
# Database Design and Management

**Surendra Panpaliya**

# Agenda

- Database Design Principles
- Normalization and denormalization
- Entity-Relationship (ER) modeling

# Agenda

01

**Creating and Managing Tables**

02

**Creating tables with SQL**

03

**Altering and dropping tables**

04

**Indexing for performance**

# Database Design Principles

Surendra Panpaliya

# Database Design Principles

Effective database design

Crucial for creating a database

Robust

Scalable

Efficient

# Understand the Requirements

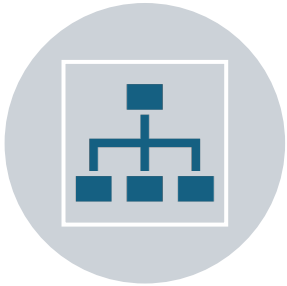
Data to be stored

Relationships between different data entities

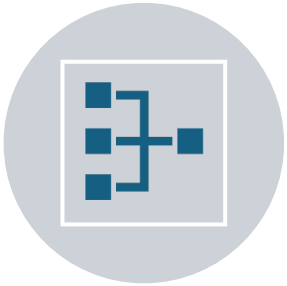
Queries that will be run

# Data Modeling

Use data modeling techniques



to represent the data structure.



Entity-Relationship (ER) modeling

is a popular method.

# Normalization

Organize data to minimize redundancy

dependency by dividing a database into

two or more tables

defining relationships between the tables.

# Denormalization

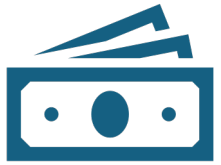
- Necessary for performance optimization
- Especially for read-heavy applications.
- Involves combining tables
- To reduce the number of joins

# Consistency and Integrity

Ensure data integrity

through

Constraints



Transactions

# Consistency and Integrity

Use Primary keys,

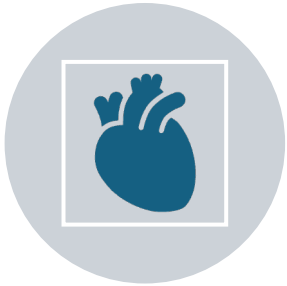
Foreign keys

Unique constraints

To enforce data integrity

# Scalability

Design the database



to scale horizontally (sharding) or

vertically (increasing resources)

based on anticipated growth.

# Security

Incorporate security measures

To protect sensitive data,

including Access controls,

Encryption, and Auditing

# Performance

OPTIMIZE DATABASE PERFORMANCE

WITH INDEXING

QUERY OPTIMIZATION

CACHING STRATEGIES

# Database Design Principles for Healthcare

Designing a robust and efficient database

for the healthcare domain requires

careful consideration of several principles

to ensure data integrity,

security, and

performance.

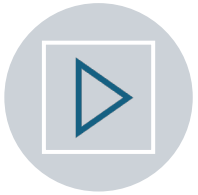
# Understand the Requirements

STAKEHOLDER ANALYSIS

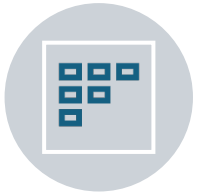
DATA REQUIREMENTS

REGULATORY COMPLIANCE

# Stakeholder Analysis



Identify and Understand



Needs of all stakeholders

doctors, nurses

administrative staff

IT personnel

# Data Requirements

Determine the types of data

to be stored.

patient records

appointment schedules

medical histories, prescriptions

billing information

# Regulatory Compliance

Ensure compliance with

Healthcare regulations.

HIPAA (Health Insurance Portability and Accountability Act)

for data privacy and security

# Data Modeling

Entities and Relationships

Normalization

Denormalization

# Entities and Relationships

Identify key entities

Patients, Doctors,

Appointments, MedicalRecords,

Prescriptions, Billing

Define their relationships.

# Normalization

Apply normalization rules

Up to the 3rd Normal Form.

To eliminate data redundancy

Ensure data integrity

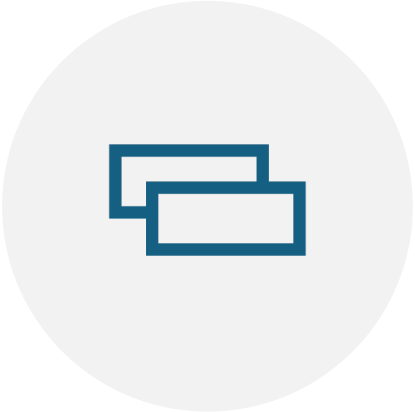
# Denormalization

Denormalize necessary

To improve performance

For read-heavy operations

# Schema Design



**TABLES AND COLUMNS**

**PRIMARY KEYS**

**FOREIGN KEYS**

# Tables and Columns

Create tables for each entity

with appropriate columns and data types

Use meaningful column names

Document their purpose

# Primary Keys

Define primary keys

for all tables

to uniquely identify

Records

# Foreign Keys

Use foreign keys

to establish relationships

between tables,

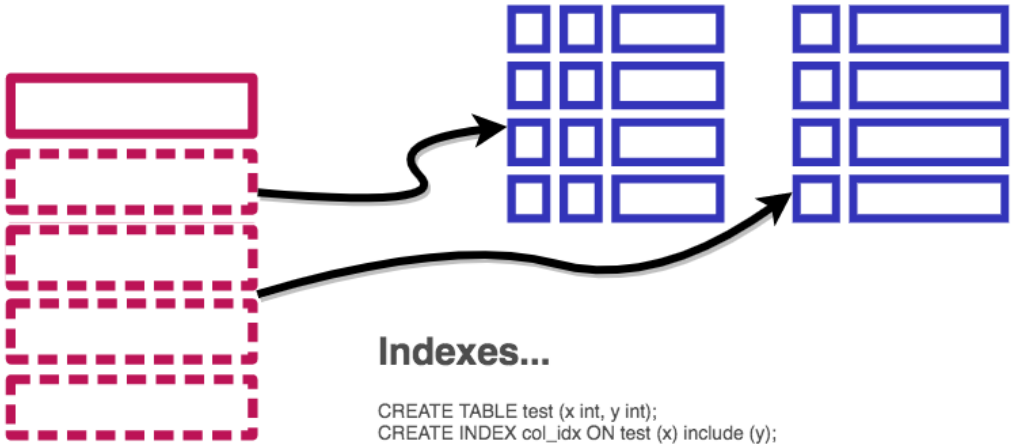
ensuring

referential integrity

# Constraints and Indexing

**Constraints**

**Indexes**



# Constraints

Apply appropriate constraints

UNIQUE, CHECK, NOT NULL

to enforce data validity

# Indexes

Create indexes on

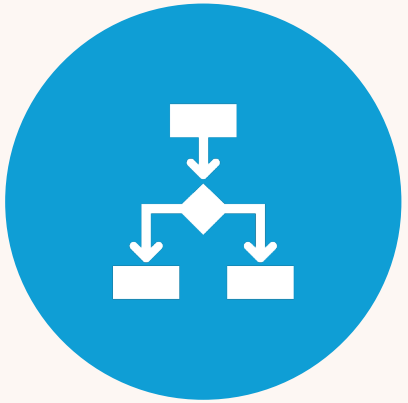
frequently queried columns

to enhance performance

# Indexes

USE COMPOSITE INDEXES

FOR QUERIES INVOLVING



MULTIPLE COLUMNS.

# Security and Access Control

Roles and Permissions

Encryption

Audit Logs

# Roles and Permissions

Define roles and

grant appropriate permissions

to ensure users have access

only to the data they need

# Encryption

Use encryption for sensitive data both at rest in transit.

# Audit Logs

Implement

audit logging

to track

data access

modifications.

# Data Integrity and Consistency

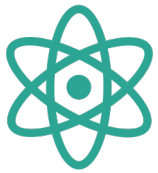
TRANSACTIONS

TRIGGERS AND STORED PROCEDURES

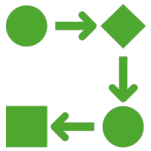
# Transactions

Use transactions

to ensure



atomicity of operations



for multi-step processes

# Triggers and Stored Procedures

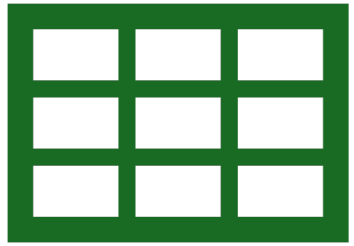
IMPLEMENT TRIGGERS

STORED PROCEDURES

FOR COMPLEX VALIDATION

BUSINESS LOGIC ENFORCEMENT

# Performance Optimization



**Query Optimization**

**Partitioning**

# Query Optimization

ANALYZE AND

OPTIMIZE QUERIES

FOR BETTER PERFORMANCE

USING EXPLAIN

ANALYZE.

# Partitioning

USE TABLE PARTITIONING

FOR LARGE DATASETS

TO IMPROVE

QUERY PERFORMANCE

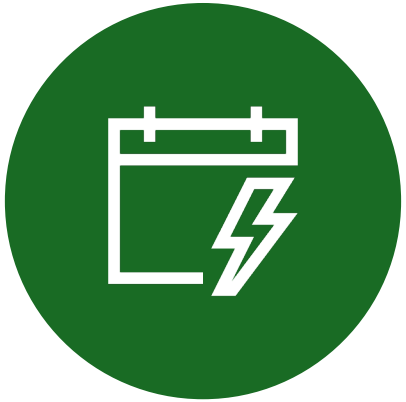
MAINTENANCE

# Scalability and Availability

- Replication
- Backup and Restore

# Replication

SET UP REPLICATION



FOR HIGH AVAILABILITY

DISASTER RECOVERY.

# Backup and Restore

IMPLEMENT

REGULAR BACKUP STRATEGIES

ENSURE THE ABILITY

TO RESTORE DATA QUICKLY

# Compliance and Auditing

REGULATORY COMPLIANCE

AUDIT TRAILS

# Regulatory Compliance

ENSURE

THE DATABASE DESIGN

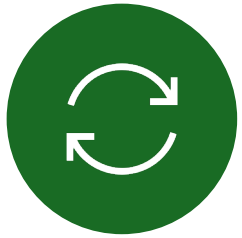
OPERATIONS COMPLY

WITH RELEVANT HEALTHCARE

REGULATIONS

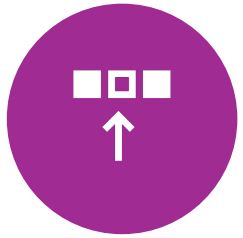
# Audit Trails

MAINTAIN AUDIT TRAILS



FOR CRITICAL DATA CHANGES

TO SUPPORT REGULATORY



COMPLIANCE

FORENSIC ANALYSIS

# Example PostgreSQL Schema for Healthcare Domain

Surendra Panpaliya

# Patients table

```
CREATE TABLE Patients (
    patient_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    date_of_birth DATE NOT NULL,
    gender VARCHAR(10),
    contact_number VARCHAR(15),
    email VARCHAR(100) UNIQUE,
    address TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

# Doctors table

```
CREATE TABLE Doctors (
    doctor_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    specialty VARCHAR(100),
    contact_number VARCHAR(15),
    email VARCHAR(100) UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

# Appointments table

```
CREATE TABLE Appointments (
    appointment_id SERIAL PRIMARY KEY,
    patient_id INT NOT NULL,
    doctor_id INT NOT NULL,
    appointment_date TIMESTAMP NOT NULL,
    status VARCHAR(20) CHECK (status IN ('Scheduled', 'Completed', 'Cancelled')),
    notes TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (patient_id) REFERENCES Patients(patient_id),
    FOREIGN KEY (doctor_id) REFERENCES Doctors(doctor_id)
);
```

# MedicalRecords table

```
CREATE TABLE MedicalRecords (
    record_id SERIAL PRIMARY KEY,
    patient_id INT NOT NULL,
    doctor_id INT,
    record_date TIMESTAMP NOT NULL,
    description TEXT,
    treatment TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (patient_id) REFERENCES Patients(patient_id),
    FOREIGN KEY (doctor_id) REFERENCES Doctors(doctor_id)
);
```

# Prescriptions table

```
CREATE TABLE Prescriptions (
    prescription_id SERIAL PRIMARY KEY,
    patient_id INT NOT NULL,
    doctor_id INT NOT NULL,
    prescription_date TIMESTAMP NOT NULL,
    medication TEXT NOT NULL,
    dosage TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (patient_id) REFERENCES Patients(patient_id),
    FOREIGN KEY (doctor_id) REFERENCES Doctors(doctor_id)
);
```

# Billing table

```
CREATE TABLE Billing (
    bill_id SERIAL PRIMARY KEY,
    patient_id INT NOT NULL,
    appointment_id INT,
    amount DECIMAL(10, 2) NOT NULL,
    payment_status VARCHAR(20) CHECK (payment_status IN ('Pending', 'Paid', 'Cancelled')),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (patient_id) REFERENCES Patients(patient_id),
    FOREIGN KEY (appointment_id) REFERENCES Appointments(appointment_id)
);
```

# PostgreSQL Schema for Healthcare Domain

This schema design example showcases how to structure a PostgreSQL database

for a healthcare application,

ensuring data integrity,

security, and performance

while following best practices

in database design.

# Normalization and Denormalization

Surendra Panpaliya

GKTCS Innovations

# Normalization

PROCESS OF ORGANIZING DATA

IN A DATABASE

TO REDUCE REDUNDANCY AND

IMPROVE DATA INTEGRITY

# Normalization

Involves dividing

large tables into smaller

related tables

using relationships

between them.

# Normalization Steps

**FIRST NORMAL FORM (1NF)**

**SECOND NORMAL FORM (2NF)**

**THIRD NORMAL FORM (3NF)**

# First Normal Form (1NF)

Ensure that each table

has a primary key.

Eliminate duplicate columns

from the same table.

# First Normal Form (1NF)

Create separate tables

for each group of related data and

identify each row with

a unique column

or set of columns

# Second Normal Form (2NF)

Ensure the table is in 1NF.

Remove subsets of data

that apply to multiple rows

place them in separate tables.

# Second Normal Form (2NF)

CREATE RELATIONSHIPS BETWEEN

THESE NEW TABLES AND

THEIR PREDECESSORS

USING FOREIGN KEYS.

# Third Normal Form (3NF)

Ensure the table is in 2NF.

Remove columns

that are not dependent

on the primary key

# Unnormalized Table

| PatientID | PatientName | DoctorID | DoctorName | Appointment Date | Medication | Dosage |
|---|---|---|---|---|---|---|
| 1 | John Doe | 10 | Dr. Smith | 2024-07-01 | Aspirin | 100mg |
| 1 | John Doe | 10 | Dr. Smith | 2024-07-15 | Ibuprofen | 200mg |
| 2 | Jane Doe | 20 | Dr. Brown | 2024-07-05 | Paracetamol | 500mg |

# Normalized Tables

## Patients Table

| PatientID | PatientName |
| --- | --- |
| 1 | John Doe |
| 2 | Jane Doe |

# Doctors Table

| DoctorID | DoctorName |
| --- | --- |
| 10 | Dr. Smith |
| 20 | Dr. Brown |

# Appointments Table

| AppointmentID | PatientID | DoctorID | AppointmentDate |
|---|---|---|---|
| 1 | 1 | 10 | 2024-07-01 |
| 2 | 1 | 10 | 2024-07-15 |
| 3 | 2 | 20 | 2024-07-05 |

# Medications Table

| MedicationID | AppointmentID | Medication | Dosage |
|---|---|---|---|
| 1 | 1 | Aspirin | 100mg |
| 2 | 2 | Ibuprofen | 200mg |
| 3 | 3 | Paracetamol | 500mg |

# Denormalization

Process of combining

normalized tables

to improve read performance

by reducing

the number of joins.

# Denormalization

PROCESS MAY INTRODUCE

REDUNDANCY

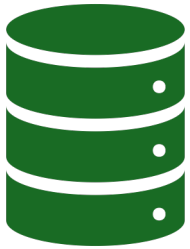
BUT CAN BE USEFUL

FOR OPTIMIZING

READ-HEAVY OPERATIONS.

# Denormalization Considerations

Query Performance



Data Redundancy


Use Cases

# Query Performance

Combine tables

to reduce the number of joins

required in queries.

Useful for read-heavy operations

where speed is crucial.

# Data Redundancy

Introduce redundancy

to improve performance,

but manage it carefully

to maintain data integrity

# Use Cases

- Reports and dashboards
- where denormalized data
- can be queried quickly.
- Applications with
- high read-to-write ratio.

# Normalized Patients Table

| PatientID | PatientName |
| --- | --- |
| 1 | John Doe |
| 2 | Jane Doe |

# Normalized Appointments Table

| AppointmentID | PatientID | DoctorID | AppointmentDate |
|---|---|---|---|
| 1 | 1 | 10 | 2024-07-01 |
| 2 | 1 | 10 | 2024-07-15 |
| 3 | 2 | 20 | 2024-07-05 |

# Denormalized Table

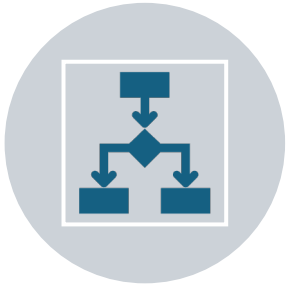
| PatientID | PatientName | AppointmentID | DoctorID | AppointmentDate | Medication | Dosage |
|---|---|---|---|---|---|---|
| 1 | John Doe | 1 | 10 | 2024-07-01 | Aspirin | 100mg |
| 1 | John Doe | 2 | 10 | 2024-07-15 | Ibuprofen | 200mg |
| 2 | Jane Doe | 3 | 20 | 2024-07-05 | Paracetamol | 500mg |

# Query Performance

Denormalized data

allows for quicker read access

as it eliminates



the need for multiple joins.

# Query Performance

Ideal for applications

requiring fast read operations,

such as healthcare dashboards

showing patient data,

appointments, and

medications in a single view.

# Conclusion and Summary

Normalization

denormalization

have their place

in database design.

# Conclusion and Summary

Normalization ensures

data integrity and

reduces redundancy,

which is crucial for

write-heavy operations and

maintaining consistency.

# Conclusion and Summary

Denormalization

used to optimize read performance

in read-heavy applications,

such as Reporting and

dashboard applications.

# Entity-Relationship (ER) Modeling

Surendra Panpaliya

# Entity-Relationship (ER) Model

Visually represents

Data

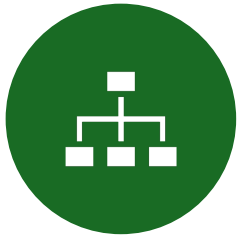
Relationships between

different entities

in a system.

# ER Model in Healthcare

ER MODEL HELPS IN



ORGANIZING

UNDERSTANDING

VARIOUS DATA COMPONENTS

INTERACTIONS.

# Key Entities and Relationships

Surendra Panpaliya

GKTCS Innovations

# Patient

| Attributes: |
| --- |
| PatientID (PK), |
| FirstName, LastName, |
| DateOfBirth, Gender, |
| ContactNumber, Email, |
| Address |

# Doctor

| Attributes: |
| --- |
| DoctorID (PK), |
| FirstName, LastName, |
| Specialty, |
| ContactNumber, |
| Email |

# Appointment

| Attributes: |
| --- |
| AppointmentID (PK), |
| PatientID (FK), |
| DoctorID (FK), |
| AppointmentDate, |
| Status, Notes |

# Appointment

| Relationships: |
| --- |
| Each appointment is |
| associated with |
| one patient and |
| one doctor. |

# MedicalRecord

| Attributes: |
| --- |
| RecordID (PK), PatientID (FK), |
| DoctorID (FK), RecordDate, |
| Description, Treatment |

# MedicalRecord

| |
|---|
| Relationships: |
| Each medical record is |
| associated with one patient and |
| optionally one doctor. |

# Prescription

| Attributes: |
| --- |
| PrescriptionID (PK), |
| PatientID (FK), |
| DoctorID (FK), |
| PrescriptionDate, |
| Medication, Dosage |

# Prescription

| Relationships: |
| --- |
| Each prescription is |
| associated with |
| one patient and |
| one doctor. |

# Billing

| Attributes: |
| --- |
| BillID (PK), |
| PatientID (FK), |
| AppointmentID (FK), |
| Amount, |
| PaymentStatus |

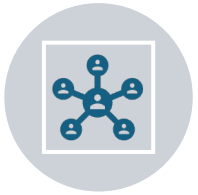
# Billing

| Relationships: |
| --- |
| Each bill is associated with |
| one patient and |
| one appointment. |

# User

- Attributes:
- UserID (PK),
- Username,
- PasswordHash,
- Role (Admin, Doctor, Patient)

# User



Relationships:

Each user can be

associated with

a patient,

doctor, or

administrative role.

# Insurance

Attributes:

InsuranceID (PK),

ProviderName,

PlanType,

PatientID (FK)

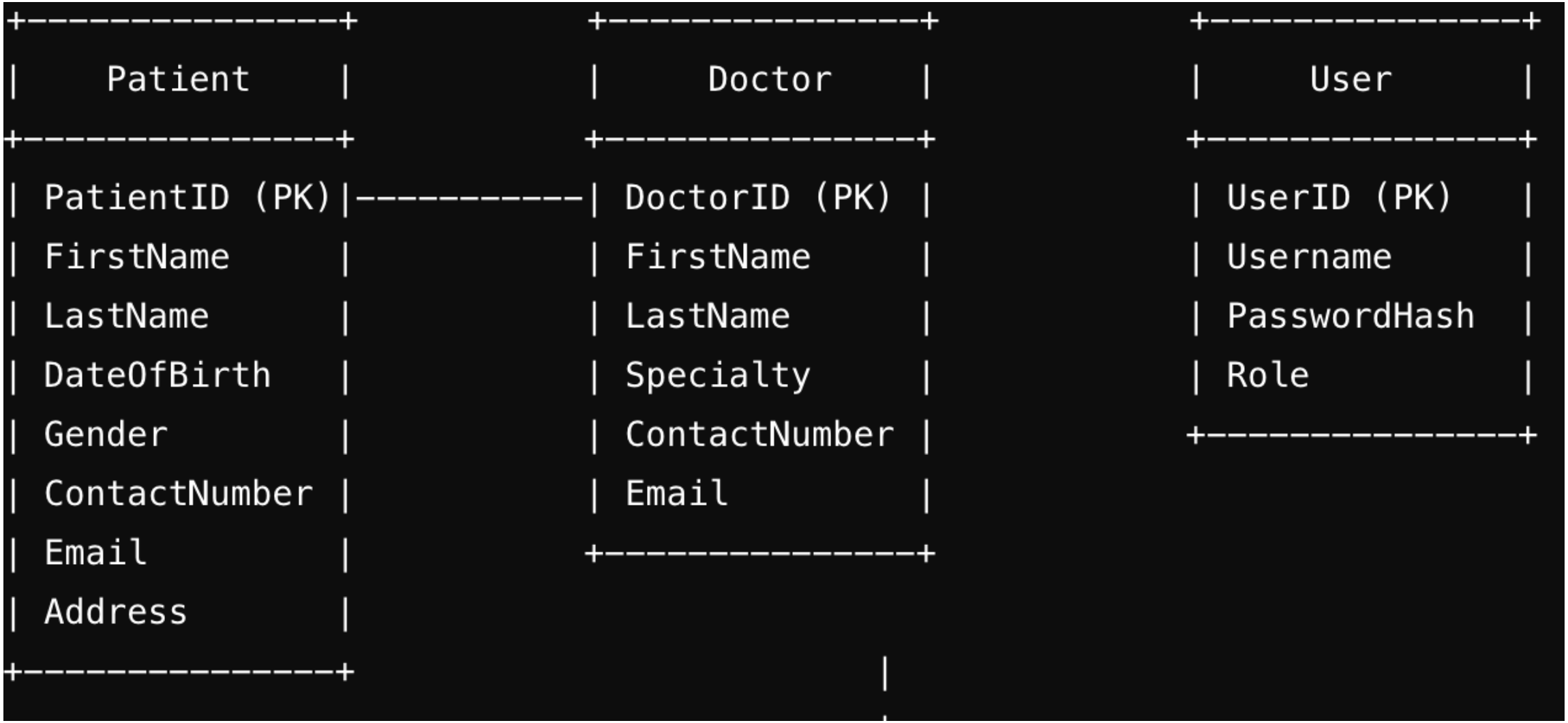
# Insurance

RELATIONSHIPS:

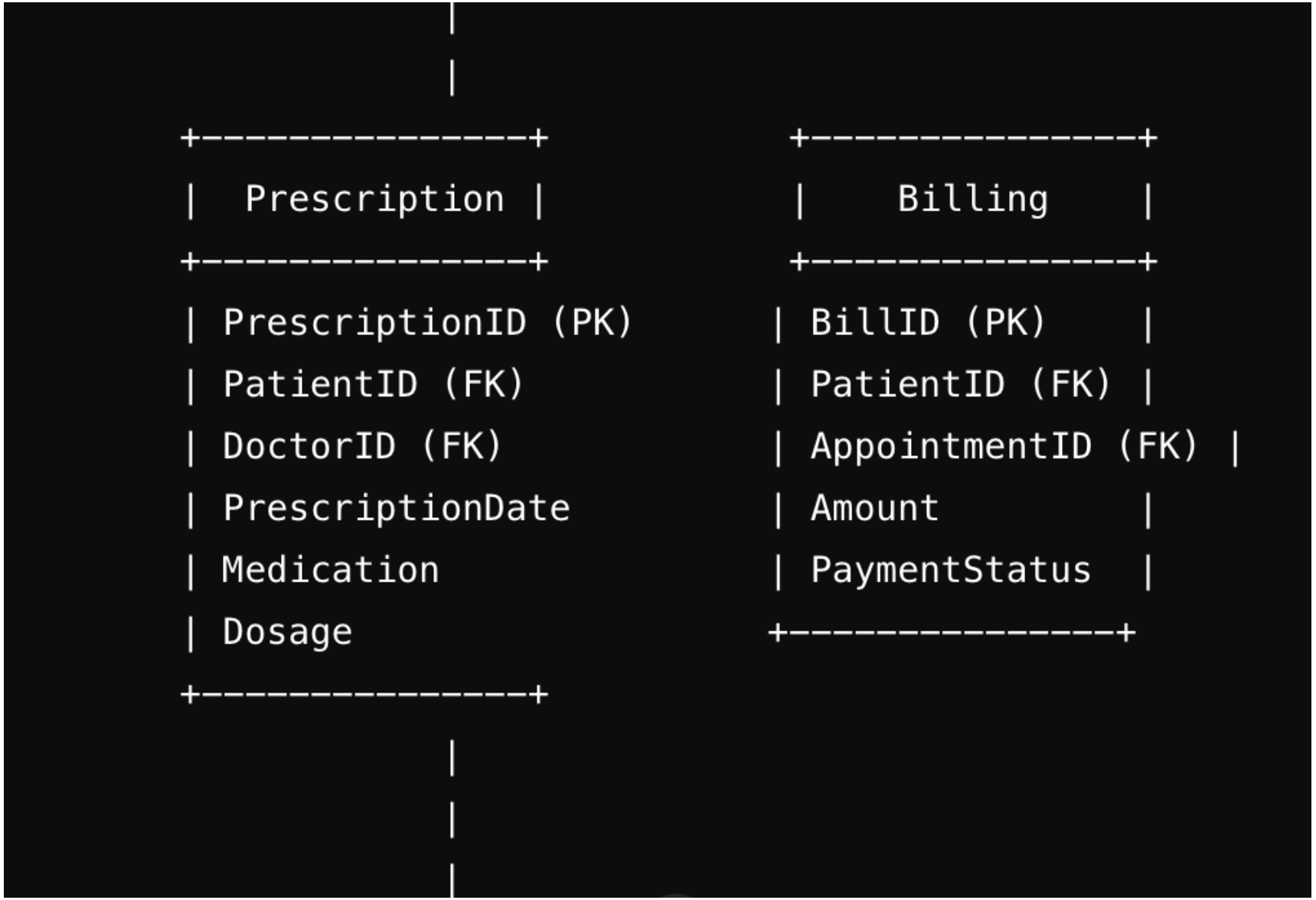
EACH INSURANCE PLAN IS

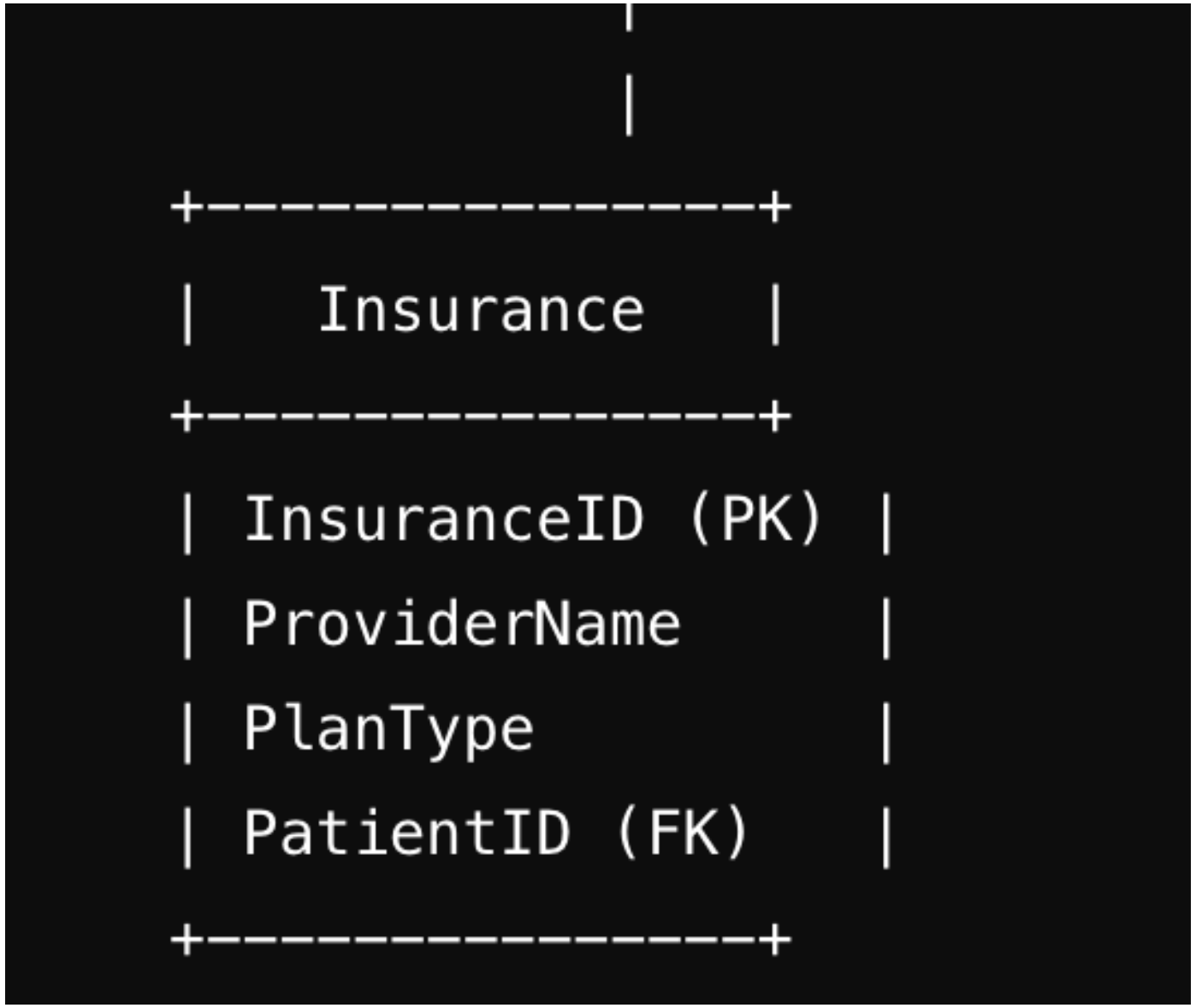
ASSOCIATED WITH

ONE PATIENT.


Patient
PatientID (PK)
FirstName
LastName
DateOfBirth
Gender
ContactNumber
Email
Address
Doctor
DoctorID (PK)
FirstName
LastName
Specialty
ContactNumber
Email
User
UserID (PK)
Username
PasswordHash
Role


Appointment
AppointmentID (PK)
PatientID (FK)
DoctorID (FK)
AppointmentDate
Status
Notes
MedicalRecord
RecordID (PK)
PatientID (FK)
DoctorID (FK)
RecordDate
Description
Treatment


Prescription
PrescriptionID (PK)
PatientID (FK)
DoctorID (FK)
PrescriptionDate
Medication
Dosage
Billing
BillID (PK)
PatientID (FK)
AppointmentID (FK)
Amount
PaymentStatus


Insurance
InsuranceID (PK)
ProviderName
PlanType
PatientID (FK)

# Managing Tables

Surendra Panpaliya

GKTCS Innovations

# Altering Tables

Allows to modify

the structure

of existing tables

Adding, Modifying

Dropping columns

# Altering Tables

Adding a New Column to a Table

```
ALTER TABLE Patients
ADD COLUMN middle_name VARCHAR(50);
```

Add a new column middle_name to the Patients table

# Modifying an Existing Column

Change the data type of the contact_number column in the Doctors table

```
ALTER TABLE Doctors
ALTER COLUMN contact_number TYPE VARCHAR(20);
```

# Set a default value for the status column in the Appointments table

```
ALTER TABLE Appointments
ALTER COLUMN status SET DEFAULT 'Scheduled';
```

# Dropping a Column from a Table

- **Remove the specialty column from the Doctors table:**
- ALTER TABLE Doctors
- DROP COLUMN specialty;

# Renaming a Column

**Rename the contact_number column to phone_number in the Patients table**

```
ALTER TABLE Patients
RENAME COLUMN contact_number TO phone_number;
```

# Adding a Foreign Key Constraint

- **Add a foreign key constraint to the MedicalRecords table to reference the Doctors table:**

- ALTER TABLE MedicalRecords
- ADD CONSTRAINT fk_doctor
- FOREIGN KEY (doctor_id) REFERENCES Doctors(doctor_id);

# Dropping a Foreign Key Constraint

**Drop the foreign key constraint from the MedicalRecords table**

```
ALTER TABLE MedicalRecords
DROP CONSTRAINT fk_doctor;
```

# Dropping Tables

- Removes Tables
- Entirely from
- Database

# Drop the Insurance table

```
DROP TABLE Insurance;
```

# Dropping a Table Only If It Exists

**Drop the Billing table only if it exists**

```
DROP TABLE IF EXISTS Billing;
```

# Dropping Multiple Tables

**Drop the Appointments and Prescriptions tables**

```
DROP TABLE Appointments, Prescriptions;
```

# Dropping a Table and Its Dependent Objects

**Drop the Users table and any objects that depend on it (e.g., foreign key constraints)**

```
DROP TABLE Users CASCADE;
```

# Example Workflow

**Scenario:**

**Adding a Column and**

**Then Dropping a Table**

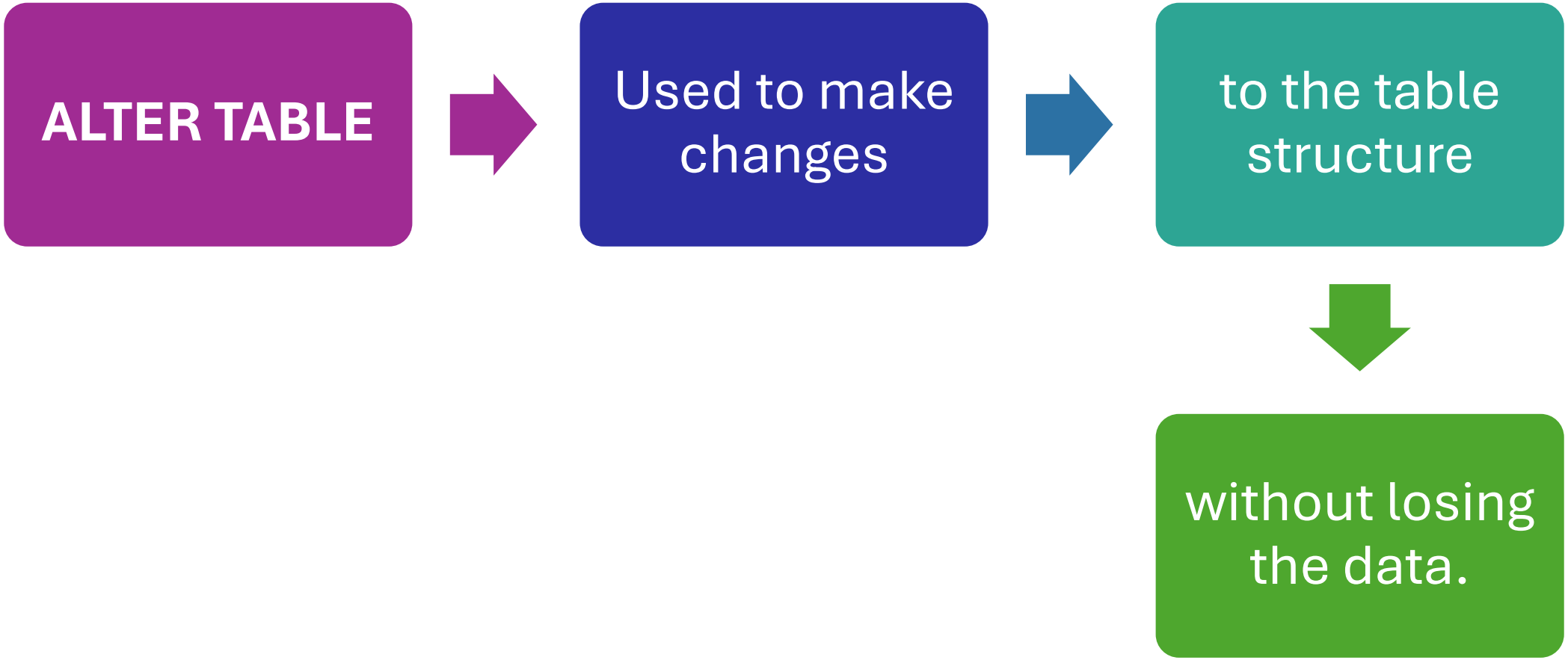
# Step1: Add a new column emergency_contact to the Patients table

```
ALTER TABLE Patients
ADD COLUMN emergency_contact VARCHAR(50);
```

# Step 2: Drop the MedicalRecords table

```
DROP TABLE MedicalRecords;
```

# Important Notes



# Important Notes

**DROP TABLE**

remove the table and

all its data from

the database permanently.

# Important Notes

USING **CASCADE**

WITH **DROP TABLE**

ENSURES THAT

ALL DEPENDENT OBJECTS

ARE ALSO DROPPED.

# Important Notes

**IF EXISTS** is useful

To avoid errors

When attempting

To drop tables

That may not exist

# Indexing for Performance

**Surendra Panpaliya**

**GKTCS Innovations**

# Indexing for Performance

Indexing is

a powerful technique

to improve

the performance of

database queries.

# Indexing for Performance

Proper indexing

can significantly

speed up

data retrieval operations.

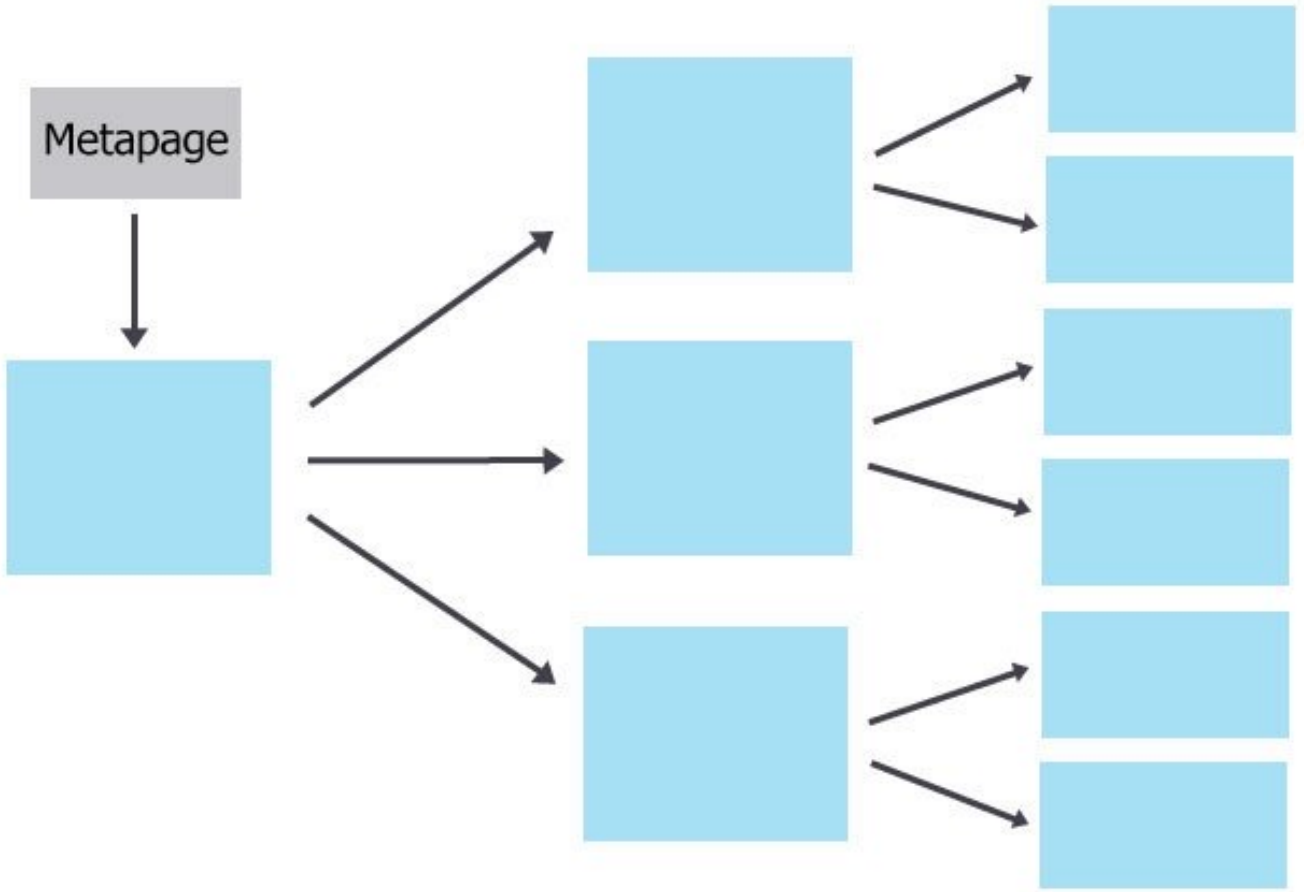
# Types of Indexes

- B-tree Index
- Hash Index
- GIN (Generalized Inverted Index
- GiST (Generalized Search Tree)
- BRIN (Block Range INdex)

# B-tree Index

- Default
- Most common type of index
- Ideal for equality
- Range queries

# PostgreSQL complete B-tree



# Creating a B-tree Index

**Index on the email column of the Patients table**

CREATE INDEX idx_patients_email ON Patients(email);

# Creating a Unique Index

**Ensure email addresses are unique in the Doctors table:**

```
CREATE UNIQUE INDEX idx_doctors_email_unique ON
Doctors(email);
```

# Creating a Composite Index

**Index on the first_name and last_name columns of the Patients table**

```
CREATE INDEX idx_patients_name ON Patients(first_name, last_name);
```

# Creating a Partial Index

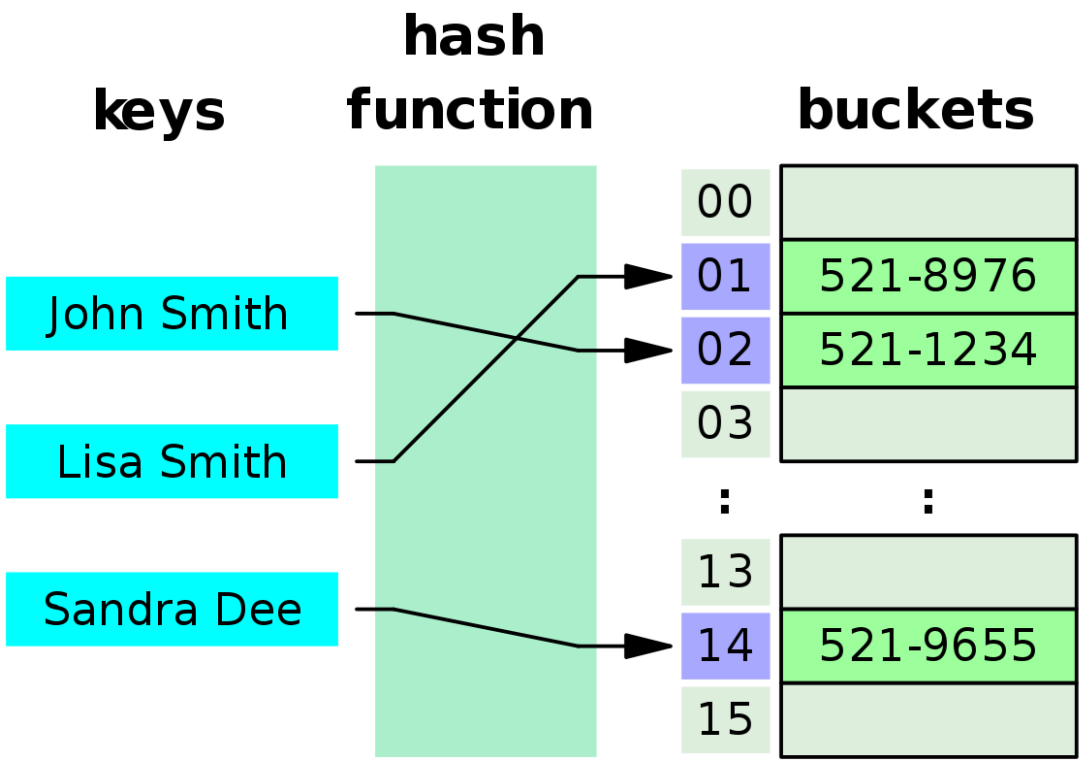
**Index on active appointments only in the Appointments table**

```
CREATE INDEX idx_active_appointments ON
Appointments(appointment_date)
```

```
WHERE status = 'Scheduled';
```

Useful for equality comparisons.

# Hash Index



# GIN (Generalized Inverted Index)

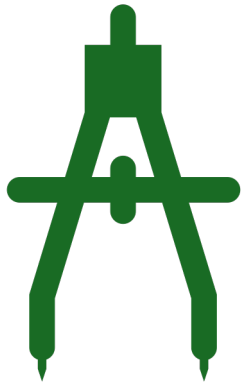
Ideal for array and full-text search operations.

# Creating a GIN Index

**Index for full-text search on the description column of the MedicalRecords table:**

```
CREATE INDEX idx_medicalrecords_description_gin ON MedicalRecords
USING gin(to_tsvector('english', description));
```

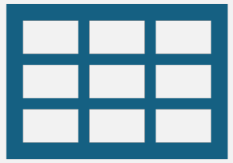
# GiST (Generalized Search Tree)



Useful for complex data types

like geometric data.

# BRIN (Block Range INdex)



EFFICIENT FOR VERY LARGE TABLES

WHERE DATA IS NATURALLY CLUSTERED.

# Creating a BRIN Index

**Index on the appointment_date column of the Appointments table for large datasets**

CREATE INDEX idx_appointments_date_brin ON Appointments

USING brin(appointment_date);

# Managing Indexes

Surendra Panpaliya

# Dropping an Index

**Drop the index on the email column of the Patients table**

```
DROP INDEX idx_patients_email;
```

# Reindexing a Table

**Rebuild all indexes on the Patients table**

```
REINDEX TABLE Patients;
```

# Viewing Indexes

**List all indexes in the current database**

postgres#\di

# Analyzing Index Usage

**Analyze the Appointments table**

**to update statistics for query planner**

ANALYZE Appointments;

# Best Practices for Indexing

Surendra Panpaliya

GKTCS Innovations

# Index Selective Columns

- Index columns that are frequently
- used in
- WHERE clauses,
- JOIN conditions
- ORDER BY clauses.

# Avoid Over-Indexing

Too many indexes

can slow down

INSERT, UPDATE, and DELETE operations

due to the overhead of

maintaining the indexes.

# Use Composite Indexes

For queries

that filter on multiple columns,

use composite indexes

to improve performance.

# Consider Index Types

Choose the appropriate index type

based on

the query patterns

data types.

# Consider Index Types

FOR EXAMPLE,

USE GIN INDEXES

FOR FULL-TEXT SEARCH.

# Monitor Index Usage

Regularly monitor

Analyze index usage

to identify and remove

unused or redundant indexes.

# Use Partial Indexes

Create partial indexes

to index only a subset of data

that meets specific criteria.

# Maintain Indexes

REGULARLY REINDEX TABLES AND

UPDATE STATISTICS

TO ENSURE OPTIMAL PERFORMANCE.

# Scenario

Optimizing a Query on the Appointments Table

# Identify the Query to Optimize

```
SELECT * FROM Appointments
WHERE doctor_id = 1 AND status = 'Scheduled'
ORDER BY appointment_date DESC;
```

# Create a Composite Index

```
CREATE INDEX idx_appointments_doctor_status_date ON
Appointments(doctor_id, status, appointment_date);
```

# Analyze the Table

```
ANALYZE Appointments;
```

# Check the Execution Plan

```
EXPLAIN ANALYZE SELECT * FROM Appointments
WHERE doctor_id = 1 AND status = 'Scheduled'
ORDER BY appointment_date DESC;
```

# Summary

Following indexing techniques

Best practices

Can significantly improve

Performance of queries

# Summary

Ensuring efficient

data retrieval

better overall database

performance

# Thank You


- Surendra Panpaliya
- Founder and CEO
- GKTCS Innovations
- https://www.gktcs.com