Database Management System (DBMS) Comprehensive Documentation

Table of Contents

Unit I: Introduction to Database Management System

1.1 Introduction and Applications of DBMS

Definition

A **Database Management System (DBMS)** is an organized collection of software tools and data that facilitates the storage, retrieval, and manipulation of data in databases. It serves as an intermediary between end-users, applications, and the database itself to ensure data consistency, security, and efficient access.

Key Components of DBMS:

- **Hardware:** Physical devices like servers, storage systems, and networking equipment.
- **Software:** The DBMS software (e.g., MySQL, Oracle, PostgreSQL) that manages data.
- **Data:** The actual information stored in the database.
- **Procedures:** Rules and guidelines for data management.
- **Database Access Languages:** SQL, QBE (Query By Example), etc.

Purpose

The primary purposes of a DBMS include:

1. **Data Abstraction:** Simplifies data interaction by hiding the complex details of data storage and management.
2. **Data Independence:** Changes in data storage do not affect application programs.
3. **Data Integrity and Security:** Enforces data accuracy and protects data from unauthorized access.
4. **Efficient Data Access:** Provides optimized mechanisms for data retrieval and storage.
5. **Data Administration:** Facilitates administration tasks like backup, recovery, and user management.

Examples

- **Relational DBMS (RDBMS):** MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server.
- **NoSQL DBMS:** MongoDB, Cassandra, Redis, Couchbase.
- **Object-Oriented DBMS:** ObjectDB, db4o.
- **Hierarchical DBMS:** IBM Information Management System (IMS).
- **Network DBMS:** Integrated Data Store (IDS), IDMS.

Applications

DBMSs are utilized across various industries and scenarios. Below are some real-world applications:

- **Banking Systems:**
  - Managing customer accounts and transactions.
  - Facilitating online banking services.
  - Ensuring secure and accurate transaction records.
- **E-commerce Platforms:**
  - Handling product catalogs, inventory management, orders, and customer data.
  - Supporting recommendation systems and personalized user experiences.
- **Healthcare Systems:**
  - Managing patient records, appointment scheduling, and medical histories.
  - Ensuring compliance with regulations like HIPAA.
- **Educational Institutions:**
  - Storing student records, grades, course registrations, and faculty information.
  - Supporting learning management systems (LMS).
- **Inventory Management:**
  - Tracking stock levels, orders, suppliers, and product details.
  - Enabling real-time inventory updates and reorder alerts.
- **Telecommunications:**
  - Managing subscriber information, call records, billing, and network configurations.
  - Supporting customer service operations.
- **Government Agencies:**
  - Handling citizen data, public services, tax records, and administrative functions.
  - Ensuring data transparency and accessibility.

Diagram 1.1: DBMS Architecture Overview

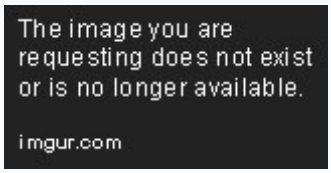

Figure 1.1: Three-Tier DBMS Architecture

Explanation:

1. **Users/Clients:** Interact with the DBMS through applications, queries, forms, or reports.
2. **DBMS Software:** Handles query processing, transaction management, storage management, and security.
3. **Physical Database:** The actual storage of data on hardware devices.

1.2 Key Concepts

Understanding foundational concepts is crucial for grasping DBMS operations.

Data vs. Information

- **Data:**
  - **Definition:** Raw, unprocessed facts and figures without context.
  - **Examples:** Numbers, characters, dates.
  - **Usage:** Basis for creating information.
- **Information:**
  - **Definition:** Processed or organized data that is meaningful and useful.
  - **Examples:** Reports, analytics, summaries.
  - **Usage:** Supports decision-making and analysis.

Data Items, Records, and Files

- **Data Items:**
  - **Definition:** The smallest units of data, representing single attributes.
  - **Examples:** A field in a table like "FirstName" or "Salary."
- **Records:**
  - **Definition:** A collection of related data items that describe a single entity.
  - **Examples:** An employee record containing EmployeeID, FirstName, LastName, DepartmentID.
- **Files:**
  - **Definition:** A collection of related records.
  - **Examples:** An Employee file storing all employee records.

Metadata

- **Definition:**
  - Data about data; it describes the structure, constraints, and properties of data stored in the database.
- **Components:**
  - **Schema:** The overall design of the database, including tables, fields, relationships.
  - **Data Dictionary:** A detailed repository of metadata, including definitions, constraints, and associations.
- **Importance:**
  - Facilitates data management and consistency.
  - Enables users and applications to understand data structures and constraints.

Data Dictionary

- **Definition:**
  - A centralized repository within the DBMS that stores metadata about the database's structure, constraints, and relationships.
- **Contents:**
  - **Tables:** Names, definitions, and relationships.
  - **Columns:** Data types, constraints, default values.
  - **Indexes:** Information about indexed columns and structures.
  - **Views:** Definitions of virtual tables.
  - **Procedures and Functions:** Stored procedures, triggers, and user-defined functions.
- **Benefits:**
  - Enhances data consistency and integrity.

- Supports efficient data retrieval and management.
- Aids in database maintenance and schema changes.

1.3 Database Architecture

DBMS architecture defines how data is stored, managed, and accessed. The most widely accepted model is the **Three-Level Architecture**, also known as **ANSI/SPARC Architecture**.

Three-Level Architecture

1. **Internal Level (Physical Level):**
    a. **Description:** Details how data is physically stored on storage media.
    b. **Focus Areas:** Data structures, file organizations, indexing methods, storage allocation.
    c. **User Interaction:** DBAs primarily interact at this level for performance tuning and storage optimization.
2. **Conceptual Level (Logical Level):**
    a. **Description:** Defines the logical structure of the entire database for a community of users.
    b. **Focus Areas:** Entities, relationships, constraints, views, and overall schema without considering physical storage.
    c. **User Interaction:** Database designers and application developers interact at this level to design and understand the database schema.
3. **External Level (View Level):**
    a. **Description:** Represents how individual users or user groups view the data.
    b. **Focus Areas:** Specific subsets of the database tailored to user requirements, security considerations.
    c. **User Interaction:** End-users interact with the database through applications that present data as per their needs.

Benefits of Three-Level Architecture

- **Data Abstraction:** Separates physical data storage from logical data structure and user views.
- **Data Independence:**
    - **Physical Data Independence:** Changes in the internal level do not affect the conceptual and external levels.
    - **Logical Data Independence:** Changes in the conceptual level do not impact the external views.
- **Simplifies Database Management:** Facilitates easier maintenance, scalability, and security management.

Diagram 1.3: Three-Level Architecture

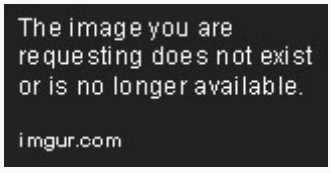

Figure 1.3: ANSI/SPARC Three-Level Architecture

Explanation:

- **External Views:** Different user perspectives and interactions.
- **Conceptual Schema:** Single, integrated view of the entire database.
- **Internal Schema:** Efficient physical storage mechanisms.

1.4 View of Data

The **View of Data** refers to how data is perceived and interacted with at different abstraction levels in the DBMS architecture.

Physical Level

- **Description:**
  - Deals with the actual storage of data on hardware devices.
  - Concerns data formats, storage locations, and access paths.
- **Components:**
  - **Storage Structures:** How data is organized on disks (e.g., B-trees, hash tables).
  - **File Organizations:** Sequential, indexed, hashed file structures.
  - **Indexing:** Structures that improve data retrieval speed (e.g., primary indexes, secondary indexes).
- **Importance:**
  - Optimization of storage for performance.
  - Efficient management of large datasets.
  - Minimizing storage costs and access times.

Logical Level

- **Description:**
  - Defines what data is stored and the relationships among data elements.
  - Focuses on data structures like tables, views, and relationships.
- **Components:**
  - **Tables:** Structures consisting of rows and columns.
  - **Views:** Virtual tables created by querying one or more tables.
  - **Constraints:** Rules to maintain data integrity (e.g., primary keys, foreign keys).
- **Importance:**
  - Provides a clear and organized representation of data.
  - Facilitates data manipulation and querying.
  - Ensures data consistency and integrity.

Conceptual Level

- **Description:**
  - Offers a high-level, unified view of the entire database.
  - Integrates multiple user views into a single framework.
- **Components:**
  - **Entities and Relationships:** Abstract representations of real-world objects and their interactions.
  - **Schema:** Comprehensive blueprint defining all logical structures.
  - **Security Policies:** Rules governing data access and usage.

- **Importance:**
    - Serves as a bridge between the external and internal levels.
    - Ensures that all user views are consistent and integrated.
    - Facilitates data sharing and reduces redundancy.

Diagram 1.4: Views of Data

Figure 1.4: Data Views Across Levels

Explanation:

- **Physical Level:** Data storage details.
- **Logical Level:** Schema definitions and table structures.
- **Conceptual Level:** Overall database blueprint.

1.5 Database Languages

DBMS incorporates various languages to facilitate different operations, each serving distinct purposes within the database lifecycle.

Data Definition Language (DDL)

- **Purpose:** Define, modify, and remove database objects like schemas, tables, indexes, and views.
- **Key Commands:**
    - **CREATE:** Establish new database objects.
    - **ALTER:** Modify existing database objects.
    - **DROP:** Remove database objects.
    - **TRUNCATE:** Remove all records from a table without deleting the table itself.
    - **RENAME:** Change the name of a database object.
- **Examples:**

```
-- Create a new table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    EnrollmentDate DATE
);
```

```
-- Alter a table to add a new column
ALTER TABLE Students
ADD Email VARCHAR(100);

-- Drop a table
DROP TABLE Students;
```

Data Manipulation Language (DML)

- **Purpose:** Retrieve and manipulate data stored in the database.
- **Key Commands:**
  - **SELECT:** Retrieve data from one or more tables.
  - **INSERT:** Add new records to a table.
  - **UPDATE:** Modify existing records.
  - **DELETE:** Remove records from a table.
  - **MERGE:** Insert, update, or delete records based on conditions.
- **Examples:**

```
-- Insert a new record
INSERT INTO Students (StudentID, FirstName, LastName,
EnrollmentDate)
VALUES (1001, 'Alice', 'Johnson', '2023-09-01');

-- Update a record
UPDATE Students
SET Email = 'alice.johnson@example.com'
WHERE StudentID = 1001;

-- Delete a record
DELETE FROM Students
WHERE StudentID = 1001;
```

Data Control Language (DCL)

- **Purpose:** Control access to data within the database.
- **Key Commands:**
  - **GRANT:** Provide permissions to users or roles.
  - **REVOKE:** Remove previously granted permissions.
  - **DENY:** Explicitly deny permissions (supported in some DBMS).
- **Examples:**

```
-- Grant SELECT and INSERT permissions on Students table to user
'john'
GRANT SELECT, INSERT ON Students TO john;

-- Revoke INSERT permission from user 'john'
```

```
REVOKE INSERT ON Students FROM john;
```

Transaction Control Language (TCL)

- **Purpose:** Manage transactions to ensure data integrity.
- **Key Commands:**
    - **COMMIT:** Save all changes made in the current transaction.
    - **ROLLBACK:** Undo changes made in the current transaction.
    - **SAVEPOINT:** Set a point within a transaction to which you can roll back.
    - **SET TRANSACTION:** Configure transaction properties.
- **Examples:**

```
-- Start a transaction
BEGIN TRANSACTION;

-- Perform some operations
INSERT INTO Students (StudentID, FirstName, LastName,
EnrollmentDate)
VALUES (1002, 'Bob', 'Smith', '2023-09-02');

-- Commit the transaction
COMMIT;

-- Start another transaction
BEGIN TRANSACTION;

-- Perform some operations
UPDATE Students
SET LastName = 'Williams'
WHERE StudentID = 1002;

-- Rollback the transaction
ROLLBACK;
```

Data Retrieval Language (DRL)

- **Purpose:** Special subset of SQL focused on querying data, primarily using the **SELECT** statement.
- **Features:**
    - Complex queries with joins, subqueries, aggregations.
    - Supports various clauses like WHERE, GROUP BY, HAVING, ORDER BY.
    - Facilitates sophisticated data analysis and reporting.
- **Example:**

```
-- Retrieve names of students enrolled after a specific date,
ordered by last name
```

```
SELECT FirstName, LastName
FROM Students
WHERE EnrollmentDate > '2023-01-01'
ORDER BY LastName ASC;
```

1.6 Core Functionalities

A robust DBMS offers essential functionalities that handle various aspects of data management. These core functionalities ensure that data is stored securely, accessed efficiently, and maintained consistently.

Data Storage

- **Efficient Storage Mechanisms:**
  - Utilize indexing to speed up data retrieval.
  - Implement data compression to save storage space.
  - Support various storage models (e.g., row-oriented, column-oriented).
- **Data Structures:**
  - **B-trees, B+ trees:** Commonly used for indexing due to their balanced structure and efficient search capabilities.
  - **Hash Tables:** Used for quick data retrieval based on key values.
- **Disk Management:**
  - Organize data files on disk to minimize access times.
  - Implement caching strategies to reduce disk I/O.

Query Processing

- **Query Parser:** Interprets SQL queries and checks for syntax errors.
- **Query Optimizer:** Determines the most efficient execution plan for a query using cost-based or rule-based optimization techniques.
- **Query Executor:** Executes the query based on the optimized plan, retrieving and returning the requested data.
- **Examples of Query Optimization:**
  - **Selection Pushdown:** Applying WHERE clauses early in the query execution to filter data sooner.
  - **Join Order Optimization:** Determining the most efficient sequence to perform joins between tables.

Transaction Management

- **ACID Properties:**
  - **Atomicity:** Ensures that all operations within a transaction are completed successfully; if not, the transaction is rolled back.
  - **Consistency:** Guarantees that the database moves from one valid state to another, maintaining all defined rules and constraints.
  - **Isolation:** Ensures that transactions occur independently without interference, even when executed concurrently.

    - **Durability:** Once a transaction is committed, its changes are permanent and survive system failures.
- **Concurrency Control:**
    - **Locking Mechanisms:** Prevent simultaneous conflicting operations on the same data.
        - **Pessimistic Locking:** Locks data resources to prevent conflicts.
        - **Optimistic Locking:** Assumes conflicts are rare and handles them if they occur.
    - **Deadlock Detection and Resolution:** Identifies and resolves situations where two or more transactions are waiting indefinitely for each other to release locks.
- **Recovery Mechanisms:**
    - **Logging:** Maintains logs of all transactions to facilitate recovery after failures.
    - **Checkpointing:** Saves the state of the database at specific points to minimize recovery time.

Backup and Recovery

- **Backup Strategies:**
    - **Full Backup:** Entire database is backed up at once.
    - **Incremental Backup:** Only changes since the last backup are saved.
    - **Differential Backup:** Changes since the last full backup are saved.
- **Recovery Techniques:**
    - **Point-in-Time Recovery:** Restores the database to a specific moment.
    - **Crash Recovery:** Restores the database after a system crash using logs and backups.
- **Disaster Recovery Planning:**
    - **Redundancy:** Maintaining duplicate copies of data in different locations.
    - **Failover Mechanisms:** Automatically switching to backup systems in case of failures.

Security Management

- **Authentication:** Verifying user identities through credentials like usernames and passwords.
- **Authorization:** Granting or restricting access to database objects based on user roles and permissions.
- **Encryption:** Protecting data at rest and in transit to prevent unauthorized access.
- **Auditing:** Tracking and logging user activities to monitor for suspicious behavior and ensure compliance.

Data Integrity Management

- **Entity Integrity:** Ensures that each table has a primary key and that it uniquely identifies each record.
- **Referential Integrity:** Maintains valid relationships between tables through foreign keys.
- **Domain Integrity:** Ensures that data entered into a column adheres to defined data types, formats, and constraints.
- **User-Defined Integrity:** Custom rules defined by users to meet specific business requirements.

1.7 Roles in DBMS

Different stakeholders interact with the DBMS, each fulfilling distinct roles to ensure the system's effective operation and data integrity.

Database Users

1. **End Users:**
    a. **Types:**
        i. **Casual Users:** Occasional users who access the database using standard query tools.
        ii. **Naive Users:** Use predefined applications without understanding the underlying database.
        iii. **Power Users:** Utilize advanced query languages and tools to perform complex operations.
        iv. **Application Programmers:** Develop software applications that interact with the database.
    b. **Responsibilities:**
        i. Accessing and manipulating data to fulfill business needs.
        ii. Creating reports, forms, and queries.
2. **Database Developers:**
    a. **Role:**
        i. Design and develop database schemas, stored procedures, triggers, and functions.
        ii. Optimize queries and ensure efficient database performance.
    b. **Responsibilities:**
        i. Writing complex SQL queries and managing database objects.
        ii. Collaborating with application developers to integrate databases with software applications.
3. **Database Administrators (DBAs):**
    a. **Role:**
        i. Oversee the overall management, maintenance, and security of the database.
    b. **Responsibilities:**
        i. Installing, configuring, and upgrading DBMS software.
        ii. Designing and implementing database schemas and structures.
        iii. Monitoring database performance and tuning for optimal efficiency.
        iv. Managing user access, roles, and permissions.
        v. Performing regular backups and ensuring data recovery mechanisms.
        vi. Ensuring data integrity and enforcing security policies.
        vii. Planning for database scalability and growth.
4. **System Analysts:**
    a. **Role:**
        i. Analyze system requirements and design database solutions to meet organizational needs.
    b. **Responsibilities:**
        i. Gathering and documenting business requirements.
        ii. Designing logical and physical database models.
        iii. Collaborating with stakeholders to ensure database solutions align with business objectives.

Diagram 1.7: Roles in DBMS

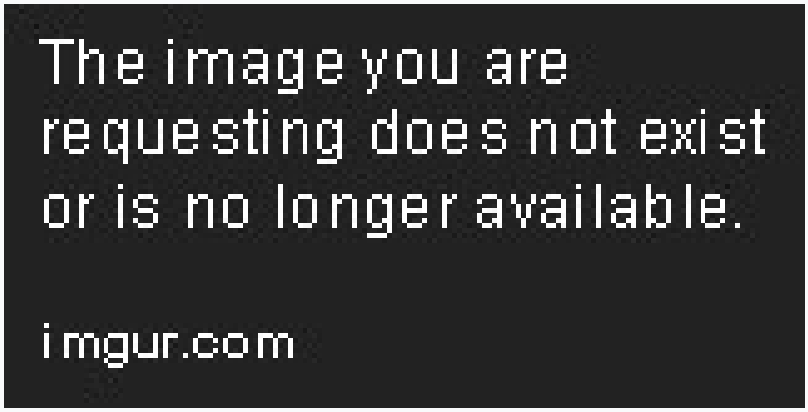

Figure 1.7: Different Roles in DBMS Architecture

Explanation:

- **End Users:** Direct interaction with the database for various tasks.
- **Developers:** Create and optimize database structures and queries.
- **DBAs:** Manage and maintain the overall database environment.
- **System Analysts:** Bridge between business requirements and database solutions.

Unit II: Relational Model and Entity-Relationship Model

2.1 Relational Database Structure

The **Relational Model** is one of the most widely used data models, introduced by E.F. Codd in 1970. It structures data into tables (relations) consisting of rows and columns.

Tables (Relations)

- **Definition:**
  - A table is a collection of related data entries and consists of rows and columns.
- **Components:**
  - **Rows (Tuples):** Each row represents a unique record in the table.
  - **Columns (Attributes):** Each column represents a specific property or field of the entity.
- **Properties:**
  - **Uniqueness:** Each table has a primary key that uniquely identifies each row.
  - **No Ordering:** The order of rows and columns is irrelevant in a mathematical model.
  - **Atomic Values:** Each cell in the table holds a single value (as per First Normal Form).

Attributes

- **Definition:**
  - Attributes are the properties or characteristics of an entity represented by columns in a table.
- **Types of Attributes:**
  - **Simple Attributes:** Indivisible attributes (e.g., FirstName).
  - **Composite Attributes:** Attributes that can be divided into smaller sub-parts (e.g., Address: Street, City, Zip Code).

    - **Derived Attributes:** Attributes derived from other attributes (e.g., Age from Date of Birth).
    - **Multi-valued Attributes:** Attributes that can have multiple values for a single entity (e.g., PhoneNumbers).
- **Examples:**
    - **Employee Table:**
        - EmployeeID (Primary Key)
        - FirstName
        - LastName
        - DepartmentID (Foreign Key)
        - Email

Schema

- **Definition:**
    - The schema defines the structure of a table, including column names, data types, and constraints.
- **Components:**
    - **Table Name:** Unique identifier for the table.
    - **Column Definitions:** Each column's name, data type, and constraints.
- **Example: Employee Table Schema**

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DepartmentID INT,
    Email VARCHAR(100) UNIQUE
);
```

Example: Employee Table

EmployeeID (PK)	FirstName	LastName	DepartmentID (FK)	Email
1	John	Doe	10	john.doe@example.com
2	Jane	Smith	20	jane.smith@example.com
3	Bob	Johnson	10	bob.johnson@example.com

- **Notes:**
    - **Primary Key (PK):** EmployeeID uniquely identifies each employee.
    - **Foreign Key (FK):** DepartmentID references the Departments table.
    - **Unique Constraint:** Email must be unique across employees.

Real-World Reference

Customer Relationship Management (CRM) Systems:

- Utilize relational databases to store customer information, interactions, sales data, and support tickets.

- Tables may include Customers, Orders, Products, SupportTickets, etc., interlinked through foreign keys to reflect relationships.

2.2 Keys and Constraints

Keys and constraints are fundamental to maintaining data integrity and establishing relationships between tables in a relational database.

Primary Key (PK)

- **Definition:**
  - A primary key is a column or a set of columns that uniquely identifies each row in a table.
- **Characteristics:**
  - **Uniqueness:** No two rows can have the same primary key value.
  - **Non-Null:** Primary key columns cannot contain NULL values.
  - **Immutability:** Primary key values should not change over time.
- **Examples:**
  - **EmployeeID** in the Employees table.
  - **StudentID** in the Students table.
  - **ISBN** in the Books table.
- **Best Practices:**
  - Use surrogate keys (e.g., auto-incremented integers) when natural keys are unsuitable.
  - Ensure primary keys are stable and unlikely to change.

Foreign Key (FK)

- **Definition:**
  - A foreign key is a column or a set of columns in a table that establishes a link between data in two tables.
- **Purpose:**
  - Enforces referential integrity by ensuring that the value in the foreign key matches a primary key in the referenced table.
- **Examples:**
  - **DepartmentID** in the Employees table referencing **DepartmentID** in the Departments table.
  - **CustomerID** in the Orders table referencing **CustomerID** in the Customers table.
- **Constraints:**
  - Can accept NULL values unless specified otherwise.
  - Must match the data type of the referenced primary key.

Unique Key

- **Definition:**
  - A unique key constraint ensures that all values in a column or a set of columns are distinct across all rows.
- **Characteristics:**
  - Similar to primary keys in enforcing uniqueness but can accept a single NULL value.

- **Examples:**
  - **Email** in the Employees table.
  - **Username** in the Users table.
- **Usage:**
  - To enforce uniqueness on non-primary key columns that require distinct values.

Composite Key

- **Definition:**
  - A composite key is a primary key composed of two or more columns used together to uniquely identify a record.
- **Example:**
  - **OrderID** and **ProductID** in the OrderDetails table to uniquely identify each order-product pair.

OrderID	ProductID	Quantity
1001	P01	10
1001	P02	5
1002	P01	7

- **Use Cases:**
  - When no single column is sufficient to ensure uniqueness.
  - In associative tables representing many-to-many relationships.

Constraints Overview

Constraints enforce rules at the database level to maintain data integrity and consistency.

1. **NOT NULL:**
   a. **Purpose:** Prevents NULL values in a column.
   b. **Usage:** Ensures that essential data is always provided.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE
);
```

2. **CHECK:**
   a. **Purpose:** Ensures that all values in a column satisfy specific conditions.
   b. **Usage:** Enforces business rules at the database level.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Age INT,
    CHECK (Age >= 18)
);
```

3. **DEFAULT:**
   a. **Purpose:** Sets a default value for a column when no value is specified during insertion.

b. **Usage:** Provides automatic values to maintain data consistency.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    HireDate DATE DEFAULT GETDATE()
);
```

4. **UNIQUE:**
    a. **Purpose:** Ensures all values in a column or a group of columns are unique.
    b. **Usage:** Enforces uniqueness without declaring a primary key.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE
);
```

5. **PRIMARY KEY:**
    a. **Purpose:** Uniquely identifies each record in a table and enforces NOT NULL and UNIQUE constraints.
    b. **Usage:** Defines the main identifier for table records.

```
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50) NOT NULL
);
```

6. **FOREIGN KEY:**
    a. **Purpose:** Ensures that the values in one table match values in another table, establishing relationships.
    b. **Usage:** Maintains referential integrity between related tables.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

Real-World References

E-commerce Systems:

- **Orders Table:**
    - **Primary Key:** OrderID.
    - **Foreign Key:** CustomerID referencing Customers table.
    - **Constraints:** CHECK to ensure OrderAmount is positive.

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE DEFAULT GETDATE(),
    OrderAmount DECIMAL(10, 2) CHECK (OrderAmount > 0),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Hospital Management Systems:

- **Patients Table:**
  - **Primary Key:** PatientID.
  - **Unique Constraint:** SocialSecurityNumber.

```
CREATE TABLE Patients (
    PatientID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    SocialSecurityNumber CHAR(9) UNIQUE,
    DateOfBirth DATE NOT NULL
);
```

2.3 Relational Query Languages

Relational Query Languages enable users to interact with relational databases, performing various operations to retrieve, manipulate, and manage data.

SQL (Structured Query Language)

SQL is the standard language for relational database management systems. It allows users to perform a wide range of operations, from data definition and manipulation to data control and transaction management.

Key Features:

1. **Declarative Nature:**
   a. Users specify what data they want without detailing how to retrieve it.
2. **Comprehensive Functionality:**
   a. Supports complex queries, aggregations, joins, views, procedures, and more.
3. **Standardization:**
   a. ANSI and ISO standardized, ensuring compatibility across different DBMSs with slight variations.

SQL Operations

1. **Data Retrieval:**
   a. **SELECT Statements:** Retrieve specific data based on criteria.
2. **Data Manipulation:**
   a. **INSERT, UPDATE, DELETE:** Modify data within tables.

3. **Data Definition:**
    a. **CREATE, ALTER, DROP:** Define and modify database structures.
4. **Data Control:**
    a. **GRANT, REVOKE:** Manage user permissions and access.

Relational Algebra Operators

- **Select (σ):** Filters rows based on a condition.

```
SELECT * FROM Employees WHERE DepartmentID = 10;
```

- **Project (π):** Selects specific columns from a table.

```
SELECT FirstName, LastName FROM Employees;
```

- **Join (⋈):** Combines rows from two tables based on a related column.

```
SELECT Employees.FirstName, Departments.DepartmentName
FROM Employees
JOIN Departments ON Employees.DepartmentID =
Departments.DepartmentID;
```

- **Union (∪):** Combines the result sets of two queries, removing duplicates.

```
SELECT EmployeeID FROM Employees
UNION
SELECT ManagerID FROM Departments;
```

- **Intersection (∩):** Retrieves common records between two queries. *(Note: Not directly supported in some DBMSs; achieved using joins or subqueries.)*
- **Difference (−):** Finds records present in one query but not in another.

```
SELECT EmployeeID FROM Employees
EXCEPT
SELECT EmployeeID FROM Managers;
```

Example SQL Query

Scenario: Retrieve First and Last Names of Employees in the Engineering Department Located in 'New York'.

```
SELECT Employees.FirstName, Employees.LastName,
Departments.DepartmentName
FROM Employees
JOIN Departments ON Employees.DepartmentID =
Departments.DepartmentID
WHERE Departments.Location = 'New York';
```

- **Explanation:**
    - **SELECT:** Specifies the columns to retrieve.

- **FROM:** Specifies the primary table.
- **JOIN:** Combines Employees and Departments tables based on DepartmentID.
- **WHERE:** Filters records where Department location is 'New York'.

Relational Query Language Extensions

- **Procedural Extensions:** Incorporate programming constructs like loops and conditionals. Example: PL/SQL (Oracle), T-SQL (SQL Server).

```
BEGIN
    IF EXISTS (SELECT * FROM Employees WHERE DepartmentID = 30)
    THEN
        UPDATE Departments SET Location = 'Boston' WHERE
DepartmentID = 30;
    END IF;
END;
```

- **Object-Oriented Extensions:** Support object-oriented features like inheritance and polymorphism. Example: SQL:1999 Object Extensions.

2.4 Entity-Relationship Model

The **Entity-Relationship (E-R) Model**, introduced by Peter Chen in 1976, is a high-level conceptual data model used to design and visualize database structures. It provides a graphical representation of entities, their attributes, and relationships.

Components of E-R Model

1. **Entities:**
    a. **Definition:** Objects or concepts that can be distinctly identified.
    b. **Types:**
        i. **Strong Entities:** Independent entities with unique identifiers (e.g., Employee, Customer).
        ii. **Weak Entities:** Dependent entities lacking unique identifiers and relying on strong entities (e.g., OrderItem).
2. **Attributes:**
    a. **Definition:** Properties or characteristics of entities.
    b. **Types:**
        i. **Simple Attributes:** Indivisible properties (e.g., FirstName).
        ii. **Composite Attributes:** Can be subdivided into smaller components (e.g., Address).
        iii. **Derived Attributes:** Obtained from other attributes (e.g., Age from DateOfBirth).
        iv. **Multi-valued Attributes:** Can hold multiple values (e.g., PhoneNumbers).
3. **Relationships:**
    a. **Definition:** Associations between entities.
    b. **Types:**
        i. **One-to-One (1:1):** Each entity in A relates to at most one entity in B, and vice versa.
        ii. **One-to-Many (1:N):** Each entity in A can relate to multiple entities in B, but each entity in B relates to only one in A.
        iii. **Many-to-Many (M:N):** Entities in A can relate to multiple entities in B and vice versa.

4. **Cardinality:**
    a. **Definition:** Specifies the numerical relationships between entities.
    b. **Notation:**
        i. **Crow's Foot Notation:** Graphical representation indicating cardinality (e.g., one, many).
5. **Keys:**
    a. **Primary Keys:** Unique identifiers for entities.
    b. **Foreign Keys:** Attributes that reference primary keys in other entities to establish relationships.

E-R Diagram Symbols

- **Rectangle:** Represents an entity.
- **Ellipse:** Represents an attribute.
- **Diamond:** Represents a relationship.
- **Lines:** Connect entities to their attributes and to relationships.

Real-World Example: University Database

- **Entities:**
    - **Student:** StudentID, Name, Major.
    - **Course:** CourseID, Title, Credits.
    - **Instructor:** InstructorID, Name, Department.
- **Relationships:**
    - **Enrolls:** Students enroll in Courses.
        - **Cardinality:** Many-to-Many (Students can enroll in multiple Courses, Courses can have multiple Students).
    - **Teaches:** Instructors teach Courses.
        - **Cardinality:** One-to-Many (An Instructor can teach multiple Courses, each Course is taught by one Instructor).

Diagram 2.4: Sample E-R Diagram

Figure 2.4: University E-R Diagram

Explanation:

- **Entities:** Represented by rectangles.
- **Attributes:** Represented by ellipses connected to respective entities.
- **Relationships:** Represented by diamonds connecting entities.
- **Cardinality:** Indicated by notation at the ends of relationship lines.

Conversion of E-R Diagram to Relational Model

1. **Entities to Tables:**
    a. Each entity becomes a table.
    b. Attributes become columns.
2. **Primary Keys:**
    a. Assign primary keys to each table based on unique identifiers.
3. **Relationships:**
    a. **One-to-One:** Implement by adding a foreign key to one of the tables.
    b. **One-to-Many:** Add a foreign key to the 'many' side table referencing the 'one' side.
    c. **Many-to-Many:** Create an associative table with foreign keys referencing both entities.
4. **Example Conversion:**

- **Student Table:**

```
CREATE TABLE Student (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100),
    Major VARCHAR(50)
);
```

- **Course Table:**

```
CREATE TABLE Course (
    CourseID INT PRIMARY KEY,
    Title VARCHAR(100),
    Credits INT
);
```

- **Instructor Table:**

```
CREATE TABLE Instructor (
    InstructorID INT PRIMARY KEY,
    Name VARCHAR(100),
    Department VARCHAR(50)
);
```

- **Enrolls (Associative Table):**

```
CREATE TABLE Enrolls (
    StudentID INT,
    CourseID INT,
    Grade CHAR(2),
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

- **Teaches Relationship:**
  - **Option 1:** Add `InstructorID` as a foreign key in the Course table.

```
ALTER TABLE Course
ADD InstructorID INT,
ADD FOREIGN KEY (InstructorID) REFERENCES Instructor(InstructorID);
```

  - **Option 2:** Create a separate Teaches table if a Course can be taught by multiple Instructors.

Unit III: Functional Dependencies and Normalization

3.1 Functional Dependencies

Functional Dependencies (FDs) are foundational to understanding normalization and ensuring data integrity within relational databases.

Definition

A **Functional Dependency (FD)** exists between two sets of attributes in a relation. For a relation R, a set of attributes X functionally determines another set of attributes Y if, for any valid instance of R, X uniquely determines Y. This is denoted as X → Y.

- **Formal Definition:**
  - For any two tuples t1 and t2 in R,
    - If t1[X] = t2[X], then t1[Y] = t2[Y].

Types of Functional Dependencies

1. **Trivial Functional Dependencies:**
   a. Occur when Y is a subset of X (e.g., {EmployeeID} → {EmployeeID}, {EmployeeID, Name} → {Name}).
2. **Non-Trivial Functional Dependencies:**
   a. Y is not a subset of X (e.g., {EmployeeID} → {Name}).
3. **Full Functional Dependency:**
   a. A non-trivial FD where removal of any attribute from X abolishes the dependency (e.g., {A, B} → C is a full FD if neither {A} → C nor {B} → C holds).
4. **Partial Functional Dependency:**
   a. Occurs when Y is functionally dependent on part of a composite key.
5. **Transitive Functional Dependency:**
   a. Occurs when X → Y and Y → Z, implying X → Z indirectly.

Armstrong's Axioms

Armstrong's Axioms are a set of inference rules used to derive all functional dependencies from a given set of FDs.

1. **Reflexivity:**
   a. If Y is a subset of X, then X → Y.
   b. Example:
      i. X = {A, B}, Y = {A}: {A, B} → {A}
2. **Augmentation:**

a. If X → Y, then XZ → YZ for any Z.
b. **Example:**
i. X = {A}, Y = {B}, Z = {C}: {A} → {B} implies {A, C} → {B, C}

3. **Transitivity:**
a. If X → Y and Y → Z, then X → Z.
b. **Example:**
i. X = {A}, Y = {B}, Z = {C}: {A} → {B} and {B} → {C} imply {A} → {C}

Closure of a Set of Attributes

- **Definition:**
  - The closure of a set of attributes X, denoted as X^+, is the set of attributes that are functionally determined by X.
- **Purpose:**
  - Used to determine candidate keys and to assess normalization.
- **Method to Compute Closure:**
  - Start with $X^+ = X$.
  - For each FD Y → Z, if Y is a subset of X^+, then add Z to X^+.
  - Repeat step 2 until no new attributes can be added.
- **Example:**
  - Given FDs:
    - {A} → {B}
    - {B} → {C}
    - {A} → {D}
  - Find A^+:
    - Start with {A}
    - {A} → {B}: Add {B} ⇒ {A, B}
    - {B} → {C}: Add {C} ⇒ {A, B, C}
    - {A} → {D}: Add {D} ⇒ {A, B, C, D}
    - No further additions: $A^+ = \{A, B, C, D\}$

Identifying Functional Dependencies

1. **Understanding Business Rules:**
   a. Derive FDs based on real-world constraints and business logic.
2. **Normalization Process:**
   a. Use FDs to identify anomalies and guide the normalization steps.
3. **Tools:**
   a. **Attribute Closure:** Determine which attributes are functionally dependent on others.
   b. **Inference Engine:** Apply Armstrong's Axioms to derive implied FDs.

Real-World Reference

Inventory Management System:

- **FDs:**
  - {ProductID} → {ProductName, SupplierID}
  - {SupplierID} → {SupplierName, SupplierContact}
- **Implications:**
  - Ensures that each ProductID uniquely identifies product details and supplier information.

- Maintains consistency in supplier details across different products.

3.2 Keys

Keys are critical for uniquely identifying tuples within a relation and for establishing relationships between tables.

Candidate Key

- **Definition:**
  - A candidate key is a minimal set of attributes that can uniquely identify any tuple in a relation.
- **Characteristics:**
  - **Minimality:** Removing any attribute from a candidate key causes it to lose its uniqueness property.
  - **Multiple Candidate Keys:** A relation can have multiple candidate keys.
- **Examples:**
  - **Students Table:**
    - Candidate Keys: {StudentID}, {Email}, {SSN}
- **Importance:**
  - Serves as potential primary keys.
  - Provides alternate access points to records.

Primary Key

- **Definition:**
  - A primary key is a candidate key selected by the database designer to uniquely identify each tuple in a table.
- **Characteristics:**
  - **Uniqueness and Non-Null:** Enforced by the DBMS.
  - **Single Key:** Each table can have only one primary key, though it can consist of multiple columns (composite key).
- **Example:**
  - **Employee Table:**
    - Primary Key: EmployeeID
- **Best Practices:**
  - Choose a surrogate key when natural keys are unstable or large.
  - Ensure the primary key is simple, minimal, and stable.

Superkey

- **Definition:**
  - A superkey is a set of one or more attributes that can uniquely identify a tuple in a relation. It can be a superset of a candidate key.
- **Characteristics:**
  - **Non-Minimality:** Superkeys may contain additional attributes beyond those required for uniqueness.
  - **Inclusion of Candidate Keys:** Every candidate key is a superkey, but not vice versa.
- **Examples:**

    - **Students Table:**
        - Superkeys: {StudentID, Email}, {StudentID, SSN}, {Email, SSN}, etc.
- **Usage:**
    - **Identifying Primary Keys:** Narrowing down to candidate keys among superkeys.
    - **Ensuring Data Uniqueness:** Through various attribute combinations.

Example: Student Table with Keys

StudentID	Email	SSN	Name
1001	alice@example.com	123-45-6789	Alice Johnson
1002	bob@example.com	987-65-4321	Bob Smith
1003	charlie@example.com	555-55-5555	Charlie Brown

- **Candidate Keys:**
    - {StudentID}, {Email}, {SSN}
- **Superkeys:**
    - {StudentID, Email}, {StudentID, SSN}, {Email, SSN}, {StudentID, Email, SSN}
- **Primary Key Selection:**
    - Choose {StudentID} as the primary key for simplicity and stability.

Diagram 3.2: Keys in a Relational Table

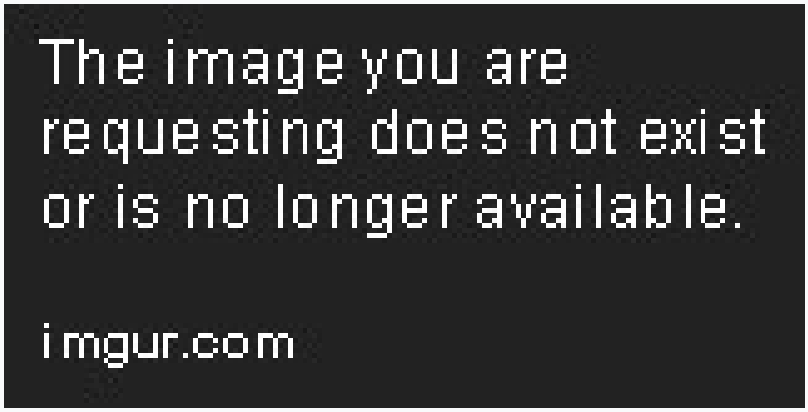

Figure 3.2: Primary, Candidate, and Foreign Keys

Explanation:

- **Primary Key:** Highlighted uniquely in the table.
- **Candidate Keys:** Indicated as potential identifiers.
- **Foreign Keys:** Shown linking to primary keys in other tables.

Real-World Reference

Library Management System:

- **Books Table:**
    - **Primary Key:** ISBN.
    - **Candidate Keys:** {ISBN}, {LibraryCode, CopyNumber}.
    - **Superkeys:** {ISBN, Title}, {ISBN, Author}.
- **Members Table:**
    - **Primary Key:** MemberID.
    - **Unique Constraint:** Email.

3.3 Normalization

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves decomposing tables into smaller, well-structured tables without losing data or creating anomalies.

Objectives of Normalization

1. **Eliminate Data Redundancy:**
    a. Prevents unnecessary duplication of data across tables.
2. **Ensure Data Integrity:**
    a. Maintains accuracy and consistency of data through enforced relationships.
3. **Simplify Data Structures:**
    a. Enhances data organization, making it easier to manage and query.
4. **Facilitate Efficient Data Access:**
    a. Improves query performance by structuring data logically.

Normal Forms

Normalization progresses through a series of normal forms (NF), each addressing specific types of anomalies. The most common normal forms up to Boyce-Codd Normal Form (BCNF) are discussed below.

First Normal Form (1NF)

- **Criteria:**
    - **Atomicity:** Each table cell must contain only one value (i.e., atomic values).
    - **Uniqueness:** Each record must be unique.
- **Purpose:**
    - Eliminates repeating groups and arrays within records.
- **Example Violation:**

EmployeeID	Name	Skills
1	John	Java, SQL
2	Jane	Python, Java

- **Normalized to 1NF:**

EmployeeID	Name	Skill
1	John	Java
1	John	SQL
2	Jane	Python
2	Jane	Java

Second Normal Form (2NF)

- **Criteria:**
    - **Already in 1NF.**

    - **No Partial Dependencies:** No non-prime attribute depends on a part of a composite primary key.
- **Purpose:**
    - Eliminates partial dependencies, ensuring that non-key attributes fully depend on the entire primary key.
- **Example Violation:**

OrderID	ProductID	ProductName	Quantity
1001	P01	Widget	10
1002	P02	Gizmo	5

- **Normalization to 2NF:**
    - **Orders Table:**

OrderID	ProductID	Quantity
1001	P01	10
1002	P02	5

    - **Products Table:**

ProductID	ProductName
P01	Widget
P02	Gizmo

Third Normal Form (3NF)

- **Criteria:**
    - **Already in 2NF.**
    - **No Transitive Dependencies:** Non-prime attributes do not depend on other non-prime attributes.
- **Purpose:**
    - Eliminates transitive dependencies to ensure that non-key attributes are only dependent on primary keys.
- **Example Violation:**

StudentID	Major	DepartmentHead
1	CS	Dr. Smith
2	EE	Dr. Jones

- **Normalization to 3NF:**
    - **Students Table:**

StudentID	Major
1	CS

```
| 2         | EE    |
```

    - **Departments Table:**

```
| Major | DepartmentHead |
|-------|----------------|
| CS    | Dr. Smith      |
| EE    | Dr. Jones      |
```

Boyce-Codd Normal Form (BCNF)

- **Criteria:**
    - A stronger version of 3NF.
    - For every non-trivial functional dependency X → Y, X must be a superkey.
- **Purpose:**
    - Addresses anomalies that 3NF does not resolve by ensuring that every determinant is a candidate key.
- **Example Violation:**

```
| CourseID | Instructor | Room  |
|----------|------------|-------|
| CS101    | Dr. Adams  | R1    |
| CS101    | Dr. Adams  | R2    |
| CS102    | Dr. Baker  | R1    |
```

- **Normalization to BCNF:**
    - **Course-Instructor Table:**

```
| CourseID | Instructor |
|----------|------------|
| CS101    | Dr. Adams  |
| CS102    | Dr. Baker  |
```

    - **Course-Room Table:**

```
| CourseID | Room |
|----------|------|
| CS101    | R1   |
| CS101    | R2   |
| CS102    | R1   |
```

Benefits of Normalization

1. **Eliminates Redundancy:** Reduces data duplication, saving storage space.
2. **Prevents Anomalies:** Avoids update, insertion, and deletion anomalies.
3. **Enhances Data Integrity:** Ensures that data dependencies are logical and enforceable.
4. **Improves Query Performance:** Structured data can lead to more efficient indexing and querying.

5. **Simplifies Maintenance:** Well-normalized databases are easier to modify and extend.

Drawbacks of Over-Normalization

1. **Increased Number of Tables:** Can lead to more complex queries involving multiple joins.
2. **Potential Performance Overhead:** Joins across many tables may slow down query performance.
3. **Complex Database Design:** Requires careful planning and understanding of relationships.

Real-World Reference

Sales Management System:

- **Issues Before Normalization:**
  - **Data Redundancy:** Repeats customer and product information in multiple sales records.
  - **Update Anomalies:** Changing a customer's address requires updates across all related sales records.
  - **Insertion Anomalies:** Inability to add a new customer without an existing sale.
- **Normalization Process:**
  - **1NF:** Ensure atomic values.
  - **2NF:** Separate customer and product details from sales records.
  - **3NF:** Remove transitive dependencies by isolating supplier information from products.
- **Normalized Tables:**
  - **Customers:** CustomerID, Name, Address.
  - **Products:** ProductID, ProductName, SupplierID.
  - **Sales:** SaleID, CustomerID, ProductID, Quantity, SaleDate.
  - **Suppliers:** SupplierID, SupplierName, ContactInfo.

3.4 Canonical Cover

A **Canonical Cover** is a minimal set of functional dependencies (FDs) that preserves the original dependencies and has no extraneous attributes. It simplifies the set of FDs, making it easier to work with during normalization.

Importance of Canonical Cover

- **Simplification:** Reduces the complexity of functional dependencies.
- **Normalization Process:** Facilitates the identification of candidate keys and normalization steps.
- **Elimination of Redundancies:** Removes unnecessary dependencies and attributes.

Steps to Find Canonical Cover

1. **Ensure Right-Hand Side (RHS) is a Single Attribute:**

a. Decompose functional dependencies so that each FD has only one attribute on the RHS.

2. **Remove Extraneous Attributes in the Left-Hand Side (LHS):**
   a. Identify and eliminate attributes that do not contribute to the dependency.
   b. An attribute is extraneous if its removal does not affect the closure.
3. **Eliminate Redundant Functional Dependencies:**
   a. Remove any FD that can be derived from others in the set.

Detailed Procedure

1. **Decompose FDs with Multiple Attributes on RHS:**
   a. Split each FD with multiple attributes on the RHS into multiple FDs with single attributes.
   b. Example:
      i. Original FD: A → B, C
      ii. Decomposed FDs: A → B and A → C
2. **Remove Extraneous Attributes from LHS:**
   a. For each FD X → Y, check if a subset of X can still determine Y through the remaining FDs.
   b. If so, remove the extraneous attribute.
   c. Example:
      i. FD: A, B → C
      ii. If {A} → C already holds, then B is extraneous and can be removed.
3. **Eliminate Redundant FDs:**
   a. For each FD X → Y, temporarily remove it from the set.
   b. Compute the closure using the remaining FDs.
   c. If Y is in the closure of X, the FD is redundant and can be eliminated.
   d. Example:
      i. Original FDs: A → B, B → C, A → C
      ii. Check if A → C is redundant by computing A^+ without A → C.
      iii. Since A → B and B → C, A → C can be derived, making it redundant.

Example: Finding Canonical Cover

Given FDs:

1. A → B
2. A → C, D
3. D → E

Step 1: Decompose FDs with Multiple Attributes on RHS

-
  - A → B
-
  - A → C
-
  - A → D
-
  - D → E

Step 2: Remove Extraneous Attributes from LHS

- All FDs have single attributes on LHS; no removal needed.

Step 3: Eliminate Redundant FDs

- Check if any FD can be derived from others.
- No redundancies found.

Canonical Cover:

- $A \rightarrow B$
- $A \rightarrow C$
- $A \rightarrow D$
- $D \rightarrow E$

Unit IV: Structured Query Language (SQL)

SQL is the standard language used to interact with relational databases. It encompasses various components, each serving a specific purpose in database management.

4.1 SQL Basics

Data Types

Data types define the nature of data that can be stored in each column of a table. Proper selection of data types is crucial for data integrity and performance.

1. **Numeric Types:**
    a. **INT:** Integer values.
    b. **DECIMAL(p, s):** Fixed-point numbers with precision p and scale s.
    c. **FLOAT, DOUBLE:** Approximate floating-point numbers.
2. **Character Types:**
    a. **CHAR(n):** Fixed-length strings up to n characters.
    b. **VARCHAR(n):** Variable-length strings up to n characters.
    c. **TEXT:** Large variable-length strings.
3. **Date/Time Types:**
    a. **DATE:** Calendar date (YYYY-MM-DD).
    b. **TIME:** Time of day (HH:MM:SS).
    c. **DATETIME:** Combination of date and time.
    d. **TIMESTAMP:** Automatic timestamping of records.
4. **Boolean Type:**
    a. **BOOLEAN:** Represents TRUE or FALSE values.
5. **Binary Types:**
    a. **BLOB:** Binary Large Objects for storing multimedia data.
6. **Enumerated Types:**
    a. **ENUM:** Predefined list of values.
7. **Other Types:**
    a. **UUID:** Universally Unique Identifier.
    b. **JSON:** Storing JSON-formatted data.

Commands Overview

1. **DDL (Data Definition Language):**
    a. **CREATE, ALTER, DROP**
2. **DML (Data Manipulation Language):**
    a. **SELECT, INSERT, UPDATE, DELETE**
3. **DCL (Data Control Language):**
    a. **GRANT, REVOKE**
4. **TCL (Transaction Control Language):**
    a. **COMMIT, ROLLBACK, SAVEPOINT**

Syntax Essentials

- **SQL Statements:** Generally begin with a command keyword (e.g., SELECT, INSERT) and end with a semicolon (`;`).
- **Case Insensitivity:** SQL keywords are case-insensitive, though maintaining consistent casing improves readability.
- **Comments:**
    - Single-line: `-- This is a comment`
    - Multi-line: `/* This is a multi-line comment */`

Example SQL Statement

Scenario: Retrieve First and Last Names of Employees in Department 10.

```
SELECT FirstName, LastName
FROM Employees
WHERE DepartmentID = 10;
```

- **Explanation:**
    - **SELECT:** Specifies columns to retrieve.
    - **FROM:** Specifies the table to query.
    - **WHERE:** Filters records based on DepartmentID.

SQL Statement Structure

1. **Clauses:** Components like SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY.
2. **Expressions:** Define what to retrieve, such as column names, calculations.
3. **Operators:** Include arithmetic, comparison, and logical operators to define conditions.
4. **Functions:** Perform operations on data, such as aggregations (SUM, AVG).

Best Practices

- **Use Aliases:** Simplifies queries, especially with multiple tables.

```
SELECT e.FirstName, e.LastName, d.DepartmentName
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

- **Consistent Naming Conventions:** Improves readability and maintenance.
- **Avoid Using SELECT :* Specify required columns to enhance performance.

- **Proper Indentation and Formatting:** Makes queries easier to read and debug.
- **Use Comments:** Document complex queries for future reference.

4.2 DDL Commands

Data Definition Language (DDL) commands manage the structure and schema of databases, including creating, modifying, and deleting database objects.

CREATE Command

- **Purpose:** Define new database objects like tables, views, indexes, and schemas.
- **Syntax:**

```
CREATE [OBJECT] object_name (
    column1 datatype constraints,
    column2 datatype constraints,
    ...
);
```

- **Examples:**
    - **Create Table:**

```
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50) NOT NULL,
    Location VARCHAR(100)
);
```

    - **Create View:**

```
CREATE VIEW EmployeeDetails AS
SELECT Employees.FirstName, Employees.LastName, Departments.DepartmentName
FROM Employees
JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

    - **Create Index:**

```
CREATE INDEX idx_employee_email
ON Employees (Email);
```

ALTER Command

- **Purpose:** Modify the structure of existing database objects.
- **Syntax:**

```
ALTER [OBJECT] object_name
ADD|DROP|MODIFY column_definition;
```

- **Examples:**
    - **Add Column:**

```
ALTER TABLE Employees
ADD Email VARCHAR(100);
```

- **Drop Column:**

```
ALTER TABLE Employees
DROP COLUMN Email;
```

- **Modify Column Data Type:**

```
ALTER TABLE Employees
MODIFY Salary DECIMAL(12, 2);
```

- **Add Constraint:**

```
ALTER TABLE Employees
ADD CONSTRAINT chk_salary CHECK (Salary > 0);
```

DROP Command

- **Purpose:** Remove existing database objects permanently.
- **Syntax:**

```
DROP [OBJECT] object_name;
```

- **Examples:**
  - **Drop Table:**

```
DROP TABLE Departments;
```

  - **Drop View:**

```
DROP VIEW EmployeeDetails;
```

  - **Drop Index:**

```
DROP INDEX idx_employee_email;
```

TRUNCATE Command

- **Purpose:** Remove all records from a table without deleting the table structure itself.
- **Characteristics:**
  - **Faster than DELETE:** Because it doesn't generate individual row delete statements and doesn't log individual row deletions.
  - **Cannot be Rolled Back:** In some DBMSs, unlike DELETE.
  - **Resets Identity Counters:** Resets any auto-incremented values back to the starting point.
- **Syntax:**

```
TRUNCATE TABLE table_name;
```

- **Example:**

```
TRUNCATE TABLE Employees;
```

RELOCATE and other Advanced DDL Commands

- **Advanced DDL Commands:** Depending on the DBMS, additional DDL commands may be available to modify storage parameters, partition tables, etc.
- **Example (Partitioning a Table in PostgreSQL):**

```
CREATE TABLE Sales (
    SaleID INT PRIMARY KEY,
    SaleDate DATE,
    Amount DECIMAL(10,2)
) PARTITION BY RANGE (SaleDate);
```

Real-World Reference

Content Management System (CMS):

- **Creating Tables:**
    - **Articles Table:**

```
CREATE TABLE Articles (
    ArticleID INT PRIMARY KEY,
    Title VARCHAR(200) NOT NULL,
    Content TEXT,
    AuthorID INT,
    PublishDate DATE,
    FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID)
);
```

- **Modifying Schema:**
    - **Add CategoryID to Articles:**

```
ALTER TABLE Articles
ADD CategoryID INT,
ADD FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID);
```

- **Dropping Unused Tables:**

```
DROP TABLE OldArticles;
```

4.3 DML Commands

Data Manipulation Language (DML) commands are used to retrieve, insert, update, and delete data within database tables. They play a vital role in day-to-day database operations.

INSERT Command

- **Purpose:** Add new records to a table.
- **Syntax:**

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

- **With SELECT:**

```
INSERT INTO table_name (column1, column2, ...)
SELECT column1, column2, ...
FROM another_table
WHERE condition;
```

- **Examples:**

```
-- Insert a single record
INSERT INTO Employees (EmployeeID, FirstName, LastName,
DepartmentID, Salary)
VALUES (1, 'John', 'Doe', 10, 60000.00);

-- Insert multiple records
INSERT INTO Employees (EmployeeID, FirstName, LastName,
DepartmentID, Salary)
VALUES
    (2, 'Jane', 'Smith', 20, 80000.00),
    (3, 'Bob', 'Johnson', 10, 55000.00);

-- Insert using SELECT
INSERT INTO Employees (EmployeeID, FirstName, LastName,
DepartmentID, Salary)
SELECT EmployeeID, FirstName, LastName, DepartmentID, Salary
FROM TempEmployees
WHERE Status = 'Active';
```

SELECT Command

- **Purpose:** Retrieve data from one or more tables.
- **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
GROUP BY column
HAVING condition
ORDER BY column [ASC|DESC];
```

- **Using SELECT :*

```
SELECT * FROM Employees;
```

- **Examples:**

```
-- Select specific columns
SELECT FirstName, LastName, Email
FROM Employees
WHERE DepartmentID = 10;

-- Select all columns
SELECT * FROM Departments;

-- Select with ordering
SELECT FirstName, LastName, Salary
FROM Employees
ORDER BY Salary DESC;

-- Select with group by and having
SELECT DepartmentID, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY DepartmentID
HAVING COUNT(*) > 5;
```

UPDATE Command

- **Purpose:** Modify existing records in a table.
- **Syntax:**

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- **Examples:**

```
-- Update a single record
UPDATE Employees
SET Salary = 65000.00
WHERE EmployeeID = 1;

-- Update multiple records
UPDATE Employees
SET DepartmentID = 30
WHERE DepartmentID = 20;

-- Update with calculations
UPDATE Employees
SET Salary = Salary * 1.05
WHERE DepartmentID = 10;
```

DELETE Command

- **Purpose:** Remove records from a table.
- **Syntax:**

```
DELETE FROM table_name
WHERE condition;
```

  - **Delete All Records:**

```
DELETE FROM table_name;
```

- **Examples:**

```
-- Delete a single record
DELETE FROM Employees
WHERE EmployeeID = 1;

-- Delete multiple records
DELETE FROM Employees
WHERE DepartmentID = 30;

-- Delete all records (use with caution)
DELETE FROM Employees;
```

MERGE Command

- **Purpose:** Combine INSERT, UPDATE, and DELETE operations based on a condition, commonly used for upsert operations.
- **Syntax:**

```
MERGE INTO target_table USING source_table
ON (condition)
WHEN MATCHED THEN
    UPDATE SET column1 = value1, column2 = value2, ...
WHEN NOT MATCHED THEN
    INSERT (column1, column2, ...) VALUES (value1, value2, ...);
```

- **Example:**

```
MERGE INTO Employees AS target
USING NewEmployees AS source
ON (target.EmployeeID = source.EmployeeID)
WHEN MATCHED THEN
    UPDATE SET target.Salary = source.Salary
WHEN NOT MATCHED THEN
    INSERT (EmployeeID, FirstName, LastName, DepartmentID, Salary)
    VALUES (source.EmployeeID, source.FirstName, source.LastName,
```

```
source.DepartmentID, source.Salary);
```

Real-World Reference

Customer Support System:

- **Inserting Support Tickets:**

```
INSERT INTO SupportTickets (TicketID, CustomerID, Issue, Status,
CreatedDate)
VALUES (5001, 1001, 'Cannot access account', 'Open', GETDATE());
```

- **Updating Ticket Status:**

```
UPDATE SupportTickets
SET Status = 'Resolved'
WHERE TicketID = 5001;
```

- **Deleting Obsolete Tickets:**

```
DELETE FROM SupportTickets
WHERE TicketID = 4000;
```

- **Selecting Active Tickets:**

```
SELECT TicketID, CustomerID, Issue
FROM SupportTickets
WHERE Status = 'Open';
```

4.4 SQL Operators

SQL operators enable the construction of complex conditions and expressions within SQL statements, enhancing the flexibility and precision of data manipulation and retrieval.

Arithmetic Operators

- **Purpose:** Perform mathematical calculations on numeric data.
- **Common Operators:**
  - `+` : Addition
  - `-` : Subtraction
  - `*` : Multiplication
  - `/` : Division
  - `%` : Modulus (remainder)
- **Examples:**

```
-- Calculate bonus as 10% of salary
SELECT FirstName, LastName, Salary, Salary * 0.1 AS Bonus
FROM Employees;

-- Compute total cost
```

```
SELECT OrderID, Quantity, UnitPrice, Quantity * UnitPrice AS
TotalCost
FROM OrderDetails;
```

Comparison Operators

- **Purpose:** Compare values to filter query results based on specific conditions.
- **Common Operators:**
  - = : Equal to
  - != or <> : Not equal to
  - < : Less than
  - > : Greater than
  - <= : Less than or equal to
  - >= : Greater than or equal to
- **Examples:**

```
-- Select employees with salary greater than 50,000
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 50000;

-- Select departments not located in 'New York'
SELECT DepartmentName
FROM Departments
WHERE Location <> 'New York';
```

Logical Operators

- **Purpose:** Combine multiple conditions in SQL statements to form complex queries.
- **Common Operators:**
  - AND : Both conditions must be true.
  - OR : At least one condition must be true.
  - NOT : Negates a condition.
- **Examples:**

```
-- Select employees in Department 10 earning over 60,000
SELECT FirstName, LastName
FROM Employees
WHERE DepartmentID = 10 AND Salary > 60000;

-- Select employees in Department 10 or 20
SELECT FirstName, LastName
FROM Employees
WHERE DepartmentID = 10 OR DepartmentID = 20;
```

```
-- Select employees not in Department 30
SELECT FirstName, LastName
FROM Employees
WHERE NOT DepartmentID = 30;
```

SET Operators

- **Purpose:** Combine the results of two or more SELECT statements.
- **Common Set Operators:**
  - `UNION`: Combines results and removes duplicates.
  - `UNION ALL`: Combines results and includes duplicates.
  - `INTERSECT`: Returns common records between two queries.
  - `EXCEPT` or `MINUS`: Returns records from the first query not present in the second.
- **Examples:**

```
-- Combine unique employee IDs from two departments
SELECT EmployeeID FROM Employees WHERE DepartmentID = 10
UNION
SELECT EmployeeID FROM Employees WHERE DepartmentID = 20;

-- Combine all employee IDs, including duplicates
SELECT EmployeeID FROM Employees WHERE DepartmentID = 10
UNION ALL
SELECT EmployeeID FROM Employees WHERE DepartmentID = 20;

-- Find common employee IDs in both departments
SELECT EmployeeID FROM Employees WHERE DepartmentID = 10
INTERSECT
SELECT EmployeeID FROM Employees WHERE DepartmentID = 20;

-- Find employee IDs in Department 10 not in Department 20
SELECT EmployeeID FROM Employees WHERE DepartmentID = 10
EXCEPT
SELECT EmployeeID FROM Employees WHERE DepartmentID = 20;
```

Concatenation Operator

- **Purpose:** Combine two or more strings into a single string.
- **Syntax Variations:**
  - In SQL Server: `+`
  - In MySQL: `CONCAT()`
  - In Oracle: `||`
- **Examples:**

```
-- SQL Server
SELECT FirstName + ' ' + LastName AS FullName
```

```
FROM Employees;

-- MySQL
SELECT CONCAT(FirstName, ' ', LastName) AS FullName
FROM Employees;

-- Oracle
SELECT FirstName || ' ' || LastName AS FullName
FROM Employees;
```

Example Usage in Context

Scenario: Retrieve Full Names and Total Salaries of Employees in Department 10 earning over 50,000.

```
SELECT
    CONCAT(FirstName, ' ', LastName) AS FullName,
    Salary + (Salary * 0.1) AS TotalSalary
FROM Employees
WHERE DepartmentID = 10 AND Salary > 50000;
```

- **Explanation:**
  - **CONCAT():** Combines FirstName and LastName.
  - **Arithmetic Operator (+):** Calculates total salary with a 10% bonus.
  - **WHERE Clause:** Filters based on DepartmentID and Salary.

4.5 SQL Functions

SQL functions perform operations on data and return results. They can be categorized into several types, each serving unique purposes.

Date Functions

- **Purpose:** Handle and manipulate date and time data.
- **Common Functions:**
  - **GETDATE() / CURRENT_DATE:**
    - **Description:** Returns the current date and time.
    - **Example:**

```
SELECT FirstName, LastName, GETDATE() AS Today
FROM Employees;
```

  - **DATEADD():**
    - **Description:** Adds a specified interval to a date.
    - **Syntax:** `DATEADD(interval, number, date)`
    - **Example:**

```
-- Add 30 days to HireDate
SELECT FirstName, LastName, HireDate, DATEADD(day, 30, HireDate) AS ReviewDate
```

```
FROM Employees;
```

    - **DATEDIFF():**
        - **Description:** Returns the difference between two dates.
        - **Syntax:** `DATEDIFF(interval, start_date, end_date)`
        - **Example:**

```
-- Calculate tenure in years
SELECT FirstName, LastName, DATEDIFF(year, HireDate, GETDATE()) AS Tenure
FROM Employees;
```

    - **DATEPART():**
        - **Description:** Extracts a specific part of a date (e.g., year, month, day).
        - **Syntax:** `DATEPART(interval, date)`
        - **Example:**

```
-- Extract the month of HireDate
SELECT FirstName, LastName, DATEPART(month, HireDate) AS HireMonth
FROM Employees;
```

    - **CONVERT() / CAST():**
        - **Description:** Converts one data type to another.
        - **Syntax:** `CONVERT(data_type, expression)` or `CAST(expression AS data_type)`
        - **Example:**

```
-- Convert HireDate to VARCHAR
SELECT FirstName, LastName, CAST(HireDate AS VARCHAR(10)) AS HireDateString
FROM Employees;
```

Numeric Functions

- **Purpose:** Perform mathematical operations on numeric data.
- **Common Functions:**
    - **ROUND():**
        - **Description:** Rounds a number to a specified number of decimal places.
        - **Syntax:** `ROUND(number, decimals)`
        - **Example:**

```
SELECT Salary, ROUND(Salary * 1.05, 2) AS NewSalary
FROM Employees;
```

    - **CEILING():**
        - **Description:** Rounds a number up to the nearest integer.
        - **Example:**

```
SELECT Salary, CEILING(Salary / 1000.00) AS SalaryRounded
FROM Employees;
```

    - **FLOOR():**
        - **Description:** Rounds a number down to the nearest integer.
        - **Example:**

```
SELECT Salary, FLOOR(Salary / 1000.00) AS SalaryFloored
FROM Employees;
```

    - **ABS():**

        - **Description:** Returns the absolute value of a number.
        - **Example:**

```
SELECT Salary, ABS(Salary - 50000) AS SalaryDifference
FROM Employees;
```

    - **POWER():**
        - **Description:** Raises a number to the power of another number.
        - **Syntax:** `POWER(base, exponent)`
        - **Example:**

```
SELECT Salary, POWER(Salary, 2) AS SalarySquared
FROM Employees;
```

Character Functions

- **Purpose:** Manipulate and analyze string data.
- **Common Functions:**
    - **UPPER():**
        - **Description:** Converts a string to uppercase.
        - **Example:**

```
SELECT FirstName, LastName, UPPER(LastName) AS LastNameUpper
FROM Employees;
```

    - **LOWER():**
        - **Description:** Converts a string to lowercase.
        - **Example:**

```
SELECT FirstName, LastName, LOWER(FirstName) AS FirstNameLower
FROM Employees;
```

    - **CONCAT():**
        - **Description:** Concatenates two or more strings.
        - **Example:**

```
SELECT FirstName, LastName, CONCAT(FirstName, ' ', LastName) AS FullName
FROM Employees;
```

    - **SUBSTRING():**
        - **Description:** Extracts a substring from a string.
        - **Syntax:** `SUBSTRING(string, start, length)`
        - **Example:**

```
SELECT FirstName, LastName, SUBSTRING(FirstName, 1, 1) AS Initial
FROM Employees;
```

    - **LEN() / LENGTH():**
        - **Description:** Returns the length of a string.
        - **Example:**

```
SELECT FirstName, LEN(FirstName) AS NameLength
FROM Employees;
```

Conversion Functions

- **Purpose:** Convert data from one type to another to facilitate operations and ensure compatibility.

- **Common Functions:**
    - **CAST():**
        - **Description:** Converts an expression from one data type to another.
        - **Syntax:** `CAST(expression AS data_type)`
        - **Example:**

```
SELECT FirstName, LastName, CAST(Salary AS VARCHAR(10)) AS SalaryText
FROM Employees;
```

    - **CONVERT():**
        - **Description:** Similar to CAST, but with additional formatting options (DBMS-specific).
        - **Syntax:** `CONVERT(data_type, expression, style)`
        - **Example (SQL Server):**

```
SELECT FirstName, LastName, CONVERT(VARCHAR(10), HireDate, 101) AS HireDateFormatted
FROM Employees;
```

    - **TO_CHAR(), TO_NUMBER():**
        - **Description:** Convert data types in Oracle.
        - **Example (Oracle):**

```
SELECT TO_CHAR(HireDate, 'MM/DD/YYYY') AS HireDateFormatted
FROM Employees;
```

Aggregate Functions

- **Purpose:** Perform calculations on a set of values to return a single summary value.
- **Common Functions:**
    - **COUNT():**
        - **Description:** Counts the number of rows or non-NULL values.
        - **Example:**

```
SELECT COUNT(*) AS TotalEmployees
FROM Employees;
```

    - **SUM():**
        - **Description:** Calculates the total sum of a numeric column.
        - **Example:**

```
SELECT DepartmentID, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY DepartmentID;
```

    - **AVG():**
        - **Description:** Calculates the average value of a numeric column.
        - **Example:**

```
SELECT DepartmentID, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY DepartmentID;
```

    - **MIN() & MAX():**
        - **Description:** Retrieves the minimum and maximum values.
        - **Example:**

```
SELECT DepartmentID, MIN(Salary) AS LowestSalary, MAX(Salary) AS HighestSalary
FROM Employees
```

```
GROUP BY DepartmentID;
```

Miscellaneous Functions

- **Purpose:** Provide additional functionalities not covered by the above categories.
- **Common Functions:**
    - **COALESCE():**
        - **Description:** Returns the first non-NULL expression among its arguments.
        - **Example:**

```
SELECT FirstName, LastName, COALESCE(Email, 'No Email') AS Contact
FROM Employees;
```

    - **NULLIF():**
        - **Description:** Returns NULL if the two expressions are equal; otherwise, returns the first expression.
        - **Example:**

```
SELECT FirstName, LastName, NULLIF(MiddleName, '') AS MiddleName
FROM Employees;
```

    - **CASE():**
        - **Description:** Implements conditional logic within queries.
        - **Syntax:**

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE resultN
END
```

        - **Example:**

```
SELECT FirstName, LastName,
    CASE
        WHEN Salary > 70000 THEN 'High'
        WHEN Salary BETWEEN 50000 AND 70000 THEN 'Medium'
        ELSE 'Low'
    END AS SalaryGrade
FROM Employees;
```

Real-World Reference

Retail Sales Analysis:

- **Calculating Total Sales and Average Sales per Product:**

```
SELECT ProductID, SUM(SaleAmount) AS TotalSales, AVG(SaleAmount) AS
AverageSales
FROM Sales
GROUP BY ProductID;
```

- **Determining Sales Performance:**

```
SELECT ProductID,
    CASE
```

```
        WHEN SUM(SaleAmount) > 100000 THEN 'Top Performer'
        WHEN SUM(SaleAmount) BETWEEN 50000 AND 100000 THEN 'Average
Performer'
        ELSE 'Low Performer'
    END AS PerformanceCategory
FROM Sales
GROUP BY ProductID;
```

- **Handling Missing Data in Customer Emails:**

```
SELECT CustomerID, FirstName, LastName, COALESCE(Email, 'No Email
Provided') AS ContactEmail
FROM Customers;
```

4.6 Data Organization

Data Organization in SQL involves structuring query results for better readability, analysis, and reporting. Key clauses that facilitate data organization include **GROUP BY**, **HAVING**, and **ORDER BY**.

GROUP BY Clause

- **Purpose:** Groups rows that have the same values in specified columns into summary rows.
- **Usage:** Commonly used with aggregate functions to perform calculations on each group.
- **Syntax:**

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
WHERE condition
GROUP BY column1, column2;
```

- **Example:**

```
-- Calculate the total salary per department
SELECT DepartmentID, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY DepartmentID;
```

- **Note:**
    - All non-aggregated columns in the SELECT statement must be included in the GROUP BY clause.

HAVING Clause

- **Purpose:** Filters groups based on a condition, similar to how the WHERE clause filters individual rows.

- **Usage:** Used in conjunction with GROUP BY and aggregate functions.
- **Syntax:**

```
SELECT column1, aggregate_function(column2)
FROM table_name
WHERE condition
GROUP BY column1
HAVING aggregate_condition;
```

- **Example:**

```
-- Find departments with more than 5 employees
SELECT DepartmentID, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY DepartmentID
HAVING COUNT(*) > 5;
```

- **Explanation:**
  - **WHERE:** Filters rows before grouping.
  - **HAVING:** Filters groups after aggregation.

ORDER BY Clause

- **Purpose:** Sorts the result set based on specified columns in ascending or descending order.
- **Usage:** Enhances data readability and aids in trend analysis.
- **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

- **Examples:**

```
-- Sort employees by salary in descending order
SELECT FirstName, LastName, Salary
FROM Employees
ORDER BY Salary DESC;

-- Sort departments by name ascending and then by location
descending
SELECT DepartmentName, Location
FROM Departments
ORDER BY DepartmentName ASC, Location DESC;
```

Combining Clauses for Data Organization

Scenario: Retrieve the average salary per department, only for departments with an average salary above 60,000, sorted by average salary descending.

```
SELECT DepartmentID, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY DepartmentID
HAVING AVG(Salary) > 60000
ORDER BY AverageSalary DESC;
```

- **Explanation:**
  - **SELECT:** Retrieves DepartmentID and the average salary.
  - **FROM:** Specifies the Employees table.
  - **GROUP BY:** Groups records by DepartmentID.
  - **HAVING:** Filters groups where the average salary exceeds 60,000.
  - **ORDER BY:** Sorts the results in descending order of average salary.

Real-World Reference

Sales Reporting:

- **Generate Monthly Sales Reports:**

```
SELECT MONTH(SaleDate) AS SaleMonth, YEAR(SaleDate) AS SaleYear,
SUM(SaleAmount) AS TotalSales
FROM Sales
WHERE SaleDate >= '2023-01-01' AND SaleDate < '2024-01-01'
GROUP BY MONTH(SaleDate), YEAR(SaleDate)
HAVING SUM(SaleAmount) > 100000
ORDER BY SaleYear ASC, SaleMonth ASC;
```

- **Analyze Product Performance:**

```
SELECT ProductID, COUNT(*) AS NumberOfSales, AVG(SaleAmount) AS
AverageSale
FROM Sales
GROUP BY ProductID
HAVING COUNT(*) > 50
ORDER BY AverageSale DESC;
```

- **Customer Demographics:**

```
SELECT AgeGroup, COUNT(*) AS CustomerCount
FROM (
    SELECT
        CASE
            WHEN Age BETWEEN 18 AND 25 THEN '18-25'
            WHEN Age BETWEEN 26 AND 35 THEN '26-35'
            WHEN Age BETWEEN 36 AND 45 THEN '36-45'
```

```
            ELSE '46+'
        END AS AgeGroup
    FROM Customers
) AS AgeCategories
GROUP BY AgeGroup
ORDER BY CASE AgeGroup
             WHEN '18-25' THEN 1
             WHEN '26-35' THEN 2
             WHEN '36-45' THEN 3
             ELSE 4
         END;
```

Unit V: Database Integrity Constraints

Integrity constraints are rules applied to database tables to ensure data accuracy, consistency, and reliability. They serve as safeguards against invalid data entry and maintain the relationships between tables.

5.1 Domain Integrity

Domain Integrity ensures that all data within a column adheres to the defined data type, format, and allowed values.

NOT NULL Constraint

- **Purpose:** Ensures that a column cannot have NULL values, enforcing mandatory data entry.
- **Usage:** Applied to columns that require data to be present for each record.
- **Syntax:**

```
CREATE TABLE table_name (
    column1 datatype NOT NULL,
    column2 datatype,
    ...
);
```

  - **Alter Table to Add NOT NULL:**

```
ALTER TABLE Employees
MODIFY Email VARCHAR(100) NOT NULL;
```

- **Examples:**

```
-- Create table with NOT NULL constraints
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
```

```
    Email VARCHAR(100) NOT NULL
);

-- Insert record without Email (causes error)
INSERT INTO Employees (EmployeeID, FirstName, LastName)
VALUES (1, 'John', 'Doe'); -- Error: Email cannot be NULL
```

CHECK Constraint

- **Purpose:** Ensures that all values in a column satisfy a specific condition or criteria.
- **Usage:** Enforces business rules and data validity at the column level.
- **Syntax:**

```
CREATE TABLE table_name (
    column1 datatype CHECK (condition),
    column2 datatype,
    ...
);
```

  - **Add CHECK Constraint to Existing Table:**

```
ALTER TABLE Employees
ADD CONSTRAINT chk_salary CHECK (Salary > 0);
```

- **Examples:**

```
-- Create table with CHECK constraint
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Age INT,
    CHECK (Age >= 18)
);

-- Insert record with Age below 18 (causes error)
INSERT INTO Employees (EmployeeID, FirstName, LastName, Age)
VALUES (2, 'Jane', 'Smith', 16); -- Error: Age must be >= 18
```

- **Complex Conditions:**
  - Constraints can involve multiple columns and complex logic.

```
-- Ensure that EndDate is after StartDate
CREATE TABLE Projects (
    ProjectID INT PRIMARY KEY,
    ProjectName VARCHAR(100),
    StartDate DATE,
```

```
    EndDate DATE,
    CHECK (EndDate > StartDate)
);
```

Real-World Reference

Financial Systems:

- **Accounts Table:**

```
CREATE TABLE Accounts (
    AccountID INT PRIMARY KEY,
    AccountType VARCHAR(20) CHECK (AccountType IN ('Savings',
'Checking', 'Investment')),
    Balance DECIMAL(15, 2) CHECK (Balance >= 0)
);
```

- **Explanation:**
  - **AccountType:** Must be one of the predefined types.
  - **Balance:** Must be non-negative, preventing overdrafts.

5.2 Entity Integrity

Entity Integrity ensures that each entity in a table can be uniquely identified, typically achieved through primary keys.

PRIMARY KEY Constraint

- **Purpose:** Uniquely identifies each record in a table; enforces both uniqueness and NOT NULL constraints.
- **Characteristics:**
  - Only one primary key per table.
  - Can consist of multiple columns (composite key).
- **Syntax:**

```
CREATE TABLE table_name (
    column1 datatype PRIMARY KEY,
    column2 datatype,
    ...
);

-- Composite Primary Key
CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID)
```

```
);
```

- **Examples:**

```
-- Single-column primary key
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50) NOT NULL
);

-- Composite primary key
CREATE TABLE Enrolls (
    StudentID INT,
    CourseID INT,
    Grade CHAR(2),
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

UNIQUE Constraint

- **Purpose:** Ensures that all values in a column or a set of columns are unique, similar to primary keys but allows a single NULL value.
- **Syntax:**

```
CREATE TABLE table_name (
    column1 datatype UNIQUE,
    column2 datatype,
    ...
);

-- Multi-column unique constraint
CREATE TABLE Employees (
    Email VARCHAR(100),
    PhoneNumber VARCHAR(15),
    UNIQUE (Email, PhoneNumber)
);
```

- **Examples:**

```
-- Unique constraint on Email
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
```

```
    Email VARCHAR(100) UNIQUE
);

-- Insert duplicate Email (causes error)
INSERT INTO Employees (EmployeeID, FirstName, LastName, Email)
VALUES (1, 'John', 'Doe', 'john.doe@example.com');

INSERT INTO Employees (EmployeeID, FirstName, LastName, Email)
VALUES (2, 'Jane', 'Smith', 'john.doe@example.com'); -- Error:
Duplicate Email
```

Real-World Reference

Online Retail System:

- **Users Table:**

```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Username VARCHAR(50) UNIQUE,
    Email VARCHAR(100) UNIQUE,
    PasswordHash CHAR(64) NOT NULL
);
```

- **Explanation:**
  - **Username and Email:** Must be unique to prevent duplicate accounts.
  - **PasswordHash:** Enforces security by requiring a value (hashed password).

5.3 Referential Integrity

Referential Integrity ensures that relationships between tables remain consistent. It is enforced through foreign keys, which link tables together by referencing the primary key of another table.

FOREIGN KEY Constraint

- **Purpose:** Establish a link between two tables by ensuring that a value in one table exists in another table.
- **Syntax:**

```
CREATE TABLE child_table (
    column1 datatype,
    column2 datatype,
    FOREIGN KEY (column1) REFERENCES parent_table (columnX)
);

-- With ON DELETE and ON UPDATE actions
CREATE TABLE Orders (
```

```
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
        ON DELETE CASCADE
        ON UPDATE NO ACTION
);
```

- **Examples:**

```
-- Departments Table
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50) NOT NULL
);

-- Employees Table with Foreign Key
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

- **Explanation:**
  - **DepartmentID** in Employees must correspond to an existing DepartmentID in Departments.
  - Prevents orphan records in the Employees table.

ON DELETE and ON UPDATE Actions

- **ON DELETE CASCADE:**
  - **Behavior:** Automatically deletes related records in the child table when a record in the parent table is deleted.
  - **Use Case:** Ensures that related records do not remain orphaned.

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
        ON DELETE CASCADE
);
```

- **ON DELETE SET NULL:**

    - **Behavior:** Sets the foreign key to NULL in the child table when the corresponding parent record is deleted.
    - **Use Case:** Maintains the integrity of child records without deleting them.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    ManagerID INT,
    FOREIGN KEY (ManagerID) REFERENCES Employees(EmployeeID)
        ON DELETE SET NULL
);
```

- **ON DELETE NO ACTION / RESTRICT:**
    - **Behavior:** Prevents deletion of a parent record if related child records exist.
    - **Use Case:** Ensures critical relationships are maintained.

```
CREATE TABLE Projects (
    ProjectID INT PRIMARY KEY,
    ProjectName VARCHAR(100),
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
        ON DELETE NO ACTION
);
```

- **ON UPDATE CASCADE:**
    - **Behavior:** Automatically updates foreign key values in the child table when the corresponding primary key in the parent table changes.
    - **Use Case:** Maintains referential integrity during updates.

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100)
);

CREATE TABLE OrderDetails (
    OrderDetailID INT PRIMARY KEY,
    OrderID INT,
    ProductID INT,
    Quantity INT,
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
        ON UPDATE CASCADE
);
```

Real-World Reference

Hospital Management System:

- **Patients and Appointments:**

```
-- Patients Table
CREATE TABLE Patients (
    PatientID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DateOfBirth DATE
);

-- Appointments Table with Foreign Key
CREATE TABLE Appointments (
    AppointmentID INT PRIMARY KEY,
    PatientID INT,
    AppointmentDate DATE,
    DoctorID INT,
    FOREIGN KEY (PatientID) REFERENCES Patients(PatientID)
        ON DELETE CASCADE
);
```

- **Explanation:**
  - **Foreign Key:** Ensures that every appointment is linked to an existing patient.
  - **ON DELETE CASCADE:** Automatically deletes appointments if the corresponding patient record is removed, maintaining data consistency.

Unit VI: Advanced SQL

Advanced SQL features provide enhanced capabilities for complex data manipulation, retrieval, and analysis, enabling more sophisticated database operations.

6.1 Joins

Joins are used to combine rows from two or more tables based on related columns. They are essential for querying data across multiple tables.

Types of Joins

1. **INNER JOIN:**
   a. **Purpose:** Returns records with matching values in both tables.
   b. **Use Case:** Retrieve combined data only when there is a match in both tables.
   c. Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2 ON table1.common_column = table2.common_column;
```

   d. Example:

```
SELECT Employees.FirstName, Departments.DepartmentName
FROM Employees
```

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

2. **LEFT OUTER JOIN (LEFT JOIN):**
    a. **Purpose:** Returns all records from the left table and the matched records from the right table. Unmatched records from the right table appear as NULL.
    b. **Use Case:** Identify records in the left table without corresponding matches in the right table.
    c. **Syntax:**

```
SELECT columns
FROM table1
LEFT JOIN table2 ON table1.common_column = table2.common_column;
```

    d. **Example:**

```
SELECT Employees.FirstName, Departments.DepartmentName
FROM Employees
LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

3. **RIGHT OUTER JOIN (RIGHT JOIN):**
    a. **Purpose:** Returns all records from the right table and the matched records from the left table. Unmatched records from the left table appear as NULL.
    b. **Use Case:** Identify records in the right table without corresponding matches in the left table.
    c. **Syntax:**

```
SELECT columns
FROM table1
RIGHT JOIN table2 ON table1.common_column = table2.common_column;
```

    d. **Example:**

```
SELECT Employees.FirstName, Departments.DepartmentName
FROM Employees
RIGHT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

4. **FULL OUTER JOIN:**
    a. **Purpose:** Returns all records when there is a match in either left or right table. Non-matching records appear as NULL on the side without a match.
    b. **Use Case:** Combine all records from both tables, including unmatched ones.
    c. **Syntax:**

```
SELECT columns
FROM table1
FULL OUTER JOIN table2 ON table1.common_column = table2.common_column;
```

    d. **Example:**

```
SELECT Employees.FirstName, Departments.DepartmentName
FROM Employees
FULL OUTER JOIN Departments ON Employees.DepartmentID =
Departments.DepartmentID;
```

5. **SELF JOIN:**
    a. **Purpose:** Joins a table to itself, allowing for comparisons within the same table.
    b. **Use Case:** Analyze hierarchical or relational data within a single table.
    c. **Syntax:**

```
SELECT A.columns, B.columns
FROM table1 A
JOIN table1 B ON A.common_column = B.common_column;
```

    d. **Example:**

```
SELECT A.FirstName AS Employee, B.FirstName AS Manager
FROM Employees A
LEFT JOIN Employees B ON A.ManagerID = B.EmployeeID;
```

Join Conditions and Equijoin vs. Non-Equijoin

1. **Join Conditions:**
    a. Define how rows from different tables are matched.
    b. Commonly based on equality (=) but can involve other operators.
2. **Equijoin:**
    a. Uses equality (=) in the join condition.
    b. Most common type of join.

```
SELECT Employees.FirstName, Departments.DepartmentName
FROM Employees
JOIN Departments ON Employees.DepartmentID =
Departments.DepartmentID;
```

3. **Non-Equijoin:**
    a. Uses operators other than equality (<, >, >=, <=, !=) in the join condition.
    b. Useful for range-based matches.

```
SELECT Employees.FirstName, Salaries.SalaryGrade
FROM Employees
JOIN Salaries ON Employees.Salary BETWEEN Salaries.MinSalary AND
Salaries.MaxSalary;
```

Join Performance Considerations

- **Indexes:**
    - Proper indexing on join columns can significantly improve join performance.
- **Join Order:**
    - Optimizing the sequence in which tables are joined can reduce computation time.
- **Data Volume:**
    - Large datasets may require optimized query strategies to maintain performance.
- **Use of Temporary Tables:**
    - In some cases, breaking complex joins into stages using temporary tables can enhance performance.

Real-World Reference

Employee Hierarchy Analysis:

- **Scenario:** Determine each employee's manager and department.

```
SELECT e.FirstName AS EmployeeName, m.FirstName AS ManagerName,
d.DepartmentName
FROM Employees e
LEFT JOIN Employees m ON e.ManagerID = m.EmployeeID
JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

- **Explanation:**
  - **Self Join:** Employees table joined to itself to associate managers.
  - **Inner Join:** Departments table ensures only employees with valid departments are listed.

6.2 Subqueries

Subqueries, also known as nested queries or inner queries, are queries embedded within other SQL statements. They enable complex data retrieval operations by using the results of one query within another.

Types of Subqueries

1. **Nested Subquery:**
   a. **Definition:** A subquery placed within another subquery.
   b. **Usage:** Allows multiple layers of data retrieval and filtering.
   c. **Example:**

```
SELECT FirstName, LastName
FROM Employees
WHERE DepartmentID IN (
    SELECT DepartmentID
    FROM Departments
    WHERE Location = 'New York'
);
```

2. **Correlated Subquery:**
   a. **Definition:** A subquery that references columns from the outer query, making it dependent on the outer query.
   b. **Usage:** Performs row-by-row processing and is evaluated for each row in the outer query.
   c. **Example:**

```
SELECT e1.FirstName, e1.LastName
FROM Employees e1
WHERE e1.Salary > (
    SELECT AVG(e2.Salary)
    FROM Employees e2
    WHERE e2.DepartmentID = e1.DepartmentID
```

```
);
```

3. **Non-Correlated Subquery:**
    a. **Definition:** A subquery that does not reference columns from the outer query and can be executed independently.
    b. **Usage:** Simplifies queries where the subquery result is static relative to the outer query.
    c. **Example:**

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > (
    SELECT AVG(Salary)
    FROM Employees
);
```

4. **Multi-Column Subquery:**
    a. **Definition:** A subquery that returns multiple columns.
    b. **Usage:** Used in scenarios requiring comparison of multiple attributes.
    c. **Example:**

```
SELECT FirstName, LastName
FROM Employees
WHERE (DepartmentID, Salary) IN (
    SELECT DepartmentID, MAX(Salary)
    FROM Employees
    GROUP BY DepartmentID
);
```

5. **Exists and Not Exists Subqueries:**
    a. **Purpose:** Checks for the existence or non-existence of rows satisfying criteria.
    b. **Syntax:**

```
SELECT columns
FROM table1
WHERE EXISTS (subquery);
```

    c. **Examples:**

```
-- Employees with existing orders
SELECT FirstName, LastName
FROM Employees
WHERE EXISTS (
    SELECT * FROM Orders
    WHERE Orders.EmployeeID = Employees.EmployeeID
);

-- Departments without any employees
SELECT DepartmentName
FROM Departments
WHERE NOT EXISTS (
```

```
    SELECT * FROM Employees
    WHERE Employees.DepartmentID = Departments.DepartmentID
);
```

Advantages of Using Subqueries

- **Modularity:** Break down complex queries into manageable parts.
- **Reusability:** Use results from one query within another.
- **Flexibility:** Enable dynamic data retrieval based on conditional logic.

Disadvantages of Using Subqueries

- **Performance Overhead:** Can lead to slower query execution, especially with correlated subqueries.
- **Complexity:** Nested queries can be harder to read and maintain.
- **Less Optimization:** Some DBMS optimizers handle joins more efficiently than certain subqueries.

Best Practices

- **Prefer Joins Over Subqueries:** When possible, use joins for better performance and readability.
- **Use EXISTS Instead of IN for Large Datasets:** EXISTS can be more efficient, especially with indexed tables.
- **Limit Nesting Depth:** Keep subqueries to a manageable level to maintain readability and performance.
- **Utilize CTEs (Common Table Expressions):** Simplify complex subqueries by using WITH clauses.

```
WITH DepartmentSales AS (
    SELECT DepartmentID, SUM(SaleAmount) AS TotalSales
    FROM Sales
    GROUP BY DepartmentID
)
SELECT Departments.DepartmentName, DepartmentSales.TotalSales
FROM Departments
JOIN DepartmentSales ON Departments.DepartmentID =
DepartmentSales.DepartmentID
WHERE DepartmentSales.TotalSales > 100000;
```

Real-World Reference

Employee Bonus Allocation:

- **Scenario:** Assign bonuses to employees whose salary exceeds the average salary in their department.

```
UPDATE Employees e1
SET Bonus = 5000
WHERE e1.Salary > (
```

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

- **Explanation:**
  - **Correlated Subquery:** Computes the average salary for each department relative to each employee.

6.3 Advanced Operators

Advanced SQL operators extend the capabilities of basic operators, allowing for more nuanced and powerful data manipulations and retrievals.

IN Operator

- **Purpose:** Specifies a list of possible values for a column, filtering records that match any value in the list.
- **Syntax:**

```
SELECT columns
FROM table
WHERE column IN (value1, value2, ...);
```

- **Example:**

```
SELECT FirstName, LastName
FROM Employees
WHERE DepartmentID IN (10, 20, 30);
```

- **Use Case:** Select employees belonging to multiple departments without multiple OR conditions.

NOT IN Operator

- **Purpose:** Specifies a list of values to exclude, filtering out records that match any value in the list.
- **Syntax:**

```
SELECT columns
FROM table
WHERE column NOT IN (value1, value2, ...);
```

- **Example:**

```
SELECT FirstName, LastName
FROM Employees
WHERE DepartmentID NOT IN (10, 20);
```

- **Use Case:** Identify employees not assigned to core departments.

ANY Operator

- **Purpose:** Compares a value to each value in a list or subquery, returning TRUE if any comparison is TRUE.
- **Syntax:**

```
SELECT columns
FROM table
WHERE column operator ANY (subquery);
```

  - **Common Operators:** =, !=, <, >, <=, >=
- **Example:**

```
-- Select employees earning more than any employee in Department 10
SELECT FirstName, LastName, Salary
FROM Employees
WHERE Salary > ANY (
    SELECT Salary
    FROM Employees
    WHERE DepartmentID = 10
);
```

- **Use Case:** Identify high-performing employees compared to peers in a specific department.

ALL Operator

- **Purpose:** Compares a value to all values in a list or subquery, returning TRUE if all comparisons are TRUE.
- **Syntax:**

```
SELECT columns
FROM table
WHERE column operator ALL (subquery);
```

  - **Common Operators:** =, !=, <, >, <=, >=
- **Example:**

```
-- Select employees earning more than all employees in Department 10
SELECT FirstName, LastName, Salary
FROM Employees
WHERE Salary > ALL (
    SELECT Salary
    FROM Employees
    WHERE DepartmentID = 10
);
```

- **Use Case:** Identify top earners who surpass all counterparts in a specific department.

EXISTS Operator

- **Purpose:** Checks for the existence of rows in a subquery that meet a specified condition.
- **Syntax:**

```
SELECT columns
FROM table1
WHERE EXISTS (subquery);
```

- **Example:**

```
-- Select departments that have at least one employee
SELECT DepartmentName
FROM Departments d
WHERE EXISTS (
    SELECT 1
    FROM Employees e
    WHERE e.DepartmentID = d.DepartmentID
);
```

- **Use Case:** Identify departments actively employing staff.

NOT EXISTS Operator

- **Purpose:** Checks for the non-existence of rows in a subquery that meet a specified condition.
- **Syntax:**

```
SELECT columns
FROM table1
WHERE NOT EXISTS (subquery);
```

- **Example:**

```
-- Select departments with no employees
SELECT DepartmentName
FROM Departments d
WHERE NOT EXISTS (
    SELECT 1
    FROM Employees e
    WHERE e.DepartmentID = d.DepartmentID
);
```

- **Use Case:** Identify departments that are currently vacant.

Example Usage in Context

Scenario: Identify Employees Who Have Not Submitted Any Expense Reports.

```
SELECT e.FirstName, e.LastName
FROM Employees e
WHERE NOT EXISTS (
    SELECT 1
    FROM ExpenseReports er
    WHERE er.EmployeeID = e.EmployeeID
);
```

- **Explanation:**
  - **NOT EXISTS:** Filters out employees who have submitted at least one expense report.
  - **Subquery:** Checks for the existence of related records in the ExpenseReports table.

6.4 Case Expressions

The **CASE Expression** in SQL allows for conditional logic within queries, enabling dynamic data transformation and categorization based on specified conditions.

Purpose

- **Conditional Data Transformation:** Assign values based on conditions.
- **Data Categorization:** Group data into categories based on business logic.
- **Dynamic Calculations:** Modify or compute values conditionally.

Syntax

There are two forms of the CASE expression: **Simple CASE** and **Searched CASE**.

1. **Simple CASE:**
   a. Compares an expression to a set of simple expressions to determine the result.
   b. **Syntax:**

```
CASE expression
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    ...
    ELSE resultN
END
```

   c. **Example:**

```
SELECT FirstName, LastName,
    CASE DepartmentID
        WHEN 10 THEN 'Human Resources'
        WHEN 20 THEN 'Engineering'
        WHEN 30 THEN 'Sales'
        ELSE 'Other'
    END AS DepartmentName
FROM Employees;
```

2. **Searched CASE:**
    a. Evaluates a set of Boolean expressions to determine the result.
    b. **Syntax:**

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE resultN
END
```

    c. **Example:**

```
SELECT FirstName, LastName, Salary,
    CASE
        WHEN Salary > 70000 THEN 'High'
        WHEN Salary BETWEEN 50000 AND 70000 THEN 'Medium'
        ELSE 'Low'
    END AS SalaryGrade
FROM Employees;
```

Nested CASE Expressions

- **Definition:**
    - Embedding one CASE expression within another to handle multiple layers of conditions.
- **Syntax:**

```
CASE
    WHEN condition1 THEN
        CASE
            WHEN subcondition1 THEN result1
            ELSE result2
        END
    WHEN condition2 THEN result3
    ELSE result4
END
```

- **Example:**

```
SELECT FirstName, LastName, Salary,
    CASE
        WHEN Salary > 70000 THEN
            CASE
                WHEN DepartmentID = 10 THEN 'Senior Engineer'
                ELSE 'Senior Staff'
            END
        WHEN Salary BETWEEN 50000 AND 70000 THEN 'Mid-Level'
        ELSE 'Entry-Level'
```

```
    END AS PositionLevel
FROM Employees;
```

Using CASE in ORDER BY

- **Purpose:** Sort data based on dynamic conditions.
- **Example:**

```
SELECT FirstName, LastName, Salary,
    CASE
        WHEN Salary > 70000 THEN 'High'
        WHEN Salary BETWEEN 50000 AND 70000 THEN 'Medium'
        ELSE 'Low'
    END AS SalaryGrade
FROM Employees
ORDER BY
    CASE
        WHEN SalaryGrade = 'High' THEN 1
        WHEN SalaryGrade = 'Medium' THEN 2
        ELSE 3
    END;
```

Example Usage in Context

Scenario: Categorize Employees Based on Years of Service.

```
SELECT FirstName, LastName, HireDate,
    CASE
        WHEN DATEDIFF(year, HireDate, GETDATE()) >= 10 THEN
'Veteran'
        WHEN DATEDIFF(year, HireDate, GETDATE()) BETWEEN 5 AND 9
THEN 'Experienced'
        ELSE 'Newcomer'
    END AS ServiceCategory
FROM Employees;
```

- **Explanation:**
    - **DATEDIFF():** Calculates the difference in years between HireDate and the current date.
    - **CASE Expression:** Assigns a ServiceCategory based on years of service.

Real-World Reference

Sales Performance Reporting:

- **Scenario:** Assign performance tiers based on sales amounts.

```
SELECT SalespersonID, SalesAmount,
    CASE
        WHEN SalesAmount > 100000 THEN 'Excellent'
        WHEN SalesAmount BETWEEN 50000 AND 100000 THEN 'Good'
```

```
        ELSE 'Needs Improvement'
    END AS PerformanceTier
FROM SalesPerformance;
```

- **Explanation:**
    - **CASE Expression:** Categorizes sales performance into tiers for reporting and analysis.

Practical Applications

Practical applications bridge theoretical knowledge with real-world scenarios, demonstrating how database concepts and SQL commands are utilized in actual systems.

7.1 Designing E-R Diagrams and Converting to Relational Models

Designing Entity-Relationship (E-R) diagrams is the first step in database design, providing a visual blueprint of the database structure. Converting these diagrams into relational models involves creating structured tables with defined relationships.

Step-by-Step Guide

1. **Identify Entities:**
    a. Determine the primary objects or concepts that need to be represented in the database.
    b. **Example Entities:** Student, Course, Instructor.
2. **Identify Attributes:**
    a. Define properties or characteristics of each entity.
    b. Example Attributes:
        i. *Student:* StudentID, Name, Major.
        ii. *Course:* CourseID, Title, Credits.
        iii. *Instructor:* InstructorID, Name, Department.
3. **Determine Relationships:**
    a. Establish how entities interact or are associated with each other.
    b. Example Relationships:
        i. *Enrolls:* Students enroll in Courses.
        ii. *Teaches:* Instructors teach Courses.
4. **Draw E-R Diagram:**
    a. Use standardized symbols to represent entities, attributes, and relationships.
    b. Clearly indicate cardinality (e.g., one-to-many).
5. **Assign Primary Keys:**
    a. Choose unique identifiers for each entity.
    b. **Example:** StudentID for Student, CourseID for Course.
6. **Normalize the Design:**
    a. Apply normalization principles to eliminate redundancy and ensure data integrity.
7. **Convert to Relational Model:**
    a. Create tables based on entities and relationships.
    b. Define primary and foreign keys to establish links between tables.

Example: Student-Course Enrollment

Entities:

- **Student:**
  - **Attributes:** StudentID (PK), Name, Major.
- **Course:**
  - **Attributes:** CourseID (PK), Title, Credits.
- **Instructor:**
  - **Attributes:** InstructorID (PK), Name, Department.

Relationships:

- **Enrolls:** Many-to-Many between Student and Course.
- **Teaches:** One-to-Many from Instructor to Course.

E-R Diagram:

Figure 7.1: Student-Course Enrollment E-R Diagram

Relational Model Conversion:

1. **Students Table:**

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100),
    Major VARCHAR(50)
);
```

2. **Courses Table:**

```
CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    Title VARCHAR(100),
    Credits INT,
    InstructorID INT,
    FOREIGN KEY (InstructorID) REFERENCES Instructors(InstructorID)
);
```

3. **Instructors Table:**

```
CREATE TABLE Instructors (
    InstructorID INT PRIMARY KEY,
    Name VARCHAR(100),
    Department VARCHAR(50)
);
```

4. **Enrollments (Associative Table):**

```
CREATE TABLE Enrollments (
    StudentID INT,
    CourseID INT,
    Grade CHAR(2),
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

Real-World Reference

Hospital Management System:

- **Entities:** Patients, Doctors, Appointments, Departments.
- **Relationships:** Patients schedule Appointments with Doctors in specific Departments.
- **E-R Diagram:**

Figure 7.2: Hospital Management E-R Diagram

- **Relational Model Conversion:**
  - **Patients Table**
  - **Doctors Table**
  - **Appointments Table**
  - **Departments Table**
- **Considerations:**
  - **Appointments:** Associative entity linking Patients and Doctors.
  - **Departments:** Linked to Doctors, indicating their specialization.

7.2 Implementing SQL Commands for Database Creation, Manipulation, and Querying

Implementing SQL commands involves translating database designs into actual database structures and performing operations on the data.

Example: Employee Management System

Scenario: Create and manage an Employee Management System with Departments and Employees, including inserting data, querying, updating, and deleting records.

1. Create Tables

- **Departments Table:**

```
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50) NOT NULL,
    Location VARCHAR(100)
);
```

- **Employees Table:**

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DepartmentID INT,
    Salary DECIMAL(10, 2),
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

2. Insert Data

- **Insert into Departments:**

```
INSERT INTO Departments (DepartmentID, DepartmentName, Location)
VALUES
    (10, 'Human Resources', 'New York'),
    (20, 'Engineering', 'San Francisco'),
    (30, 'Sales', 'Chicago');
```

- **Insert into Employees:**

```
INSERT INTO Employees (EmployeeID, FirstName, LastName,
DepartmentID, Salary)
VALUES
    (1, 'John', 'Doe', 10, 60000.00),
    (2, 'Jane', 'Smith', 20, 80000.00),
    (3, 'Bob', 'Johnson', 10, 55000.00),
```

```
    (4, 'Alice', 'Williams', 30, 70000.00);
```

3. Query Data

- **Select All Employees:**

```
SELECT * FROM Employees;
```

- **Select Employees in Engineering Department:**

```
SELECT FirstName, LastName, Salary
FROM Employees
WHERE DepartmentID = 20;
```

- **Join Employees with Departments:**

```
SELECT Employees.FirstName, Employees.LastName,
Departments.DepartmentName, Employees.Salary
FROM Employees
INNER JOIN Departments ON Employees.DepartmentID =
Departments.DepartmentID;
```

- **Aggregate Query - Average Salary per Department:**

```
SELECT DepartmentID, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY DepartmentID;
```

4. Update Data

- **Increase Salary by 5% for Employees in Human Resources:**

```
UPDATE Employees
SET Salary = Salary * 1.05
WHERE DepartmentID = 10;
```

- **Change Department of an Employee:**

```
UPDATE Employees
SET DepartmentID = 30
WHERE EmployeeID = 3;
```

5. Delete Data

- **Delete an Employee Record:**

```
DELETE FROM Employees
WHERE EmployeeID = 4;
```

- **Delete All Employees in Sales Department:**

```
DELETE FROM Employees
WHERE DepartmentID = 30;
```

6. Advanced Queries

- **Retrieve Employees with Salary Above Average:**

```
SELECT FirstName, LastName, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

- **List Departments and Number of Employees:**

```
SELECT d.DepartmentName, COUNT(e.EmployeeID) AS NumberOfEmployees
FROM Departments d
LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID
GROUP BY d.DepartmentName;
```

7. Backup and Restore

- **Backup (SQL Server Example):**

```
BACKUP DATABASE EmployeeDB
TO DISK = 'C:\Backups\EmployeeDB.bak';
```

- **Restore (SQL Server Example):**

```
RESTORE DATABASE EmployeeDB
FROM DISK = 'C:\Backups\EmployeeDB.bak'
WITH REPLACE;
```

Real-World Reference

Inventory Management System:

- **Tables: Products, Suppliers, Orders, Inventory.**
- **Creating Products Table:**

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100) NOT NULL,
    SupplierID INT,
    Price DECIMAL(10, 2),
    FOREIGN KEY (SupplierID) REFERENCES Suppliers(SupplierID)
);
```

- **Creating Suppliers Table:**

```
CREATE TABLE Suppliers (
    SupplierID INT PRIMARY KEY,
    SupplierName VARCHAR(100) NOT NULL,
```

```
    ContactInfo VARCHAR(150)
);
```

- **Inserting Data:**

```
INSERT INTO Suppliers (SupplierID, SupplierName, ContactInfo)
VALUES
    (1, 'SupplierA', 'contactA@example.com'),
    (2, 'SupplierB', 'contactB@example.com');

INSERT INTO Products (ProductID, ProductName, SupplierID, Price)
VALUES
    (101, 'Widget', 1, 25.50),
    (102, 'Gizmo', 2, 35.75);
```

- **Querying Inventory Levels:**

```
SELECT Products.ProductName, Inventory.Quantity
FROM Products
JOIN Inventory ON Products.ProductID = Inventory.ProductID
WHERE Inventory.Quantity < 50;
```

- **Updating Product Price:**

```
UPDATE Products
SET Price = Price * 1.10
WHERE ProductID = 101;
```

7.3 Solving Normalization Problems up to BCNF

Normalization problems require identifying functional dependencies and performing table decompositions to achieve higher normal forms, eliminating anomalies and ensuring data integrity.

Problem Statement

Given the `Order` table:

OrderID	ProductID	ProductName	Quantity	SupplierID	SupplierName
1001	P01	Widget	10	S01	SupplierA
1002	P02	Gizmo	5	S02	SupplierB
1003	P01	Widget	7	S01	SupplierA

Step 1: Identify Functional Dependencies

1. **OrderID → ProductID, ProductName, Quantity, SupplierID, SupplierName**
2. **ProductID → ProductName, SupplierID, SupplierName**

3. **SupplierID → SupplierName**

Explanation:

- Each **OrderID** uniquely identifies the order details.
- Each **ProductID** uniquely determines the product details.
- Each **SupplierID** uniquely determines the supplier name.

Step 2: Analyze Normal Forms

1. **First Normal Form (1NF):**
    a. **Check:** All cells contain atomic values (no repeating groups).
    b. **Result:** Satisfies 1NF.
2. **Second Normal Form (2NF):**
    a. **Check:** No partial dependencies on a composite primary key.
    b. **Issue:** Assuming OrderID is the primary key, no partial dependencies exist as it's not a composite key.
    c. **Result:** Satisfies 2NF.
3. **Third Normal Form (3NF):**
    a. **Check:** No transitive dependencies (non-prime attributes do not depend on other non-prime attributes).
    b. Issue:
        i. **ProductID → SupplierID → SupplierName**, creating a transitive dependency.
        ii. **SupplierID → SupplierName** is a transitive dependency through ProductID.
    c. **Result:** Violates 3NF.
4. **Boyce-Codd Normal Form (BCNF):**
    a. **Check:** For every non-trivial FD X → Y, X must be a superkey.
    b. Issue:
        i. In FD ProductID → ProductName, SupplierID, SupplierName, ProductID is not a superkey.
        ii. Violates BCNF.

Step 3: Normalize to BCNF

1. **Decompose Based on Non-BCNF FD: ProductID → ProductName, SupplierID, SupplierName**
    a. Create Products Table:

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    SupplierID INT,
    FOREIGN KEY (SupplierID) REFERENCES Suppliers(SupplierID)
);
```

    b. Create Suppliers Table:

```
CREATE TABLE Suppliers (
    SupplierID INT PRIMARY KEY,
    SupplierName VARCHAR(100)
);
```

    c. Modify Orders Table:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    ProductID INT,
    Quantity INT,
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

2. **Resulting Tables:**
    a. **Orders Table:**

OrderID	ProductID	Quantity
1001	P01	10
1002	P02	5
1003	P01	7

    b. **Products Table:**

ProductID	ProductName	SupplierID
P01	Widget	S01
P02	Gizmo	S02

    c. **Suppliers Table:**

SupplierID	SupplierName
S01	SupplierA
S02	SupplierB

3. **Verify Normal Forms:**
    a. **Orders Table:** Satisfies BCNF as OrderID is the primary key.
    b. **Products Table:** Satisfies BCNF as ProductID is the primary key.
    c. **Suppliers Table:** Satisfies BCNF as SupplierID is the primary key.

Diagram 7.3: Normalization Process to BCNF

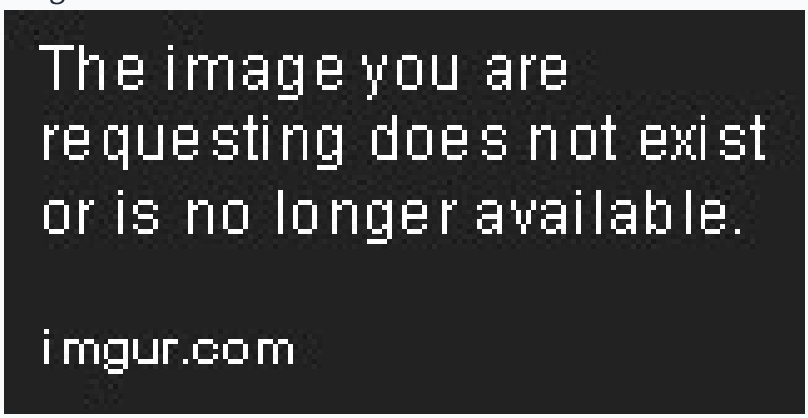

Figure 7.3: Decomposing Order Table into BCNF

Explanation:

- **Original Dependency:** ProductID determines Supplier details, violating BCNF.
- **Decomposition:** Split into Orders, Products, and Suppliers tables to eliminate transitive dependencies.

7.4 Use Case Examples

Use case examples illustrate practical implementations of database concepts, providing hands-on scenarios for better understanding.

7.4.1 Employee Database

Entities:

- **Employees:** EmployeeID, FirstName, LastName, DepartmentID, Salary.
- **Departments:** DepartmentID, DepartmentName, Location.
- **Projects:** ProjectID, ProjectName, DepartmentID.
- **ProjectAssignments:** EmployeeID, ProjectID, AssignmentDate.

Use Cases:

1. **Adding New Employees and Assigning to Departments.**
2. **Tracking Employee Participation in Projects.**
3. **Generating Reports on Department-wise Employee Count and Salary Expenditure.**

Sample SQL Queries:

- **List Employees in Engineering Department:**

```
SELECT e.FirstName, e.LastName
FROM Employees e
INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID
WHERE d.DepartmentName = 'Engineering';
```

- **Assign Employee to Project:**

```
INSERT INTO ProjectAssignments (EmployeeID, ProjectID,
AssignmentDate)
VALUES (1, 'PRJ1001', '2023-09-15');
```

- **Generate Department-wise Salary Report:**

```
SELECT d.DepartmentName, COUNT(e.EmployeeID) AS EmployeeCount,
SUM(e.Salary) AS TotalSalary
FROM Departments d
LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID
GROUP BY d.DepartmentName;
```

- **Identify Employees Not Assigned to Any Projects:**

```
SELECT e.FirstName, e.LastName
FROM Employees e
WHERE NOT EXISTS (
    SELECT 1
    FROM ProjectAssignments pa
    WHERE pa.EmployeeID = e.EmployeeID
```

```
);
```

7.4.2 Student Academic Database

Entities:

- **Students:** StudentID, Name, Major, AdmissionDate.
- **Courses:** CourseID, Title, Credits, InstructorID.
- **Instructors:** InstructorID, Name, Department.
- **Enrollments:** StudentID, CourseID, Grade.

Use Cases:

1. **Enrolling Students in Courses.**
2. **Assigning Instructors to Courses.**
3. **Calculating Student GPAs and Course Averages.**
4. **Generating Reports on Course Enrollments and Instructor Performance.**

Sample SQL Queries:

- **Enroll a Student in a Course:**

```
INSERT INTO Enrollments (StudentID, CourseID, Grade)
VALUES (1001, 'CS101', 'A');
```

- **Calculate GPA:**

```
SELECT StudentID, AVG(
    CASE Grade
        WHEN 'A' THEN 4.0
        WHEN 'B' THEN 3.0
        WHEN 'C' THEN 2.0
        WHEN 'D' THEN 1.0
        ELSE 0.0
    END
) AS GPA
FROM Enrollments
GROUP BY StudentID;
```

- **List Courses with High Enrollment:**

```
SELECT c.CourseID, c.Title, COUNT(e.StudentID) AS EnrollmentCount
FROM Courses c
JOIN Enrollments e ON c.CourseID = e.CourseID
GROUP BY c.CourseID, c.Title
HAVING COUNT(e.StudentID) > 50;
```

- **Assign Instructor to Course:**

```
UPDATE Courses
SET InstructorID = 2001
```

```
WHERE CourseID = 'CS101';
```

7.4.3 Inventory Management System

Entities:

- **Products:** ProductID, ProductName, SupplierID, Price.
- **Suppliers:** SupplierID, SupplierName, ContactInfo.
- **Inventory:** ProductID, Quantity, WarehouseLocation.
- **Orders:** OrderID, ProductID, Quantity, SupplierID, OrderDate.
- **Warehouses:** WarehouseID, Location.

Use Cases:

1. **Tracking Product Stock Levels in Warehouses.**
2. **Managing Supplier Information and Contact Details.**
3. **Processing Purchase Orders and Managing Inventory Restocking.**
4. **Generating Reports on Inventory Status and Supplier Performance.**

Sample SQL Queries:

- **Check Low Stock Products:**

```
SELECT p.ProductName, i.Quantity, w.Location
FROM Products p
JOIN Inventory i ON p.ProductID = i.ProductID
JOIN Warehouses w ON i.WarehouseLocation = w.WarehouseID
WHERE i.Quantity < 50;
```

- **Place an Order with Supplier:**

```
INSERT INTO Orders (OrderID, ProductID, Quantity, SupplierID,
OrderDate)
VALUES (5001, 'P03', 100, 'S03', GETDATE());
```

- **Update Inventory After Order Shipment:**

```
UPDATE Inventory
SET Quantity = Quantity - 100
WHERE ProductID = 'P03' AND WarehouseLocation = 'W01';
```

- **Generate Supplier Performance Report:**

```
SELECT s.SupplierName, COUNT(o.OrderID) AS NumberOfOrders,
SUM(o.Quantity) AS TotalQuantity
FROM Suppliers s
JOIN Orders o ON s.SupplierID = o.SupplierID
GROUP BY s.SupplierName
HAVING COUNT(o.OrderID) > 10
ORDER BY TotalQuantity DESC;
```

Real-World Reference

E-commerce Platform:

- **Managing Customer Orders:**

```
-- Create Customers Table
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100) UNIQUE,
    RegistrationDate DATE
);

-- Create Orders Table
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    TotalAmount DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

-- Insert Data
INSERT INTO Customers (CustomerID, Name, Email, RegistrationDate)
VALUES (1, 'Alice Wonderland', 'alice@example.com', '2023-01-15'),
       (2, 'Bob Builder', 'bob@example.com', '2023-02-20');

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount)
VALUES (101, 1, '2023-03-10', 250.00),
       (102, 2, '2023-03-12', 150.00),
       (103, 1, '2023-03-15', 300.00);

-- Query Customer Orders
SELECT c.Name, o.OrderID, o.OrderDate, o.TotalAmount
FROM Customers c
JOIN Orders o ON c.CustomerID = o.CustomerID
WHERE c.CustomerID = 1;
```

- **Explanation:**
    - **Customers and Orders Tables:** Linked through CustomerID.
    - **Foreign Key Enforcement:** Ensures orders are associated with existing customers.
    - **Queries:** Retrieve orders placed by a specific customer.