh.kumar@example.com
GEN_SNEH_B002	Sneha Sharma	sneha.sharma@example.com
GEN_AMIT_C003	Amit Patel	amit.patel@example.com
GEN_POOJ_D004	Pooja Reddy	pooja.reddy@example.com
GEN_MANI_E005	Manish Verma	manish.verma@example.com
GEN_ANJA_F006	Anjali Gupta	anjali.gupta@example.com
GEN_VIKR_G007	Vikram Singh	vikram.singh@example.com
GEN_NEHA_H008	Neha Joshi	neha.joshi@example.com

Stored Procedure

A stored procedure is a precompiled set of SQL statements that can be stored in the database and executed on demand. It is essentially a program that resides in the database and performs operations such as querying, inserting, updating, or deleting data. Stored procedures allow database operations to be executed in a modular, reusable way**.**

Modularity:

- Stored procedures allow for modular programming. You can break down complex tasks into smaller procedures that can be reused, making the database code easier to manage and maintain.

Performance:

- Since stored procedures are precompiled, they can execute faster than sending individual SQL queries from an application. The database server compiles the SQL statements once and keeps them ready for future execution.

Security:

- Stored procedures help enhance database security by restricting direct access to the database tables. Users can be granted permission to execute stored procedures without directly accessing the underlying data.

Maintainability:

- They provide a single point of change. If the business logic in a stored procedure needs to change, you can modify it in one place, and it will reflect wherever the procedure is used.

Types of Parameters in Stored Procedures

1. **IN:** Input parameters, which allow values to be passed to the procedure.
2. **OUT:** Output parameters, which allow the procedure to return a value.
3. **INOUT:** A parameter that can be used both for input and output.

Scenario Example

Imagine the following situation in the pharmacy:

1. **Regular Inventory Check:**
    - The inventory manager wants to see which products in the 'Pharma' category are currently not available.
    - The pharmacy often faces supply chain issues, so keeping track of unavailable items is crucial to ensure restocking happens in time.

```
DELIMITER $$

CREATE PROCEDURE GetNotAvailableProductsByCategory(IN input_category VARCHAR(50))
BEGIN
    SELECT
        product_id,
        product_name,
        product_description,
        product_price,
        quantity,
        manufacturer,
        status
    FROM
        store_inventory
    WHERE
        category = input_category
        AND status = 'Not Available';
END $$

DELIMITER ;
```

product_id	product_name	product_description	product_price	quantity	manufacturer	status
PH005	Vitamin C 500mg	Supports immune system	11.50	200	NutraLife	Not Available
PH008	Calcium 500mg	Bone health supplement	14.99	100	HealthPlus	Not Available
PH011	Melatonin 5mg	Sleep support supplement	12.99	150	NatureHealth	Not Available
PH014	Fish Oil 1000mg	Omega-3 supplement	21.50	50	PureLife	Not Available

Running the Procedure:

- The manager runs the procedure to get a list of unavailable items in the 'Pharma' category:

```
CALL GetNotAvailableProductsByCategory('Pharma');
```

To handle above situation, let's modify the stored procedure to include this condition

```
DELIMITER $$

CREATE PROCEDURE UpdateProductQuantity(IN productID VARCHAR(10), IN newQuantity INT)
BEGIN
    -- Update the quantity of the product
    UPDATE store_inventory
    SET quantity = newQuantity,
        status = CASE
                    WHEN newQuantity = 0 THEN 'Not Available'
                    ELSE 'Available'
                 END
    WHERE product_id = productID;
END $$

DELIMITER ;
```

Out of Stock:

Status changes from **available** to **Not available** when product is **Out of Stock**.

```
CALL UpdateProductQuantity('PH005', 0);
```

Restocking:

Status changes from **Not available** to **Available** again when product is **restocked**.

```
CALL UpdateProductQuantity('PH005', 150);
```

Cursor

- A cursor is like a control structure that allows you to **iterate through rows in a database table** or the result set of a query.
- It enables you to **process each row individually**, which can be useful for row-by-row operations like updates, deletes, or inserts.

- It's particularly handy when the processing requires complex logic that can't be easily handled by SQL alone.

Types of Cursors

Implicit Cursor:

Automatically created by the database when a single `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement is executed.

Explicit Cursor:

Manually defined by the programmer for more complex operations where you need to control the row-by-row processing.

The typical flow of using an explicit cursor involves several steps:

- **Declare**: A cursor is declared to define the SQL query that will retrieve the data.
- **Open**: The cursor is opened, which executes the query and stores the result set.
- **Fetch**: Rows are fetched one by one (or in bulk) from the cursor. Each fetch operation moves the cursor to the next row in the result set.
- **Process**: The fetched row is processed, which might include data manipulation or conditional logic.
- **Close**: The cursor is closed to release the memory and resources used.

Advantages of Cursors

- Allows **detailed manipulation** of query results.
- Ideal for scenarios where a **single SQL statement** is not enough.
- Useful for **batch processing** large datasets by breaking them into manageable chunks.

Disadvantages of Cursors

- **Performance Overhead**: They can be slower because they handle rows individually.
- **Resource Intensive**: Consume more memory and locks when open.
- Not suitable for large data sets if row-by-row processing isn't necessary.

```sql
CREATE PROCEDURE process_user_data()
BEGIN
    -- Step 1: Declare the variables to hold the data
    DECLARE done INT DEFAULT 0;
    DECLARE user_name VARCHAR(100);
    DECLARE user_age INT;
    DECLARE user_email VARCHAR(100);

    -- Step 2: Declare a cursor for the SELECT statement
    DECLARE user_cursor CURSOR FOR
    SELECT name, age, email FROM user_rec;

    -- Step 3: Declare a handler to handle the end of the cursor
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    -- Step 4: Open the cursor
    OPEN user_cursor;

    -- Step 5: Fetch rows from the cursor
    read_loop: LOOP
        FETCH user_cursor INTO user_name, user_age, user_email;

        -- Check if there are no more rows to fetch
        IF done = 1 THEN
            LEAVE read_loop;
        END IF;

        -- Example operation: Print user details
        SELECT CONCAT('User: ', user_name, ', Age: ', user_age, ', Email: ', user_email);
    END LOOP;

    -- Step 6: Close the cursor
    CLOSE user_cursor;
END $$
```

CONCAT('User: ', user_name, ', Age: ', user_age, ', Email: ', user_email)
User: Mona Agarwal, Age: 35, Email: mona.agarwal@example.com

Result 48 Result 49 Result 50 Result 51 Result 52 Result 53 Result 54 Result 55 Result 56 Result 57 Result 58 Result 59

Trigger :

A **trigger** in SQL is a set of instructions (a stored program) that automatically executes in response to a specific event on a table or view. It is commonly used to enforce business rules, maintain audit trails, or automatically update related data.

How a Trigger Works

A trigger is activated by events such as:

1. **INSERT**: When a new row is added to a table.
2. **UPDATE**: When data in an existing row is modified.
3. **DELETE**: When a row is removed from a table.

Triggers can run **before** or **after** these events, based on how they are defined:

- **BEFORE trigger**: Executes before the triggering event.
- **AFTER trigger**: Executes after the triggering event.


```
CREATE TRIGGER trigger_name

AFTER | BEFORE INSERT | UPDATE | DELETE

ON table_name

FOR EACH ROW

BEGIN

   -- SQL statements

END;
```

```sql
CREATE TRIGGER after_user_rec_insert
AFTER INSERT ON user_rec
FOR EACH ROW
BEGIN
    DECLARE generated_password VARCHAR(15);

    -- Generate a random password (simple logic for demonstration)
    SET generated_password = CONCAT('p@ss', LEFT(NEW.name, 4), FLOOR(RAND() * 1000));

    INSERT INTO credential (user_name, general_purp_id, rolebased_id, insurance_id, password, mobile_num)
    VALUES (
        NEW.name,
        NEW.general_purp_id,
        NEW.rolebased_id,
        NEW.insurance_id,
        generated_password,
        NEW.contact
    );
END $$

DELIMITER ;
```