Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

INTRODUCTION TO DATABASES: DBMS VS. RDBMS

CHAPTER 1: UNDERSTANDING DATABASES AND THEIR IMPORTANCE

A database is a structured collection of data that allows efficient storage, retrieval, and management of information. In today's digital world, databases play a critical role in various industries, from finance and healthcare to e-commerce and social media. They enable organizations to store vast amounts of data systematically and retrieve it as needed for decision-making and operations.

The primary purpose of a database is to ensure data integrity, security, and accessibility. Without databases, businesses would struggle to manage customer records, transactions, employee data, and other essential information. A well-structured database provides fast access to data, supports multiple users, and maintains consistency across all records.

Key Features of a Database

- **Efficient Data Storage:** Stores large amounts of data systematically.
- **Data Retrieval:** Enables fast and accurate access to data.

- **Security and Access Control:** Protects data from unauthorized access.
- **Data Integrity:** Ensures consistency and accuracy of stored information.
- **Scalability:** Adapts to growing amounts of data over time.

Example:

Consider an online shopping website like Amazon. The company maintains a database that stores customer details, product information, order history, and payment records. When a user searches for a product, the database retrieves relevant details instantly, ensuring a smooth shopping experience.

Chapter 2: What is a Database Management System (DBMS)?

A **Database Management System (DBMS)** is a software application that interacts with databases to create, modify, manage, and retrieve data efficiently. DBMS serves as an intermediary between users and databases, ensuring that data is organized and accessible when needed.

DBMS provides tools for data storage, querying, and manipulation, eliminating the need for manually managing large sets of records. It also helps in enforcing security policies, maintaining data consistency, and preventing data redundancy.

Key Features of DBMS

- **Data Organization:** Structures data into tables or files for easy management.

- **Data Redundancy Reduction:** Eliminates duplicate records, ensuring efficiency.
- **Data Security:** Implements access controls and authentication mechanisms.
- **Backup and Recovery:** Ensures data safety in case of system failures.
- **Concurrency Control:** Allows multiple users to access data simultaneously.

Example:

A **university database system** uses a DBMS to manage student records, course registrations, and faculty details. Instead of manually updating spreadsheets, the university relies on DBMS for real-time data updates and retrieval.

Exercise:

1. Define a database and explain its key functions.
2. List at least three advantages of using a DBMS.
3. Provide an example of a DBMS used in a real-world scenario.

CHAPTER 3: INTRODUCTION TO RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)

A **Relational Database Management System (RDBMS)** is an advanced type of DBMS that stores data in structured tables with predefined relationships between them. Unlike traditional DBMS, RDBMS enforces rules that maintain data accuracy and consistency through **relations, constraints, and normalization techniques**.

RDBMS follows the **ACID (Atomicity, Consistency, Isolation, Durability) properties**, ensuring reliable transactions. It allows users to retrieve data using **Structured Query Language (SQL)**, a powerful language designed for managing relational databases.

Key Features of RDBMS

- **Structured Tables:** Data is stored in related tables, reducing complexity.
- **Data Integrity Constraints:** Ensures data validity using primary and foreign keys.
- **SQL Support:** Enables querying, updating, and managing data efficiently.
- **Multi-User Access:** Allows multiple users to access and modify data simultaneously.
- **Data Normalization:** Reduces redundancy and improves database efficiency.

Example:

A **hospital management system** uses RDBMS to store patient details, medical records, doctor schedules, and billing information. The relationships between these entities help in retrieving data efficiently—for example, linking a patient's ID to their medical history and billing information.

Exercise:

1. What are the major differences between DBMS and RDBMS?
2. Describe how SQL is used in an RDBMS.
3. Provide an example of a relational database in a business setting.

Chapter 4: DBMS vs. RDBMS – Key Differences

While both **DBMS and RDBMS** manage databases, there are significant differences in how they structure and process data. Understanding these differences helps in choosing the right system for various applications.

DBMS (Non-Relational) vs. RDBMS (Relational)

Feature	**DBMS (Database Management System)**	**RDBMS (Relational Database Management System)**
Data Storage	Stores data in files or unstructured formats	Stores data in structured tables
Data Relationships	No predefined relationships	Establishes relationships using keys
Query Language	Simple data handling, no SQL support	Uses SQL for querying and data manipulation
Data Redundancy	Higher chances of duplication	Redundancy minimized using normalization
Security & Multi-User Access	Less secure, limited multi-user support	High security with concurrent multi-user access
Examples	XML, File Systems, Hierarchical Databases	MySQL, Oracle, SQL Server, PostgreSQL

Example:

A **library system** using **DBMS** may store books and member details in separate files without linking them. In contrast, an **RDBMS-based library system** connects books, authors, and member information using relational keys, enabling efficient book tracking and borrowing history.

Exercise:

1. Compare and contrast DBMS and RDBMS in a tabular format.
2. Explain why businesses prefer RDBMS over DBMS in modern applications.
3. Provide an example of an industry where DBMS is still used instead of RDBMS.

CHAPTER 5: CASE STUDY – IMPLEMENTING RDBMS IN E-COMMERCE

Scenario:

ABC Retail is a growing e-commerce business that started with a simple **DBMS-based system** for managing product inventory and customer details. However, as the company expanded, challenges emerged:

- **Duplicate customer records** due to manual entry.
- **Inefficient order tracking** due to separate files for customers and orders.
- **Difficulty in analyzing sales data** without structured queries.

Solution:

The company migrated to **RDBMS (MySQL)**, establishing relationships between customers, orders, and products. This transition enabled:

- **Accurate order tracking** with relational keys linking orders to customers.
- **Better customer management** by reducing data duplication.
- **Advanced reporting capabilities** using SQL queries to generate sales insights.

Results:

- **Improved Efficiency:** Faster data retrieval and order processing.
- **Better Customer Experience:** Accurate tracking of customer preferences and purchases.
- **Scalability:** The ability to handle increasing data as the business grows.

Discussion Questions:

1. What challenges did ABC Retail face with DBMS?
2. How did RDBMS improve data management and operations?
3. Can you think of another industry that benefits from RDBMS?

CONCLUSION

Databases are the backbone of modern information systems, and the choice between **DBMS and RDBMS** depends on the complexity of data relationships and application requirements. While **DBMS** is suitable for simple, standalone applications, **RDBMS** is the preferred choice for businesses and enterprises due to its structured approach, SQL support, and efficiency.

By mastering **database management concepts**, professionals can build robust systems that enhance decision-making, streamline operations, and support large-scale applications. Whether managing an online store, hospital records, or banking transactions, **understanding databases is an essential skill in today's data-driven world**.

DATABASE DESIGN PRINCIPLES

CHAPTER 1: UNDERSTANDING DATABASE DESIGN

Database design is the foundation of any database management system, ensuring data is organized efficiently for storage, retrieval, and management. Proper database design enhances **data integrity, reduces redundancy, and improves query performance**. A well-structured database helps businesses and organizations manage large volumes of data effectively while maintaining security and reliability.

The process of designing a database involves understanding business requirements, defining relationships, and structuring data logically. **Poor database design** can lead to inefficiencies such as slow queries, inconsistent data, and storage issues. **A good database design** follows best practices like normalization, indexing, and security implementation to improve data organization and retrieval.

Key Principles of Database Design

- **Minimizing Data Redundancy:** Avoiding duplicate data across tables.
- **Ensuring Data Integrity:** Maintaining accuracy and consistency of stored data.
- **Optimizing Performance:** Structuring tables and queries for fast retrieval.
- **Enhancing Security:** Implementing user roles and permissions.

- **Scalability:** Designing for future expansion and increased data volume.

Example:

A university designing a student records database must **ensure each student's details, course enrollments, and grades** are stored in separate but linked tables. A poorly designed database may store redundant information in multiple tables, leading to inconsistencies when updating records.

CHAPTER 2: KEY STEPS IN DATABASE DESIGN

Step 1: Requirement Analysis

Before designing a database, it is essential to understand business needs and data flow. This involves gathering information from stakeholders to identify:

- **Types of data to be stored (e.g., customer details, transactions).**
- **User access levels and security requirements.**
- **Expected data volume and system scalability needs.**

Step 2: Conceptual Design - Entity-Relationship (ER) Model

The **ER model** is a high-level database design that visually represents entities (objects), attributes (properties), and relationships between entities. This phase helps in understanding how data will interact within the system.

Key Components of ER Model:

- **Entities:** Represent real-world objects (e.g., students, courses, orders).
- **Attributes:** Define entity properties (e.g., Student Name, Order Date).
- **Relationships:** Show associations between entities (e.g., A student enrolls in courses).

Example:

A hospital management system may have entities like **Patients, Doctors, Appointments, and Prescriptions**. The **relationship between Patients and Doctors** would be **many-to-many**, meaning a patient can visit multiple doctors, and a doctor can treat multiple patients.

Exercise:

1. Identify three real-world entities and their attributes.
2. Draw an ER diagram for a library management system.
3. Describe how relationships between entities affect database structure.

CHAPTER 3: NORMALIZATION – ELIMINATING DATA REDUNDANCY

What is Normalization?

Normalization is the process of **organizing data into structured tables** to reduce redundancy and improve integrity. It ensures that databases store only necessary information without unnecessary duplication.

Types of Normalization Forms:

- **First Normal Form (1NF):** Ensures that each column contains atomic (indivisible) values.
- **Second Normal Form (2NF):** Eliminates partial dependencies by ensuring all non-key attributes depend on the entire primary key.
- **Third Normal Form (3NF):** Removes transitive dependencies, ensuring no attribute is dependent on a non-key attribute.

Example:

Consider a **student database** that initially stores courses in a single column as:

Student ID	Student Name	Courses Taken
101	John Doe	Math, Science
102	Jane Smith	English, Math

This violates **1NF** because a single column stores multiple values. **Normalization** would separate courses into a different table, linking them using **foreign keys** to maintain integrity.

Exercise:

1. Convert a non-normalized table into 1NF, 2NF, and 3NF.
2. Identify a real-world scenario where normalization improves efficiency.
3. Explain why normalization is important in large-scale applications.

Chapter 4: Indexing and Query Optimization

What is Indexing?

Indexing improves **database performance** by speeding up search queries. An index is a **data structure that allows quick lookups**, similar to an index in a book.

Types of Indexing:

- **Primary Index:** Based on the primary key for faster lookups.
- **Clustered Index:** Sorts the data based on index order.
- **Non-Clustered Index:** Creates a separate structure for faster searches.

Example:

In an e-commerce database, indexing the **"Product Name"** column ensures that users searching for specific products receive results instantly instead of scanning millions of records sequentially.

Exercise:

1. Create an index for a "Customer Name" column in a database table.
2. Compare the performance of indexed vs. non-indexed queries.
3. Identify use cases where indexing is essential.

CHAPTER 5: SECURITY MEASURES IN DATABASE DESIGN

Ensuring Database Security

Database security involves implementing **measures to prevent unauthorized access, data breaches, and corruption**. With increasing cyber threats, organizations must secure sensitive data against internal and external attacks.

Key Security Measures:

- **User Authentication & Access Control:** Assigning roles and permissions.
- **Encryption:** Protecting sensitive data through encryption techniques.
- **Regular Backups:** Ensuring data recovery in case of failures.
- **SQL Injection Prevention:** Protecting against malicious queries.

Example:

A **banking database** stores customer transactions, account details, and passwords. Using encryption and multi-factor authentication (MFA) ensures only authorized users access this sensitive data.

Exercise:

1. List three security measures used in modern databases.
2. Explain the impact of SQL injection on an insecure database.
3. Discuss how role-based access control enhances database security.

CHAPTER 6: CASE STUDY – DESIGNING A DATABASE FOR AN ONLINE RETAIL STORE

Scenario:

ABC Retail wants to create a **structured database** to manage its growing e-commerce business. Previously, they stored customer orders and product details in spreadsheets, leading to **slow performance, duplicate data, and security risks**.

Challenges Faced:

1. **Data Redundancy:** Customer details were stored multiple times.
2. **Slow Search Queries:** Finding specific orders took too long.
3. **Security Issues:** Lack of user authentication led to unauthorized data access.

Solution:

ABC Retail implemented an **RDBMS using MySQL** with the following improvements:

1. **Customer, Order, and Product tables** were created to store structured data.
2. **Indexes were added** to product and order tables for faster search.
3. **User roles** were assigned to restrict data access.

Results:

- **50% faster query performance.**
- **Elimination of duplicate customer records.**
- **Improved security through role-based authentication.**

Discussion Questions:

1. What were the primary issues in ABC Retail's previous database system?
2. How did normalization and indexing improve performance?
3. How can security measures be further enhanced in online retail databases?

CONCLUSION

Designing an efficient database requires careful planning, normalization, indexing, and security implementation. **A well-designed database ensures data accuracy, minimizes redundancy, and enhances performance,** making it essential for modern businesses. By understanding **database design principles**, professionals can create scalable, secure, and high-performance database systems.

SQL Basics: Data Types, Constraints, and Keys

Chapter 1: Introduction to SQL and Its Importance

Structured Query Language (SQL) is a powerful language used to manage and manipulate relational databases. It allows users to create, retrieve, update, and delete data efficiently. SQL is the backbone of modern database management, enabling businesses to store and analyze large datasets systematically.

SQL is widely used in various applications, from small-scale websites to enterprise-level business management systems. Its importance lies in its ability to handle structured data, enforce rules, and maintain consistency across databases. The two main types of SQL operations are:

- **Data Definition Language (DDL):** Includes commands like CREATE, ALTER, and DROP to define the database structure.
- **Data Manipulation Language (DML):** Includes commands like SELECT, INSERT, UPDATE, and DELETE to manipulate data.

Key Benefits of SQL:

- **Data Integrity:** Ensures accuracy and consistency of data.
- **Efficiency:** Optimized queries retrieve data quickly.
- **Scalability:** Supports databases of all sizes, from small applications to enterprise systems.

- **Standardized Language:** Works with most relational database management systems (RDBMS) like MySQL, PostgreSQL, and Oracle.

Example:

A bank uses SQL to store customer details, transaction history, and loan records. Queries like SELECT * FROM Customers WHERE Balance > 10000; allow bank employees to fetch relevant data efficiently.

CHAPTER 2: SQL DATA TYPES

What are SQL Data Types?

SQL data types define the kind of values that can be stored in a table's columns. Choosing the correct data type ensures efficient storage, improves performance, and maintains data integrity.

Common SQL Data Types:

1. **Numeric Data Types:** Used to store numbers, including integers and decimals.
   - INT (Integer): Whole numbers without decimal places.
   - DECIMAL(p, s) or NUMERIC(p, s): Fixed precision decimal values.
   - FLOAT and DOUBLE: Floating-point numbers with varying precision.
2. **String Data Types:** Used to store text-based values.
   - CHAR(n): Fixed-length string with n characters.
   - VARCHAR(n): Variable-length string up to n characters.

- TEXT: Used for large text fields like descriptions and comments.

3. **Date and Time Data Types:** Used for storing dates, times, and timestamps.
   - DATE: Stores a date in YYYY-MM-DD format.
   - TIME: Stores only time values in HH:MM:SS format.
   - DATETIME: Stores both date and time values.
4. **Boolean Data Type:** Stores TRUE or FALSE values. Some databases represent it as TINYINT(1), where 1 = TRUE and 0 = FALSE.

Example:

A student database may use the following data types:

```
CREATE TABLE Students (

  StudentID INT PRIMARY KEY,

  Name VARCHAR(100),

  BirthDate DATE,

  GPA DECIMAL(3,2),

  Enrolled BOOLEAN

);
```

Exercise:

1. Define three different data types used in SQL and give an example of each.

2. Create a table named Orders with columns for OrderID, CustomerName, OrderDate, and TotalAmount.
3. Explain why choosing the correct data type is important for database performance.

CHAPTER 3: SQL CONSTRAINTS

What are Constraints?

SQL constraints enforce rules on data in a table to maintain **accuracy, integrity, and reliability**. They prevent invalid data entry and ensure consistency.

Types of SQL Constraints:

1. **NOT NULL:** Ensures that a column cannot store NULL values.
2. **UNIQUE:** Ensures all values in a column are distinct.
3. **PRIMARY KEY:** Uniquely identifies each record in a table.
4. **FOREIGN KEY:** Maintains referential integrity by linking two tables.
5. **CHECK:** Enforces specific conditions on data values.
6. **DEFAULT:** Assigns a default value if no data is provided.

Example:

In a Products table, constraints ensure valid data storage:

```
CREATE TABLE Products (

   ProductID INT PRIMARY KEY,

   Name VARCHAR(100) NOT NULL,
   Price DECIMAL(10,2) CHECK (Price > 0),
   Stock INT DEFAULT 10
);
```

Here, NOT NULL ensures every product has a name, CHECK ensures the price is positive, and DEFAULT assigns a stock value of 10 if none is provided.

Exercise:

1. Create a Customers table with constraints on CustomerID, Name, and Email.
2. Explain the difference between UNIQUE and PRIMARY KEY constraints.
3. Why is the FOREIGN KEY constraint important in relational databases?

Chapter 4: SQL Keys – Primary Key, Foreign Key, and More

What are SQL Keys?

Keys in SQL are essential for identifying and managing relationships between tables. They help ensure data integrity and establish connections between related records.

Types of SQL Keys:

1. **Primary Key:** A column (or combination of columns) that uniquely identifies each row in a table.
2. **Foreign Key:** A column that references a primary key in another table, ensuring referential integrity.

3. **Composite Key:** A key consisting of multiple columns used when no single column can uniquely identify a record.
4. **Candidate Key:** A column eligible to be a primary key but not necessarily chosen.
5. **Surrogate Key:** A system-generated unique identifier, often an auto-incrementing number.

Example:

A Students table and an Enrollments table are linked using a **foreign key**:

```
CREATE TABLE Students (

  StudentID INT PRIMARY KEY,

  Name VARCHAR(100)

);


CREATE TABLE Enrollments (

  EnrollmentID INT PRIMARY KEY,

  StudentID INT,

  CourseName VARCHAR(100),

  FOREIGN KEY (StudentID) REFERENCES Students(StudentID)

);
```

Here, StudentID in Enrollments is a **foreign key** that references StudentID in Students, ensuring each enrollment record is associated with a valid student.

Exercise:

1. Create an Employees table with EmployeeID as the primary key.
2. Create a second table, Departments, where DepartmentID is the primary key and is referenced in Employees as a foreign key.
3. Explain why **foreign keys** help maintain relational integrity.

Chapter 5: Case Study – Implementing SQL Keys and Constraints in an E-commerce Database

Scenario:

XYZ E-commerce wants to design a relational database to manage customers, orders, and products.

Challenges Faced:

1. **Duplicate customer records** due to lack of unique identifiers.
2. **Incorrect order tracking** due to missing relationships between tables.
3. **Data inconsistencies** due to the absence of constraints.

Solution:

XYZ implemented an **RDBMS using MySQL** with structured keys and constraints:

- **Primary keys** for Customers, Orders, and Products tables.
- **Foreign keys** linking orders to customers and products.

- **Constraints** to ensure accurate data entry.

Results:

- **Eliminated data duplication.**
- **Improved order tracking and efficiency.**
- **Maintained data consistency across the system.**

Discussion Questions:

1. Why were **primary and foreign keys** essential in XYZ's database?
2. How did constraints improve data integrity?
3. What additional security measures can be implemented?

CONCLUSION

SQL is a fundamental tool for managing relational databases, ensuring **efficient data storage, consistency, and accuracy**. Understanding **data types, constraints, and keys** is crucial for designing reliable database systems. By applying these concepts, businesses can maintain structured and high-performing databases.

CREATING AND MANAGING TABLES IN SQL

CHAPTER 1: INTRODUCTION TO TABLES IN SQL

Tables are the fundamental structures in SQL databases used to store, organize, and manage data. A table consists of **rows (records) and columns (fields)**, where each column has a specific data type defining the kind of values it can hold. Tables enable efficient data storage and retrieval, ensuring databases function effectively.

When designing a table, it is crucial to define appropriate **data types, constraints, and relationships** to maintain **data integrity and consistency**. Poorly designed tables can lead to **data redundancy, slow query performance, and difficulties in scaling the database**. The structure of a table should be well-planned to accommodate future data expansion while ensuring optimal performance.

Key Features of Tables in SQL

- **Structured Data Storage:** Data is stored in a tabular format with rows and columns.
- **Defined Data Types:** Each column has a specific data type such as VARCHAR, INT, DATE, etc.
- **Constraints for Data Integrity:** Rules like PRIMARY KEY, NOT NULL, UNIQUE, and FOREIGN KEY help maintain accuracy.
- **Relationships Between Tables:** Using keys (Primary and Foreign), multiple tables can be linked.
- **Scalability:** Tables can store and manage vast amounts of data efficiently.

Example:

A school database stores **student information** in a table named Students. Each student's data, including **ID, Name, Age, and Grade**, is stored in a structured format for quick retrieval.

```
CREATE TABLE Students (
  StudentID INT PRIMARY KEY,
  Name VARCHAR(100) NOT NULL,
  Age INT CHECK (Age > 3),
  Grade VARCHAR(10)
);
```

This example defines a Students table where:

- StudentID uniquely identifies each student (PRIMARY KEY).
- Name cannot be left empty (NOT NULL).
- Age must be greater than 3 (CHECK).

CHAPTER 2: CREATING TABLES IN SQL

SQL Command for Creating a Table

To create a table in SQL, the **CREATE TABLE** statement is used. It defines the table name, column names, data types, and any constraints required for ensuring data integrity.

Syntax for Creating a Table:

```
CREATE TABLE TableName (
  Column1 DataType Constraints,

  Column2 DataType Constraints,

  Column3 DataType Constraints

);
```

Example:

A company wants to create a table named Employees to store employee details:

```
CREATE TABLE Employees (

  EmployeeID INT PRIMARY KEY,

  FirstName VARCHAR(50) NOT NULL,

  LastName VARCHAR(50) NOT NULL,

  Email VARCHAR(100) UNIQUE,

  HireDate DATE DEFAULT CURRENT_DATE,

  Salary DECIMAL(10,2) CHECK (Salary > 0)

);
```

Here:

- EmployeeID is the **Primary Key**, ensuring uniqueness.
- FirstName and LastName cannot be NULL.
- Email must be unique, preventing duplicate entries.
- HireDate has a **default value** of the current date.
- Salary must be a positive number (CHECK).

Exercise:

1. Create a table named Products with columns: ProductID, ProductName, Category, Price, and StockQuantity.
2. Define appropriate **data types** and **constraints** for each column.
3. Why is it important to use constraints when creating a table?

CHAPTER 3: MODIFYING AND MANAGING TABLES

Altering Tables Using ALTER TABLE

As databases grow, requirements may change, necessitating **modifications to existing tables**. SQL provides the **ALTER TABLE** statement to:

- Add new columns
- Modify existing columns
- Remove columns
- Add constraints

Example – Adding a Column:

A company wants to add a PhoneNumber column to the Employees table:

```
ALTER TABLE Employees

ADD PhoneNumber VARCHAR(15);
```

Example – Modifying a Column:

If the company decides that the PhoneNumber column should not allow NULL values:

```
ALTER TABLE Employees

MODIFY PhoneNumber VARCHAR(15) NOT NULL;
```

Example – Dropping a Column:

If PhoneNumber is no longer needed, it can be removed:

```
ALTER TABLE Employees

DROP COLUMN PhoneNumber;
```

Exercise:

1. Modify the Products table to add a new column SupplierName (VARCHAR(100)).
2. Change the Price column to ensure it cannot have negative values.
3. Remove the StockQuantity column from the Products table.

Chapter 4: Deleting Tables and Managing Data

Deleting a Table Using DROP TABLE

If a table is no longer needed, it can be permanently deleted using DROP TABLE. However, this removes all data stored in the table, so caution must be taken.

Example:

To delete the Employees table permanently:

```
DROP TABLE Employees;
```

Deleting Data Using TRUNCATE TABLE

Unlike DROP TABLE, the TRUNCATE TABLE command removes all records from a table but **keeps the structure intact**.

```
TRUNCATE TABLE Employees;
```

Exercise:

1. Create a temporary table TestTable, insert some data, and delete it using DROP TABLE.
2. Explain the difference between DROP TABLE and TRUNCATE TABLE.
3. Why is it important to back up data before deleting a table?

Chapter 5: Case Study – Implementing Tables in an Online Retail Database

Scenario:

XYZ Retail, an online shopping platform, needs to design its database for managing **customers, orders, and products** efficiently. Previously, they relied on spreadsheets, which led to **data redundancy, slow searches, and errors in customer records**.

Challenges Faced:

1. **Duplicate customer entries** due to lack of unique identifiers.
2. **Difficulty tracking orders** across different locations.
3. **Slow performance** when retrieving product information.

Solution:

XYZ Retail created structured tables using SQL:

1. Customers Table

```
CREATE TABLE Customers (

   CustomerID INT PRIMARY KEY,

   Name VARCHAR(100) NOT NULL,

   Email VARCHAR(100) UNIQUE,

   Phone VARCHAR(15)

);
```

2. Orders Table

```
CREATE TABLE Orders (

   OrderID INT PRIMARY KEY,

   CustomerID INT,

   OrderDate DATE DEFAULT CURRENT_DATE,

   Amount DECIMAL(10,2),

   FOREIGN KEY (CustomerID) REFERENCES
Customers(CustomerID)

);
```

3. Products Table

```
CREATE TABLE Products (

   ProductID INT PRIMARY KEY,

   ProductName VARCHAR(100) NOT NULL,
   Price DECIMAL(10,2) CHECK (Price > 0),
   Stock INT CHECK (Stock >= 0)
);
```

Results:

- **Eliminated duplicate records** by using PRIMARY KEY constraints.
- **Improved order tracking** with a relational structure using FOREIGN KEY.
- **Faster searches** and efficient product management.

Discussion Questions:

1. How did creating tables help XYZ Retail manage its business better?
2. Why was the FOREIGN KEY in the Orders table important?
3. How can indexing further improve database performance?

CONCLUSION

Creating and managing tables in SQL is fundamental to database operations. **Well-structured tables** ensure efficient data storage, integrity, and retrieval. By using **data types, constraints, and relationships**, businesses can build scalable and high-performance databases. Properly managing tables with ALTER, DROP, and TRUNCATE commands ensures flexibility as requirements evolve.

Understanding table management is essential for **database administrators, software developers, and data analysts**.

Mastering these concepts allows professionals to build **robust, secure, and efficient** database systems.

INNER, LEFT, RIGHT, AND FULL JOINS IN SQL

CHAPTER 1: INTRODUCTION TO SQL JOINS

In relational databases, **joins** are used to combine data from multiple tables based on a related column. **SQL Joins** help retrieve meaningful data by establishing relationships between different tables. They are especially useful when working with **normalized databases**, where data is distributed across multiple tables to avoid redundancy and ensure efficient storage.

Joins enable database users to analyze data by bringing together related records from multiple tables. Without joins, retrieving data from different tables would require multiple queries, making the process inefficient. The most commonly used SQL joins are:

- **INNER JOIN:** Retrieves only matching records between two tables.
- **LEFT JOIN:** Retrieves all records from the left table and matching records from the right table.
- **RIGHT JOIN:** Retrieves all records from the right table and matching records from the left table.
- **FULL JOIN:** Retrieves all records from both tables, including unmatched records.

Using joins effectively enhances **query efficiency, reduces redundancy, and improves data organization**.

Example:

A company maintains two tables:

1. Employees (EmployeeID, Name, DepartmentID)
2. Departments (DepartmentID, DepartmentName)

Using a join, we can fetch **employee names along with their department names**, providing meaningful insights.

```
SELECT Employees.Name, Departments.DepartmentName

FROM Employees

INNER JOIN Departments

ON Employees.DepartmentID = Departments.DepartmentID;
```

This ensures that only employees **with valid department entries** are retrieved.

Chapter 2: INNER JOIN – Retrieving Matching Records

What is INNER JOIN?

An **INNER JOIN** retrieves records that have matching values in both tables. It discards unmatched records, ensuring that only **logically connected data** is returned.

Syntax for INNER JOIN:

```
SELECT table1.column1, table2.column2

FROM table1

INNER JOIN table2

ON table1.common_column = table2.common_column;
```

Example:

Consider two tables in an e-commerce database:

- Orders (OrderID, CustomerID, OrderDate)
- Customers (CustomerID, Name, Email)

To fetch customer names along with their order details:

```
SELECT Orders.OrderID, Customers.Name, Orders.OrderDate

FROM Orders

INNER JOIN Customers

ON Orders.CustomerID = Customers.CustomerID;
```

This ensures that **only orders placed by valid customers** are retrieved. If a customer exists but has no orders, they won't be included.

Exercise:

1. Write an INNER JOIN query to fetch students' names along with the courses they have enrolled in.
2. Modify the query to fetch employee names along with their assigned projects.
3. Explain why an INNER JOIN might exclude certain records.

Chapter 3: LEFT JOIN – Retrieving All Records from the Left Table

What is LEFT JOIN?

A **LEFT JOIN (LEFT OUTER JOIN)** retrieves all records from the **left table**, along with matching records from the **right table**. If there is no match, NULL values are returned for columns from the right table.

Syntax for LEFT JOIN:

SELECT table1.column1, table2.column2

FROM table1

LEFT JOIN table2

ON table1.common_column = table2.common_column;

Example:

Consider the Employees and Departments tables:

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

LEFT JOIN Departments

ON Employees.DepartmentID = Departments.DepartmentID;

This retrieves **all employees**, including those **without assigned departments** (NULL values).

Real-World Application:

An **online store** wants to list all **customers** and their **purchases**. However, some customers may not have made any purchases. A LEFT JOIN helps in retrieving all customers while showing NULL for those without purchases.

SELECT Customers.Name, Orders.OrderID

FROM Customers

LEFT JOIN Orders

ON Customers.CustomerID = Orders.CustomerID;

Exercise:

1. Write a LEFT JOIN query to retrieve all students and the subjects they are enrolled in, including students who have not enrolled in any subjects.
2. Modify a previous INNER JOIN query to use LEFT JOIN and compare results.
3. Why does LEFT JOIN return more records than INNER JOIN?

CHAPTER 4: RIGHT JOIN – RETRIEVING ALL RECORDS FROM THE RIGHT TABLE

What is RIGHT JOIN?

A **RIGHT JOIN (RIGHT OUTER JOIN)** retrieves all records from the **right table**, along with matching records from the **left table**. If no match exists, NULL values appear for columns from the left table.

Syntax for RIGHT JOIN:

SELECT table1.column1, table2.column2

FROM table1

RIGHT JOIN table2

ON table1.common_column = table2.common_column;

Example:

In an educational system, the school maintains:

- Teachers (TeacherID, Name, SubjectID)
- Subjects (SubjectID, SubjectName)

To get all subjects, even those without assigned teachers:

```
SELECT Teachers.Name, Subjects.SubjectName

FROM Teachers

RIGHT JOIN Subjects

ON Teachers.SubjectID = Subjects.SubjectID;
```

If a subject has **no assigned teacher**, the Teachers.Name field will be **NULL**.

Real-World Application:

An **insurance company** keeps track of **policies** and **clients**. Some policies may not have been assigned to a client. Using a RIGHT JOIN, we can list all policies, ensuring that even unassigned ones appear.

```
SELECT Clients.Name, Policies.PolicyName

FROM Clients

RIGHT JOIN Policies

ON Clients.ClientID = Policies.ClientID;
```

Exercise:

1. Write a RIGHT JOIN query to retrieve all departments, including those without employees.
2. Explain the difference between LEFT JOIN and RIGHT JOIN.
3. In what scenarios is RIGHT JOIN more useful than INNER JOIN?

CHAPTER 5: FULL JOIN – RETRIEVING ALL RECORDS FROM BOTH TABLES

What is FULL JOIN?

A **FULL JOIN (FULL OUTER JOIN)** retrieves all records from **both tables**, returning NULL for missing matches on either side. It is useful when **you want to see all data, regardless of whether there are matches**.

Syntax for FULL JOIN:

SELECT table1.column1, table2.column2

FROM table1

FULL JOIN table2

ON table1.common_column = table2.common_column;

Example:

A company wants a list of **all employees and all departments, including unmatched ones**:

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

FULL JOIN Departments

ON Employees.DepartmentID = Departments.DepartmentID;

This retrieves:

- Employees with **assigned departments**
- Employees **without departments (NULL values)**

- Departments **without assigned employees (NULL values)**

Real-World Application:

A **hotel chain** wants a report on **guests and room bookings**. Even if a guest hasn't booked a room or some rooms are vacant, they want a complete report.

```
SELECT Guests.Name, Bookings.RoomNumber

FROM Guests

FULL JOIN Bookings

ON Guests.GuestID = Bookings.GuestID;
```

Exercise:

1. Write a FULL JOIN query to retrieve all products and all orders, including those that don't match.
2. Why does FULL JOIN return more records than INNER JOIN?
3. In what scenarios is FULL JOIN preferred over LEFT JOIN and RIGHT JOIN?

Chapter 6: Case Study – Analyzing Sales Data Using SQL Joins

Scenario:
XYZ Electronics wants to analyze sales trends. They maintain:

- Customers (CustomerID, Name, Email)
- Orders (OrderID, CustomerID, Amount)

Challenges Faced:

1. Some customers have never placed orders.
2. Some orders have missing customer details.
3. Management needs a complete report of all customers and orders.

Solution Using Joins:

- INNER JOIN: Fetch only customers with orders.
- LEFT JOIN: Fetch all customers, even those without orders.
- RIGHT JOIN: Fetch all orders, even those missing customer details.
- FULL JOIN: Fetch all customers and all orders.

CONCLUSION

Understanding **INNER, LEFT, RIGHT, and FULL Joins** is crucial for effective database querying. Each join type serves a unique purpose in retrieving, analyzing, and managing relational data. Mastering joins enables database professionals to design **efficient, scalable, and high-performing** database systems. 🚀

WORKING WITH VIEWS AND INDEXES IN SQL

CHAPTER 1: INTRODUCTION TO VIEWS AND INDEXES

In SQL, **views and indexes** are essential tools that improve the efficiency, organization, and security of database operations. Both are used to optimize data retrieval and enhance performance in large-scale databases.

- **Views** act as **virtual tables** that store pre-defined SQL queries. They allow users to simplify complex queries, enforce security, and provide a structured way to access specific data without exposing the underlying tables.
- **Indexes** enhance database **performance** by speeding up query execution, reducing data retrieval time, and ensuring faster lookups, especially in large datasets.

By understanding how to effectively use **views and indexes**, database administrators and developers can significantly **improve database accessibility, security, and speed** while ensuring efficient data handling.

Example:

Consider an **e-commerce database** where customer details and purchase history are stored in separate tables. A **view** can be created to show only the most recent purchases, making it easier for the customer service team to access relevant information without dealing with complex SQL queries.

```
CREATE VIEW RecentPurchases AS
```

```
SELECT Customers.Name, Orders.OrderDate, Orders.Amount

FROM Customers

JOIN Orders ON Customers.CustomerID = Orders.CustomerID

WHERE Orders.OrderDate >= '2024-01-01';
```

This ensures that employees only see relevant purchase data without modifying the actual tables.

CHAPTER 2: UNDERSTANDING VIEWS IN SQL

What is a View?

A **view** is a virtual table created from the result of a SQL query. Unlike physical tables, a view does not store data; it dynamically presents the results of a predefined query whenever accessed.

Key Features of Views:

- **Simplifies Complex Queries:** Stores predefined queries for easy access.
- **Enhances Security:** Restricts access to specific columns or records.
- **Reduces Redundancy:** Provides a logical representation without duplicating data.
- **Supports Data Integrity:** Ensures consistency by abstracting database complexity.

Syntax for Creating a View:

```
CREATE VIEW view_name AS
```

SELECT column1, column2

FROM table_name

WHERE condition;

Example – Creating a View for Active Employees:

A company wants to display **only active employees** without exposing confidential salary details.

CREATE VIEW ActiveEmployees AS

SELECT EmployeeID, Name, Department

FROM Employees

WHERE Status = 'Active';

Now, when HR queries ActiveEmployees, they only see **active employees** without needing to filter data manually.

Updating a View:

Views do not store data, but they can be **refreshed** by modifying the base query.

ALTER VIEW ActiveEmployees AS

SELECT EmployeeID, Name, Department, HireDate

FROM Employees

WHERE Status = 'Active';

Exercise:

1. Create a view to display products with stock greater than 50.
2. Modify a view to include an additional column.

3. Explain how views enhance database security.

Chapter 3: Types of Views in SQL

Simple Views vs. Complex Views

Views can be categorized into **simple** and **complex** based on their structure and functionality.

Simple Views

- Created from a **single table**.
- Allows **INSERT, UPDATE, and DELETE** operations.
- Does not contain aggregation functions.

Example:

CREATE VIEW StudentView AS

SELECT StudentID, Name, Grade

FROM Students;

Complex Views

- Created from **multiple tables using JOINs**.
- Does **not allow modification** of base tables directly.
- Often includes aggregation, grouping, or subqueries.

Example:

CREATE VIEW EmployeeDepartment AS

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;

Exercise:

1. Create a simple view to show employee names and hire dates.
2. Write a complex view using JOIN to retrieve customer order history.
3. Why can't some views be updated?

CHAPTER 4: UNDERSTANDING INDEXES IN SQL

What is an Index?

An **index** is a database structure that improves the **speed of data retrieval operations**. It works like an index in a book, allowing quick lookups instead of scanning the entire database.

Key Benefits of Indexes:

- **Speeds Up Queries:** Reduces the time required for data retrieval.
- **Improves Search Efficiency:** Especially useful for large databases.
- **Optimizes Sorting and Filtering:** Queries using ORDER BY and WHERE execute faster.
- **Enhances Data Integrity:** Ensures unique constraints are efficiently maintained.

Syntax for Creating an Index:

CREATE INDEX index_name

ON table_name (column_name);

Example – Indexing Customer Emails for Fast Lookup:

CREATE INDEX idx_customer_email

ON Customers (Email);

Now, searching for a customer by email runs significantly faster:

SELECT * FROM Customers WHERE Email = 'john.doe@email.com';

Exercise:

1. Create an index for an Employee table on the LastName column.
2. Test a query with and without an index and compare performance.
3. What are the potential drawbacks of using too many indexes?

CHAPTER 5: TYPES OF INDEXES IN SQL

Primary Types of Indexes

SQL supports different types of indexes based on use cases and performance requirements.

1. Single-Column Index

- Created on a **single column** to speed up lookups.
- Useful for primary and foreign keys.

CREATE INDEX idx_product_price

ON Products (Price);

2. Composite Index

- Created on **multiple columns** to optimize queries filtering by multiple conditions.

CREATE INDEX idx_employee_name

ON Employees (LastName, FirstName);

3. Unique Index

- Ensures all values in a column are **unique**.
- Used for enforcing business rules (e.g., no duplicate emails).

CREATE UNIQUE INDEX idx_unique_email

ON Users (Email);

Exercise:

1. Create a composite index on an Orders table for CustomerID and OrderDate.
2. Explain when to use a **single-column index vs. composite index**.
3. What happens if two indexes are created on the same column?

Chapter 6: Case Study – Optimizing an E-commerce Database Using Views and Indexes

Scenario:

XYZ Electronics, an online retailer, faced performance issues as their **customer and orders tables grew large**. Queries took longer, affecting customer experience.

Challenges Faced:

1. **Slow queries** when fetching customer purchase history.
2. **Unnecessary access to confidential data**, risking security.
3. **High database load** due to full-table scans.

Solution Using Views and Indexes:

Step 1: Creating a View for Customer Purchases

To optimize data retrieval and enhance security, a **view** was created:

```
CREATE VIEW CustomerPurchases AS

SELECT Customers.Name, Orders.OrderDate, Orders.Amount

FROM Customers

JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Now, sales analysts access CustomerPurchases instead of complex queries.

Step 2: Creating an Index on CustomerID for Faster Lookups

```
CREATE INDEX idx_customer_id

ON Orders (CustomerID);
```

This dramatically reduced query execution time from **5 seconds to 0.5 seconds**.

Results:

- **80% faster data retrieval** for customer purchase history.

- **Restricted access to sensitive customer data**.
- **Reduced server load**, improving website speed.

Discussion Questions:

1. How did views help XYZ Electronics simplify customer reports?
2. Why was indexing necessary in this case?
3. Can indexes improve performance in all situations?

CONCLUSION

Understanding and effectively using **views and indexes** is essential for database **performance optimization, security, and usability**. Views **simplify queries and enhance security**, while indexes **improve data retrieval speed**. Both are critical for maintaining high-performance databases in enterprise applications.

SUBQUERIES AND NESTED QUERIES IN SQL

CHAPTER 1: INTRODUCTION TO SUBQUERIES AND NESTED QUERIES

Subqueries and nested queries are **advanced SQL techniques** used to retrieve data from multiple levels within a single SQL statement. A **subquery** (also known as an **inner query**) is a query that is embedded within another SQL query (the **outer query**). Subqueries are primarily used for filtering, comparison, and retrieving specific values dynamically without requiring multiple separate queries.

Nested queries help in breaking down complex problems into manageable **smaller queries**, making SQL statements more structured and readable. They allow users to **retrieve data from one query and use it as a condition in another query**, providing a flexible approach to handling large databases.

Key Benefits of Subqueries and Nested Queries:

- **Simplifies Complex Queries:** Helps break down large queries into manageable parts.
- **Enhances Data Filtering:** Retrieves only the required data dynamically.
- **Eliminates the Need for Temporary Tables:** Data processing is done in a single query execution.
- **Improves Database Performance:** Reduces query execution time by limiting unnecessary records.

Example:

A company wants to find employees who **earn more than the average salary** in the organization. Instead of manually calculating the average salary and then filtering employees, a **subquery** can be used:

SELECT Name, Salary

FROM Employees

WHERE Salary > (SELECT AVG(Salary) FROM Employees);

Here, the **inner query** (SELECT AVG(Salary) FROM Employees) calculates the **average salary**, and the **outer query** retrieves employees who **earn above that value**.

CHAPTER 2: UNDERSTANDING SUBQUERIES IN SQL

What is a Subquery?

A **subquery** is an SQL query **nested inside another query**. The results of the inner query are used by the outer query to perform filtering, aggregation, or data retrieval.

Types of Subqueries:

1. **Scalar Subquery:** Returns a **single value** (e.g., total count, maximum salary).
2. **Multi-Row Subquery:** Returns **multiple rows**, often used with IN or ANY.
3. **Correlated Subquery:** Executes once for each row in the outer query, referencing values from it.

Syntax for a Subquery:

SELECT column1, column2

FROM table1

WHERE column3 = (SELECT column FROM table2 WHERE condition);

Example – Finding Employees in the Highest Salary Department:

SELECT Name, DepartmentID

FROM Employees

WHERE DepartmentID = (SELECT DepartmentID FROM Departments ORDER BY Budget DESC LIMIT 1);

- The **inner query** retrieves the department with the **highest budget**.
- The **outer query** fetches employees from that department.

Exercise:

1. Write a query to find students who scored above the **average marks** in a class.
2. Retrieve all customers who placed **more than 5 orders**.
3. Why is a **scalar subquery** useful for conditions in WHERE clauses?

CHAPTER 3: NESTED QUERIES AND THEIR USAGE

What is a Nested Query?

A **nested query** is a **query inside another query** that executes first and passes results to the **main query**. While all subqueries are nested, not all nested queries are subqueries (some nested queries use joins instead of subqueries).

Key Features of Nested Queries:

- **Can be placed inside SELECT, WHERE, or FROM clauses**.
- **Can reference multiple tables** for retrieving data.
- **Executed in a hierarchical manner**, starting from the innermost query.

Example – Finding Customers Who Ordered a Specific Product:

```
SELECT Name

FROM Customers

WHERE CustomerID IN (

   SELECT CustomerID

   FROM Orders

   WHERE ProductID = (SELECT ProductID FROM Products WHERE
ProductName = 'Laptop')

);
```

- The **innermost query** retrieves the ProductID of "Laptop".
- The **middle query** fetches CustomerIDs who ordered that product.
- The **outer query** retrieves the Name of those customers.

Exercise:

1. Write a query to retrieve employees **who have worked for more than 3 years**.
2. Modify a nested query to use **JOIN** instead of subqueries.
3. When should a **nested query** be avoided in favor of **JOINs**?

CHAPTER 4: CORRELATED SUBQUERIES – A SPECIAL TYPE OF SUBQUERY

What is a Correlated Subquery?

A **correlated subquery** runs **once for each row** in the outer query, making it more dynamic but potentially **slower in execution**. It **references columns from the outer query**, creating a dependency.

Syntax for a Correlated Subquery:

```
SELECT column1, column2

FROM table1 t1

WHERE EXISTS (

   SELECT 1

   FROM table2 t2

   WHERE t1.common_column = t2.common_column

);
```

Example – Finding Employees Who Earn Above Their Department's Average Salary:

```
SELECT Name, DepartmentID, Salary

FROM Employees e1

WHERE Salary > (SELECT AVG(Salary) FROM Employees e2 WHERE
e1.DepartmentID = e2.DepartmentID);
```

- The **inner query** calculates the **average salary per department**.
- The **outer query** fetches employees who **earn more than their department's average**.

Exercise:

1. Write a query to find students **who scored above their class average**.
2. Explain how a **correlated subquery differs from a normal subquery**.
3. Why can a **correlated subquery** be slow for large databases?

CHAPTER 5: PERFORMANCE CONSIDERATIONS AND OPTIMIZATIONS FOR SUBQUERIES

Common Performance Issues with Subqueries:

- **Slow Execution:** If a subquery runs for each row, it can slow down performance.
- **Excessive Computation:** Subqueries requiring aggregations can be computationally heavy.
- **Alternative Approaches:** Using **JOINs** instead of subqueries often improves efficiency.

Optimizing Subqueries:

1. **Use Indexes** on the columns being queried to speed up lookups.
2. **Replace Subqueries with JOINs** where possible to improve performance.
3. **Use EXISTS Instead of IN** for better performance in correlated subqueries.

Example – Using JOIN Instead of Subquery:

Instead of:

```
SELECT Name

FROM Customers

WHERE CustomerID IN (SELECT CustomerID FROM Orders WHERE Amount > 500);
```

Use:

```
SELECT DISTINCT Customers.Name

FROM Customers

JOIN Orders ON Customers.CustomerID = Orders.CustomerID

WHERE Orders.Amount > 500;
```

This method is **faster** and more efficient in large datasets.

Exercise:

1. Modify a subquery-based query to use a **JOIN** instead.
2. Explain when **EXISTS** is better than **IN** in subqueries.
3. Why should **indexes** be used in large datasets?

Chapter 6: Case Study – Using Subqueries in an Online Shopping System

Scenario:

ABC Retail, an e-commerce platform, needs to analyze **customer orders** for targeted marketing.

Challenges Faced:

1. **Identifying customers who purchased a specific product**.
2. **Finding customers who placed the most orders**.
3. **Determining high-value customers (spending above average)**.

Solution Using Subqueries:

- **Retrieve customers who purchased laptops:**

```
SELECT Name FROM Customers WHERE CustomerID IN (
   SELECT CustomerID FROM Orders WHERE ProductID = (
      SELECT ProductID FROM Products WHERE ProductName =
'Laptop'
   )
);
```

- **Find high-value customers spending above average:**

```
SELECT Name, TotalSpent
FROM Customers
WHERE TotalSpent > (SELECT AVG(TotalSpent) FROM Customers);
```

Results:

- **Targeted marketing campaigns** for laptop buyers.
- **Personalized offers** for high-value customers.
- **Optimized sales strategies based on spending data.**

Discussion Questions:

1. How did subqueries help ABC Retail with customer analysis?

2. Why is filtering data dynamically using subqueries efficient?
3. What alternatives could be used to optimize these queries?

Conclusion

Understanding **subqueries and nested queries** is crucial for efficient **data retrieval and analysis**. Mastering these concepts allows database professionals to **design flexible, scalable, and optimized** SQL queries for real-world applications. 🚀

ASSIGNMENT

DESIGN A SMALL RELATIONAL DATABASE SCHEMA AND WRITE SQL QUERIES TO MANAGE DATA.

DESIGNING A SMALL RELATIONAL DATABASE SCHEMA AND WRITING SQL QUERIES TO MANAGE DATA

CHAPTER 1: INTRODUCTION TO RELATIONAL DATABASE SCHEMA DESIGN

A **relational database schema** is a blueprint that defines how data is structured, stored, and managed in a relational database. It consists of **tables, columns, relationships, and constraints** that ensure data integrity, minimize redundancy, and improve efficiency. **A well-designed schema** is essential for optimizing queries, ensuring scalability, and maintaining data consistency.

In a relational database, data is organized into tables, where each table has a **primary key** that uniquely identifies each record. Relationships between tables are established using **foreign keys**, ensuring data is connected and meaningful. **Normalization** techniques are used to eliminate redundant data and ensure that tables follow a structured format.

Key Features of a Relational Database Schema:

- **Well-defined tables with relationships** to avoid data duplication.
- **Primary keys (PK) and foreign keys (FK)** to enforce relationships.
- **Constraints** such as NOT NULL, UNIQUE, and CHECK to maintain data integrity.
- **Indexes** to speed up data retrieval operations.

Example:

A **library management system** consists of the following tables:

1. Books (stores book details).
2. Members (stores information about members).
3. BorrowedBooks (records book borrow transactions).

Each table has a **primary key**, and relationships are maintained using **foreign keys**.

CHAPTER 2: DESIGNING A SMALL RELATIONAL DATABASE SCHEMA

Understanding the Schema Design

Let's design a **small relational database schema** for an **online retail store** that tracks **customers, products, and orders**.

Tables and Their Relationships:

1. **Customers Table** - Stores customer information.
2. **Products Table** - Stores details of products available for sale.
3. **Orders Table** - Records purchases made by customers.
4. **OrderDetails Table** - Stores individual product details for each order.

Entity-Relationship (ER) Diagram Structure:

- A **customer** can place multiple orders (**one-to-many relationship**).
- An **order** can contain multiple products (**many-to-many relationship**).
- A **product** can be purchased in multiple orders (**many-to-many relationship**).

Database Schema Definition (SQL Code):

```
-- Customers Table
CREATE TABLE Customers (
  CustomerID INT PRIMARY KEY AUTO_INCREMENT,
  Name VARCHAR(100) NOT NULL,
  Email VARCHAR(100) UNIQUE NOT NULL,
  Phone VARCHAR(15),
  Address TEXT
);

-- Products Table
CREATE TABLE Products (
  ProductID INT PRIMARY KEY AUTO_INCREMENT,
  ProductName VARCHAR(100) NOT NULL,
  Price DECIMAL(10,2) NOT NULL CHECK (Price > 0),
  StockQuantity INT NOT NULL CHECK (StockQuantity >= 0)
);

-- Orders Table
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY AUTO_INCREMENT,
  CustomerID INT,
  OrderDate DATE DEFAULT CURRENT_DATE,
  TotalAmount DECIMAL(10,2),
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE CASCADE
);

-- OrderDetails Table (Many-to-Many Relationship)
CREATE TABLE OrderDetails (
  OrderDetailID INT PRIMARY KEY AUTO_INCREMENT,
  OrderID INT,
  ProductID INT,
  Quantity INT NOT NULL CHECK (Quantity > 0),
  SubTotal DECIMAL(10,2),
  FOREIGN KEY (OrderID) REFERENCES Orders(OrderID) ON DELETE CASCADE,
  FOREIGN KEY (ProductID) REFERENCES Products(ProductID) ON DELETE CASCADE
);
```

This schema ensures **data integrity**, establishes **relationships**, and allows efficient order tracking.

Exercise:

1. Modify the schema to add a **Discount column** in the Orders table.
2. Add a **category column** to the Products table.
3. Why are FOREIGN KEY constraints important in relational databases?

CHAPTER 3: INSERTING DATA INTO THE DATABASE

Adding Customers, Products, and Orders

To populate the database, we use INSERT queries:

Inserting Customers:

```
INSERT INTO Customers (Name, Email, Phone, Address)
VALUES ('Alice Johnson', 'alice@email.com', '9876543210', '123 Main St');
```

Inserting Products:

```
INSERT INTO Products (ProductName, Price, StockQuantity)

VALUES ('Laptop', 1200.50, 10),

    ('Smartphone', 800.00, 15);
```

Inserting Orders:

```
INSERT INTO Orders (CustomerID, TotalAmount)

VALUES (1, 2000.00);
```

Inserting Order Details:

```
INSERT INTO OrderDetails (OrderID, ProductID, Quantity, SubTotal)

VALUES (1, 1, 1, 1200.50),

    (1, 2, 1, 800.00);
```

These queries **populate the database** and maintain **referential integrity** through foreign key relationships.

Exercise:

1. Add three more customers and insert their orders.
2. Insert a new product and add it to an existing order.
3. What happens if we insert an order for a non-existent CustomerID?

CHAPTER 4: RETRIEVING AND MANAGING DATA USING QUERIES

Fetching Data Using SELECT Queries

Once data is inserted, we use SELECT queries to retrieve information efficiently.

Retrieve All Customers:

```
SELECT * FROM Customers;
```

Retrieve Orders Placed by a Customer:

```
SELECT Orders.OrderID, Customers.Name, Orders.TotalAmount

FROM Orders

JOIN Customers ON Orders.CustomerID = Customers.CustomerID

WHERE Customers.Name = 'Alice Johnson';
```

Retrieve Total Sales for Each Product:

SELECT Products.ProductName, SUM(OrderDetails.Quantity) AS TotalSold

FROM OrderDetails

JOIN Products ON OrderDetails.ProductID = Products.ProductID

GROUP BY Products.ProductName;

These queries **extract meaningful insights** from the relational database.

Exercise:

1. Write a query to find the total number of orders placed by each customer.
2. Retrieve the list of products that have not been sold.
3. How does JOIN help in retrieving data from multiple tables?

CHAPTER 5: UPDATING AND DELETING RECORDS IN THE DATABASE

Updating Data Using UPDATE Queries

Updating Product Stock After Purchase:

UPDATE Products

SET StockQuantity = StockQuantity - 1

WHERE ProductID = 1;

Updating Customer Address:

UPDATE Customers

SET Address = '456 New Street'

WHERE CustomerID = 1;

Deleting Data Using DELETE Queries

Delete an Order and Its Associated Details:

DELETE FROM Orders WHERE OrderID = 1;

Since OrderDetails references Orders, deleting an order **also deletes related records** due to the ON DELETE CASCADE constraint.

Exercise:

1. Write a query to update the price of a product.
2. Delete a customer and ensure all their orders are removed.
3. Why is ON DELETE CASCADE useful in relational databases?

CHAPTER 6: CASE STUDY – IMPLEMENTING A RELATIONAL DATABASE FOR AN ONLINE BOOKSTORE

Scenario:

An online bookstore needs a **structured relational database** to track **books, customers, and orders** efficiently.

Challenges Faced:

- Customers struggle to view their past purchases.
- Books are often out of stock due to lack of tracking.
- Order processing is slow and inefficient.

Solution Using a Relational Database:

1. **Customers Table:** Stores customer details.
2. **Books Table:** Manages book inventory.
3. **Orders Table:** Tracks purchase history.
4. **OrderDetails Table:** Stores order-product relationships.

SQL Queries for Analysis:

- **Retrieve best-selling books:**

```
SELECT Books.Title, SUM(OrderDetails.Quantity) AS TotalSold

FROM OrderDetails

JOIN Books ON OrderDetails.BookID = Books.BookID

GROUP BY Books.Title;
```

- **Find customers with the most orders:**

```
SELECT Customers.Name, COUNT(Orders.OrderID) AS TotalOrders

FROM Orders

JOIN Customers ON Orders.CustomerID = Customers.CustomerID

GROUP BY Customers.Name;
```

Discussion Questions:

1. How did relational database design improve the bookstore's efficiency?
2. Why is GROUP BY useful for generating business reports?

3. How can indexing enhance query performance?

CONCLUSION

Designing a **relational database schema** and managing data using SQL queries is essential for **efficient data organization, retrieval, and analysis**. Mastering these concepts enables businesses to **scale operations, improve performance, and enhance data integrity**. 🚀