DATABASE MANAGEMENT SYSTEMS

M.Sc. Computer Science

First Year, Semester-II, Paper-I

Lesson Writers

Dr. Neelima Guntupalli
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Vasantha Rudramalla
Faculty,
Department of CS&E
Acharya Nagarjuna University

Dr. U. Surya Kameswari
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Editor

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
University College of Sciences
Acharya Nagarjuna University

Academic Advisor

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

DIRECTOR, I/c.
PROF. V. VENKATESWARLU
M.A., M.P.S., M.S.W., M.Phil., Ph.D.
CENTRE FOR DISTANCE EDUCATION
ACHARYA NAGARJUNA UNIVERSITY
NAGARJUNA NAGAR 522 510
Ph: 0863-2346222, 2346208
0863- 2346259 (Study Material)
Website www.anucde.info
E-mail: anucdedirector@gmail.com

M.Sc., (Computer Science) : DATABASE MANAGEMENT SYSTEMS

First Edition : 2025

No. of Copies :

This book is exclusively prepared for the use of students of M.Sc. (Computer Science), Centre for Distance Education, Acharya Nagarjuna University and this book is meant for limited circulation only.

Published by:

Prof. V. VENKATESWARLU
Director, I/c
Centre for Distance Education,
Acharya Nagarjuna University

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University.

M.Sc., (COMPUTER SCIENCE)
FIRST YEAR, SEMESTER – II

201CP24: DATABASE MANAGENT SYSTEMS

SYLLABUS

UNIT-I

Databases and Database Users Introduction, Characteristics of the Database Approach, Actors on the Scene. Workers behind the scene, Advantages of the using the DBMS Approach.

Database System Concepts and Architecture Data Models, Schemas and Instances, Three Schema architecture and Data Independence, Database Languages and Interfaces, Centralized and Client/Server Architecture for DBMS, Classification of Database Management Systems.

UNIT-II

Data Modeling Using the ER Model Conceptual Data models, Entity Types, Entity Sets, Attributes and Keys, Relationship types, Relationship sets, roles and structural Constraints, Weak Entity types, Relationship Types of Degree Higher than Two, Refining the ER Design for the COMPANY Database.

The Enhanced Entity-Relationship Model Sub classes, Super classes and Inheritance, Specialization and Generalization, Constraints and Characteristics of Specialization and Generalization Hierarchies, Modeling of Union Types using Categories, An Example University ERR Schema, Design Choices and Formal Definitions.

UNIT-III

The Relational Data Model and Relational Database Constraints Relational Model Concepts, Relational Model Constraints and Relational Database Schemas, Update Operations, Transactions and Dealing with Constraint Violations.

The Relational Algebra and Relational Calculus Unary Relational operations SELECT and PROJECT, Relational Algebra operations from set Theory, Binary Relational Operations JOIN and DIVISION, Additional Relational operations, Examples, The Tuple Calculus and Domain Calculus.

SQL-99 Schema Definition, Constraints, Queries and Views SQL Data Definitions and Data Types, Specifying Constraints in SQL, Schena Change Statements on SQL, Basic Queries in SQL, More Complex SQL Queries, INSERT, DELETE and UPDATE statements in SQL, Triggers and Views.

UNIT. IV

Functional Dependencies and Normalization for Relational Databases Informal Design Guidelines for Relation Schemas, Functional dependencies, Normal Forms Based in primary Keys, General Definitions of Second and Third Normal Forms, Boyce-Codd Normal Form.

Relational Database Design Algorithms and Further Dependencies Properties of Relational Decompositions, Algorithms from Relational Database Schema Design, Multivalued Dependencies and Fourth Normal Form, Join Dependencies and Fifth Normal Formo Inclusion Dependencies, Other Dependencies and Normal Forms.

UNIT-V

Document oriented data principles of schema design, designing an e-commerce data model, Nuts and bolts on databases collections and documents.

Queries and Aggregation-commerce's queries, MongoDB's query language, aggregating orders, aggregating in detail.

Updates atomic operations and deletes Document updates, e-commerce updates, atomic document processing, nuts and bolts Mongo DB updates and deletes.

Prescribed Books

Ramez Elmasri, Shamkant B. Navathe, "Fundamentals of Database Systems", Fifth Edition, Pearson Education (2007).
Chapters 1.l to I .6,2,3.1to 3.6,4.1 to 4.5,5,6, 8, 10, ll

MongoDB in Action, Kyle Banker, Manning Publication and Co.**Chapters** 4,5 and 6.

Reference Books

1. C.J. Date, A.Kannan, S. Swamynathan, "An Introduction to Database Systems", VII
2. Edition Pearson Education (2006).
3. Database system concepts, Silberschatz, Korth, Sudarshan, Mc-graw-hill,5th edition.
4. MongoDBLearn MongoDB in a simple Way, Dan Warnock.

M.Sc., (Computer Science)
MODEL QUESTION PAPER
201CP24 - DATABASE MANAG ENT SYSTEMS

Time: 3 Hours **Max. Marks: 70**

Answer ONE Question from Each Unit **5 × 14 = 70 Marks**

UNIT – I

1.

a) Define Database and Database Management System. Explain the key characteristics of the database approach. (7M)

b) Discuss various actors on the scene and workers behind the scene in a DBMS environment. (7M)

OR

2.

a) Explain the three-schema architecture of a database system with a neat diagram. (7M)

b) Discuss the data models, schemas, and instances used in database systems with examples. (7M)

UNIT – II

3.

a) Define Entity, Attribute, and Relationship with suitable examples. Explain different types of attributes and keys. (7M)

b) Explain Weak Entities and Higher-Degree Relationships with an example of the COMPANY database. (7M)

OR

4.

a) Explain the concepts of Specialization, Generalization, and Inheritance in the Enhanced ER Model. (7M)

b) Discuss the design choices and formal definitions used in the EER model. (7M)

UNIT – III

5.

a) Explain Relational Model Concepts and discuss different types of Relational Constraints. (7M)

b) Discuss Unary and Binary Operations in Relational Algebra with suitable examples. (7M)

OR

6.

a) Describe DDL, DML, and DCL commands in SQL with syntax and examples. (7M)

b) Write SQL queries for the following:

i) Create a table for STUDENT (SID, NAME, AGE, COURSE).

ii) Insert a record, update AGE, and delete a student record.

iii) Create a VIEW of all students enrolled in “DBMS”. (7M)

UNIT – IV

7.

a) Define Functional Dependency and explain 1NF, 2NF, 3NF, and BCNF with examples. (7M)

b) Discuss Design Guidelines for Relation Schemas and the problems of bad database design. (7M)

OR

8.

a) Explain Multivalued Dependencies and Fourth Normal Form with examples. (7M)

b) Describe Join Dependencies and Fifth Normal Form. Explain how they help achieve good database design. (7M)

UNIT – V

9.

a) Explain Document-oriented data and the principles of schema design in MongoDB. (7M)

b) Design an e-commerce data model using document collections and explain its structure. (7M)

OR

10.

a) Explain MongoDB's query language and describe how aggregation is performed with examples. (7M)

b) Discuss atomic updates and delete operations in MongoDB. How are they useful in e-commerce applications? (7M)

CONTENTS

LESSON- 01

DATABASES AND DATABASE USERS

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Databases and Database Users. The chapter began with understanding of Characteristics of the Database Approach, Actors on the Scene, Workers behind the scene, Advantages of the using the DBMS Approach. After completing this chapter, the student will understand the complete idea about of Databases and Database Users.

STRUCTURE

1.1. INTRODUCTION

Databases are integral to modern information systems, providing structured ways to store, retrieve, and manage data. A database is a collection of related data organized to be easily accessed, managed, and updated. Databases support a wide range of applications, from small personal projects to vast enterprise systems. This chapter explores the fundamentals of databases, the database management system (DBMS) approach, and the various roles involved in managing and using databases.

The chapter first covered the Characteristics of the Database Approach, Actors on the Scene, Workers behind the scene, Advantages of the using the DBMS Approach and etc.

1.2 CHARACTERISTICS OF THE DATABASE APPROACH

The database approach offers several distinct characteristics that set it apart from traditional file systems:

- **Data Abstraction and Independence**: Databases provide a level of abstraction that hides the complexity of data storage from users. This is achieved through three levels of abstraction: the physical level, the logical level, and the view level. Data independence allows changes in the schema at one level without affecting other levels.
- **Data Integrity and Security**: DBMS enforces data integrity by ensuring accuracy and consistency of data through constraints and rules. Security measures such as authentication and authorization protect data from unauthorized access.
- **Data Sharing and Multi-user Transaction Processing**: Databases support concurrent access by multiple users. Transactions ensure that operations are completed correctly and maintain data consistency even in the presence of concurrent access and system failures.
- **Data Redundancy and Inconsistency Minimization**: Unlike file systems, databases minimize data redundancy and inconsistency by storing data in a centralized manner, reducing duplication and the chances of conflicting data.
- **Backup and Recovery**: DBMS provide mechanisms for backing up data and recovering it in case of system failures, ensuring data durability and availability.

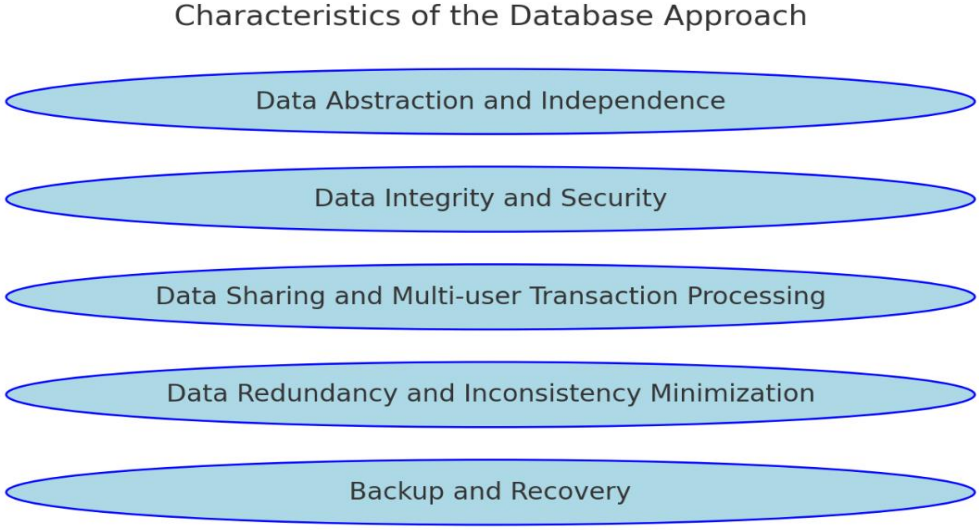


Fig 1.2 Chrematistics of DBMS

1.3 ACTORS ON THE SCENE

In the realm of databases, several key actors interact with the DBMS to perform various tasks:

- **Database Administrators (DBAs)**: DBAs are responsible for managing the DBMS, ensuring its availability, performance, and security. They handle tasks such as backup and recovery, tuning, and user management.
- **Database Designers**: These professionals design the database schema, defining the structure of the database, including tables, relationships, and constraints. They work to ensure the database meets the requirements of the application and users.
- **End Users**: End users interact with the database through applications to perform tasks such as querying, updating, and generating reports. They range from casual users with little database knowledge to sophisticated users who write complex queries.
- **Application Developers**: Developers create applications that interact with the database. They write code to perform CRUD (Create, Read, Update, Delete) operations and implement business logic.

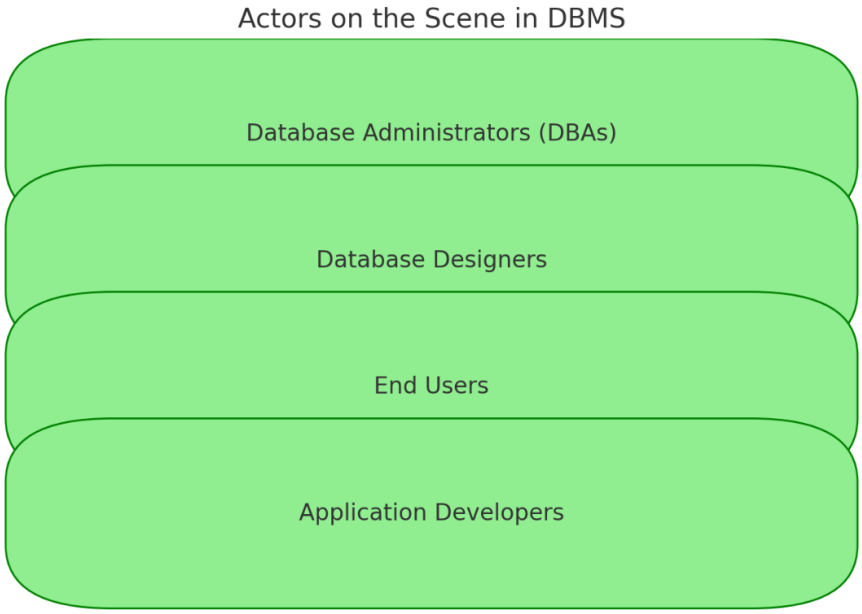


Fig 1.3 Actors in DBMS

Workers Behind the Scene

Several important roles operate behind the scenes to ensure the smooth functioning of a database system:

- **System Designers and Architects**: These professionals design the overall architecture of the database system, including hardware, software, and network components. They ensure that the system can handle the required workload and provide necessary scalability and reliability.
- **System Administrators**: System administrators manage the hardware and operating systems on which the DBMS runs. They ensure that the underlying infrastructure supports the database's performance and availability needs.

- **Data Analysts and Scientists**: These individuals analyze data to extract meaningful insights and support decision-making processes. They use various tools and techniques to process and interpret data stored in the database.
- **Support and Maintenance Staff**: These team members provide ongoing support, troubleshoot issues, and perform routine maintenance tasks to ensure the database system operates smoothly.

1.4 ADVANTAGES OF USING THE DBMS APPROACH

The Database Management System (DBMS) approach offers numerous advantages over traditional file-based data management systems. These benefits significantly enhance data management efficiency, security, and accessibility, providing a robust framework for handling data in modern organizations.

1.4.1 Advantages of DBMS

- **Improved Data Sharing**
- DBMS enables multiple users to access and share data simultaneously, promoting collaboration and information exchange within an organization.

- **Enhanced Data Security**
- DBMS provides robust security features to protect sensitive data from unauthorized access and breaches. This includes user authentication, authorization, and encryption.

- **Better Data Integration**
- By centralizing data storage, DBMS ensures that data from different sources is integrated into a single, coherent database, facilitating comprehensive data.

- **Reduced Data Redundancy**
- The DBMS approach minimizes data redundancy by storing data in a single location, ensuring that there is only one version of the data.

- **Improved Data Consistency**
- Data consistency is maintained by ensuring that any updates to the data are immediately reflected throughout the database.

- **Enhanced Data Access**
- DBMS provides powerful query languages and tools that enable users to retrieve.

- **Increased Productivity**
- By automating routine tasks and providing powerful data management tools, DBMS increases the productivity of database users and administrators.

Advantage	Description
Improved Data Sharing	Enables multiple users to access and share data simultaneously, promoting collaboration.
Enhanced Data Security	Provides mechanisms for controlled access, user authentication, authorization, and data encryption.
Better Data Integration	Centralizes data storage, integrating data from various sources for comprehensive analysis.
Reduced Data Redundancy	Minimizes duplicate data entries by storing data in a single location.
Improved Data Consistency	Ensures data accuracy and consistency through integrity constraints and transaction management.
Enhanced Data Access	Offers powerful query languages (e.g., SQL) and user-friendly interfaces for efficient data retrieval.
Increased Productivity	Automates routine data management tasks, enhancing user efficiency and freeing up time for strategic activities.
Better Decision Making	Provides access to accurate, up-to-date data, enabling better-informed decisions and comprehensive analysis.

Fig 1.4 Advantages of DBMS Concept

1.4.2 Disadvantages of DBMS

While the Database Management System (DBMS) approach offers numerous benefits, it also comes with certain disadvantages. Understanding these drawbacks is essential for making informed decisions about the adoption and implementation of DBMS solutions.

❖ **Complexity**
- System Complexity
- The design and implementation of a DBMS involve complex software and hardware components. This complexity can lead to longer development times and higher costs.
  - Maintenance Complexity
- Maintaining a DBMS requires skilled personnel to manage updates, backups, performance tuning, and troubleshooting. This can add to the operational overhead and require continuous investment in training and hiring.

❖ **Cost**
- High Initial Investment
- Implementing a DBMS involves significant initial costs, including purchasing software licenses, hardware, and additional resources for setup and integration.
- Ongoing Costs
- The ongoing expenses associated with a DBMS include maintenance, upgrades, technical support, and staff salaries. These costs can be substantial, especially for large-scale databases.

❖ **Performance**
- Performance Overheads
- While DBMSs are designed for efficiency, they can introduce performance overheads, particularly for complex queries and large datasets. These overheads may impact system responsiveness and user experience.
- Resource Intensive
- DBMSs often require significant system resources (CPU, memory, disk space) to operate effectively. This can strain existing infrastructure and necessitate additional investment in hardware.

❖ **Vulnerability to Failure**
- Single Point of Failure
- Centralized databases can become single points of failure. If the DBMS or the server hosting it fails, it can lead to significant downtime and loss of access to critical data.
- Backup and Recovery Challenges

- While DBMSs provide backup and recovery mechanisms, implementing and managing these systems can be challenging. Inadequate backup strategies can lead to data loss in the event of system failures or disasters.

❖ **Security Risks**
- Target for Attacks
- Databases are prime targets for cyber-attacks due to the valuable information they hold. A successful breach can lead to severe consequences, including data theft and financial loss.
- Complexity of Security Management
- Managing security within a DBMS involves implementing various controls, such as user authentication, authorization, and encryption. This complexity can lead to potential vulnerabilities if not handled correctly.

❖ **Vendor Dependence**
- Proprietary Systems
- Many DBMS solutions are proprietary, leading to vendor lock-in. Organizations may find it challenging to switch vendors or migrate to new systems due to compatibility issues and dependence on specific technologies.
- Limited Flexibility
- Dependence on a single vendor can limit the flexibility to customize or extend the DBMS to meet specific organizational needs, potentially stifling innovation and adaptability.

❖ **Data Migration Issues**
- Complexity of Migration
- Migrating data from legacy systems or between different DBMSs can be complex and time-consuming. It requires careful planning and execution to ensure data integrity and consistency.
- Risk of Data Loss
- During migration processes, there is a risk of data loss or corruption. Ensuring a smooth and error-free migration necessitates thorough testing and validation.

- While the DBMS approach offers numerous advantages in terms of data management, security, and accessibility, it is essential to consider the associated disadvantages. The complexity, cost, performance issues, vulnerability to failure, security risks, vendor dependence, and data migration challenges must be weighed carefully. By understanding these drawbacks, organizations can make more informed decisions about the adoption and implementation of DBMS solutions, ensuring they meet their specific needs and constraints.

1.5. APPLICATIONS OF DBMS

The applications of DBMS are vast and varied, spanning across different sectors such as banking, airlines, telecommunications, education, healthcare, retail, government, manufacturing, finance, and social media. Each example illustrates how DBMS is used to manage and optimize data handling in these industries.
Here is a list of common applications of DBMS along with examples for each:

❖ **Banking**

Example:
Application: Customer Information Management
Example DBMS: Oracle Database
Description: Used by banks to manage customer accounts, transaction records, and loan information.

❖ **Airlines**
Example:
Application: Flight Reservations
Example DBMS: MySQL
Description: Used by airlines to handle flight schedules, bookings, and cancellations.

❖ **Telecommunications**
Example:
Application: Call Records
Example DBMS: IBM Db2
Description: Used to store and manage call detail records (CDRs), billing information, and customer data.

❖ **Education**
Example:
Application: Student Information Systems
Example DBMS: PostgreSQL
Description: Used by educational institutions to manage student records, enrollment details, grades, and attendance.

❖ **Healthcare**
Example:
Application: Patient Records
Example DBMS: Microsoft SQL Server
Description: Used to store patient information, medical histories, treatment plans, and appointment schedules.

❖ **Retail**
Example:
Application: Inventory Management
Example DBMS: SAP HANA
Description: Used by retail businesses to track stock levels, manage orders, and automate restocking processes.

❖ **Government**
Example:
Application: Public Records Management
Example DBMS: Oracle Database
Description: Used to manage citizen information, property records, tax details, and other public records.

❖ **Manufacturing**
Example:
Application: Supply Chain Management
Example DBMS: SAP HANA
Description: Used to manage supplier information, procurement processes, inventory levels, and production planning.

❖ **Finance**
Example:
Application: Financial Transactions Management
Example DBMS: IBM Db2
Description: Used by financial institutions to manage transaction records, account balances, and investment portfolios.

❖ **Social Media**
Example:
Application: User Data Management
Example DBMS: Cassandra
Description: Used by social media platforms to store user profiles, posts, messages, and interaction data.

1.6 SUMMARY

Databases and their users form the backbone of modern information systems, playing a critical role in managing and organizing vast amounts of data efficiently. By leveraging the powerful capabilities of Database Management Systems (DBMS), organizations can ensure data integrity, security, and accessibility, which are essential for informed decision-making and operational efficiency. Understanding the various types of database users and the roles they play, from administrators and designers to end-users and behind-the-scenes workers, provides a comprehensive insight into the dynamic and interconnected world of databases. This foundational knowledge underscores the importance of DBMS in today's data-driven landscape and its impact on diverse industries. The chapter discussed Characteristics of the Database Approach, Actors on the Scene, Workers behind the scene, Advantages ,disadvantages of the using the DBMS Approach and applications with example.

1.7 TECHNICAL TERMS

DBMS, Database User, System Administrator, End User, Reliability, Security, Privacy, Banking, Hospital, Airline and etc.

1.8 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Illustrate about characteristics of DBMS.
2. Describe about applications of DBMS
3. Explain about advantages and disadvantages of DBMS

Short Notes:

1. Write about Database users
2. Define DBMS.
3. List out benefits of DBMS.

1.9 SUGGESTED READINGS

1. "Database System Concepts" by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
2. "Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe
3. "Database Management Systems" by Raghu Ramakrishnan and Johannes Gehrke
4. "An Introduction to Database Systems" by C.J. Date
5. "SQL and Relational Theory: How to Write Accurate SQL Code" by C.J. Date

Dr. Neelima Guntupalli

LESSON- 02

DATABASE SYSTEM CONCEPTS

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Database system concepts. The chapter began with understanding of Data Models, Schemas and Instances, Data Independence, Database Languages and Interfaces. After completing this chapter, the student will understand the complete idea about Database system concepts.

STRUCTURE

2.1. INTRODUCTION

Databases are integral to modern information systems, providing structured ways to store, retrieve, and manage data. A database is a collection of related data organized to be easily accessed, managed, and updated. Databases support a wide range of applications, from small personal projects to vast enterprise systems. This chapter explores the fundamentals of databases, the database management system (DBMS) approach, and the various roles involved in managing and using databases.

2.2 DATA MODELS

Data models are abstract frameworks that describe the structure, manipulation, and integrity of data stored in a database. They are essential for defining how data is stored, connected, and accessed. Data models are fundamental components in the design and implementation of a Database Management System (DBMS). They provide a systematic way to define and structure data, relationships, and constraints.

Here are the primary data models used in DBMS:

2.2.1 Hierarchical Data Model

- Organizes data in a tree-like structure with parent-child relationships.
- Example: File systems, early IBM mainframe databases.

Features:

- Data is represented in a hierarchy.
- Relationships are one-to-many.

Advantages:

- Simple to design and understand.
- Efficient for queries that follow the hierarchical path.

Disadvantages:

- Inflexible: difficult to re-organize and expand.
- Redundancy: requires duplication of data.

2.2.2 Network Data Model

- Represents data with records and relationships using a graph structure.
- Example: IDMS (Integrated Database Management System).

Features:

- Data is represented using records and relationships.
- Relationships are many-to-many.

Advantages:

- More flexible than the hierarchical model.
- Can handle more complex relationships.

Disadvantages:

- Complexity in design and maintenance.
- Navigation can be cumbersome.

2.2.3 Relational Data Model

- Uses tables (relations) to represent data and their relationships.
- Example: MySQL, PostgreSQL.

Features:

- Data is organized into tables with rows (tuples) and columns (attributes).
- Tables can be linked using keys (primary key, foreign key).

Advantages:
- Flexibility in query and data manipulation.
- Data integrity and normalization to reduce redundancy.
- Standardized query language (SQL).

Disadvantages:
- Performance issues with very large databases.
- Complex joins can be computationally expensive.

2.2.4 Object-Oriented Data Model
- Integrates object-oriented programming principles with database technology.
- Example: ObjectDB, db4o.

Features:
- Data is represented as objects with attributes and methods.
- Supports inheritance, polymorphism, and encapsulation.

Advantages:
- Seamless integration with object-oriented programming languages.
- Capable of handling complex data types and relationships.

Disadvantages:
- Complexity in design and implementation.
- Less mature than relational databases in terms of tools and support.

2.2.5 Entity-Relationship Model
- Uses entities and relationships to model data, focusing on the logical structure.
- Example: Used in database design phase to create ER diagrams.

Features:
- Entities represent objects or things in the real world.
- Attributes are properties of entities.
- Relationships represent associations between entities.

Advantages:
- Provides a clear and structured way to design databases.
- Facilitates the transition from conceptual design to logical and physical design.

Disadvantages:
- Primarily a design tool, not used directly for database implementation.

Understanding different data models is crucial for designing effective databases. Each model offers unique advantages and is suitable for specific applications and use cases. The hierarchical and network models are useful for specific legacy applications, while the relational model remains the most widely used due to its flexibility and robustness. The object-oriented model is ideal for applications requiring complex data representations, and the entity-relationship model is essential for conceptual database design. Selecting the appropriate data model is a fundamental step in ensuring efficient data management and retrieval in any DBMS.

2.3 SCHEMAS AND INSTANCES

In the context of Database Management Systems (DBMS), schemas and instances play crucial roles in defining and managing the structure and content of databases. Understanding these concepts is essential for database design and management.

2.3.1 Schema

- A schema is the logical structure that defines the organization of data in a database. It describes how data is organized and how the relationships among data are associated.
- The overall logical structure of the database, defined during the design phase.

Types of Schemas:

- **Physical Schema:** Defines how data is physically stored in the database. It deals with storage devices, file structures, and indexes.
- **Logical Schema:** Describes the logical structure of the entire database. It includes tables, views, and integrity constraints.
- **View Schema:** Defines how data is presented to different users. It can include subsets of data from the logical schema.

Characteristics:

- Static: Schemas are typically defined at the design phase and do not change frequently.
- Blueprint: Schemas serve as blueprints for the database structure and dictate how data is organized and accessed.

Example:

- A logical schema might define a database with tables such as Customers, Orders, and Products, specifying their attributes and relationships.

```sql
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100)
);

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

Fig Example of Schemas : Customers and Orders

2.3.2 Instance

- An instance is the actual data stored in the database at a particular moment in time. It represents a snapshot of the database's content.
- The actual data stored in the database at a particular moment in time.

Characteristics:

- Dynamic: Instances change frequently as data is inserted, updated, and deleted.
- Data Content: Instances reflect the current state of the data within the schema's structure.

Example:

- If a table named Customers is defined in the schema, an instance would include the actual rows of data in that table at any given time.

```
INSERT INTO Customers (CustomerID, Name, Email)
VALUES (1, 'John Doe', 'john.doe@example.com');
INSERT INTO Orders (OrderID, OrderDate, CustomerID)
VALUES (101, '2024-07-21', 1);
```

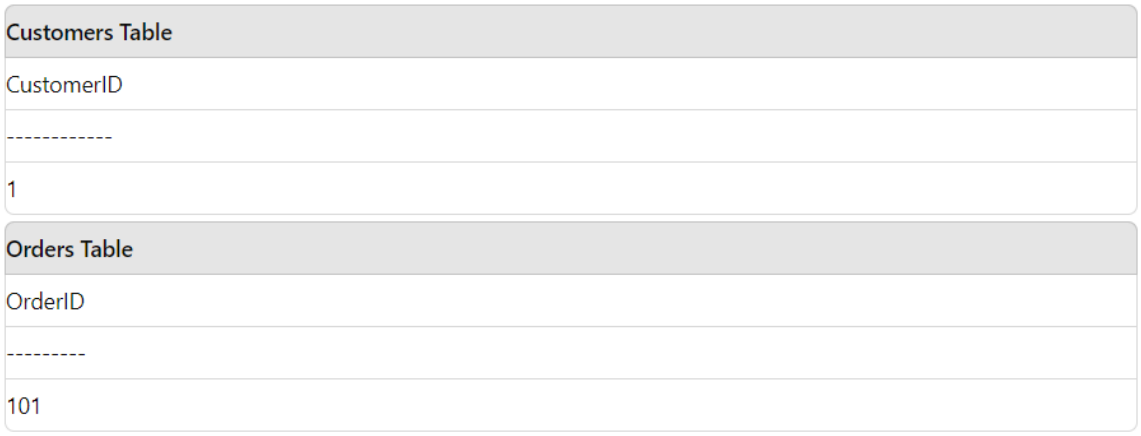


Fig The result of Insert Query

2.3. 3 Schema vs. Instance

Schema:

- **Static:** Schemas are typically static and change infrequently.
- **Blueprint:** Serves as a blueprint or framework for organizing data.
- **Definition:** Includes definitions of tables, fields, data types, relationships, views, and constraints.
- **Levels:** Can be divided into physical schema, logical schema, and view schema.

Instance:

- **Dynamic:** Instances are dynamic and change with database operations.
- **Snapshot:** Represents a snapshot of the database at a specific point in time.
- **Data:** Contains actual data entries, reflecting the current state of the database.
- **Temporal:** Can vary from one moment to the next based on data operations.

Aspect	Schema	Instance
Nature	Static, rarely changes	Dynamic, frequently changes
Role	Defines structure and organization of the database	Represents the actual data in the database at a given time
Content	Tables, fields, data types, relationships, views, constraints	Actual data entries, records in tables
Analogy	Blueprint of a building	The actual building with its current occupants and furniture
Examples	Table definitions, constraints, view definitions	Rows in a table, current data in the database

Fig Schema vs. Instance

2.4 THREE-SCHEMA ARCHITECTURE AND DATA INDEPENDENCE

The three-schema architecture is designed to separate the user applications from the physical database.

2.4.1 Three-Schema Architecture

The Three-Schema Architecture is a framework used in Database Management Systems (DBMS) to separate the user applications from the physical database. This separation provides a way to manage the complexity of data and ensures data independence. The architecture is divided into three levels: the internal schema, the conceptual schema, and the external schema.

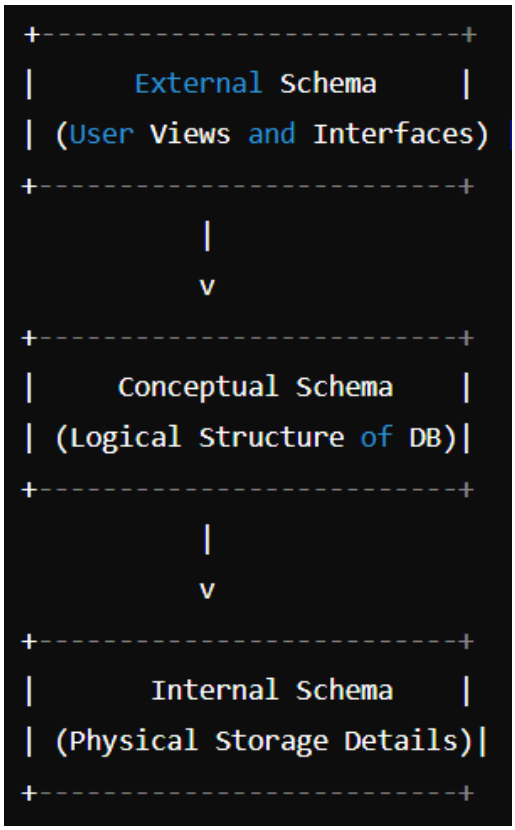


Fig Three-tier Architecture

❖ **Internal Schema**

Description:

- The internal schema defines the physical storage structure of the database. It describes how data is stored in the database and includes data structures, indexing methods, and file organization techniques.

Characteristics:

- **Storage Details:** Includes details about physical storage, such as data files, indexes, and data blocks.
- **Optimization:** Focuses on optimizing storage and access speed.
- **Data Independence:** Provides physical data independence by allowing changes to the internal schema without affecting the conceptual schema.

Example:

- A table may be stored as a B-tree index for efficient retrieval.

❖ **Conceptual Schema**

Description:

- The conceptual schema provides a unified and logical view of the entire database. It describes the structure of the whole database for a community of users, hiding the details of physical storage.

Characteristics:

- **Unified View:** Represents all entities, relationships, and constraints.
- **Logical Structure:** Independent of how data is physically stored.
- **Data Independence:** Provides logical data independence by allowing changes to the conceptual schema without affecting the external schemas.

Example:

- Defines entities like Customers and Orders, their attributes, and relationships between them

```
CREATE TABLE Customers
  (CustomerID INT PRIMARY KEY,
  Name VARCHAR(100),
  Email VARCHAR(100));
CREATE TABLE Orders
  (OrderID INT PRIMARY KEY,
  OrderDate DATE,
  CustomerID INT,
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID));
```

External Schema

Description:

- The external schema, also known as the view level, defines how individual users or user groups interact with the database. It provides a customized view of the database tailored to the needs of different users.

Characteristics:

- **User Views:** Can have multiple external schemas, each tailored to different user requirements.

- **Security:** Helps in providing different access levels to different users.
- **Simplified Interaction:** Allows users to interact with the database without needing to know its complete structure.

Example:

- An external schema for a salesperson might include only the customer names and contact information.
  CREATE VIEW Salesperson View AS
  SELECT Name, Email
  FROM Customers;

Benefits of Three-Schema Architecture

1. **Data Abstraction:**
- Separates the user applications from the physical data storage, providing a higher level of abstraction and simplifying database management.
2. **Data Independence:**
- Enhances both logical and physical data independence, making the database system more flexible and easier to maintain.
3. **Security:**
- Provides a mechanism to define multiple user views, enhancing data security by restricting access to sensitive data.
4. **Consistency:**
- Ensures that different user views are consistent with the overall conceptual schema, maintaining data integrity.

The Three-Schema Architecture is a powerful framework in DBMS that provides a structured approach to data abstraction, independence, and security. By separating the internal, conceptual, and external schemas, it allows for more flexible, efficient, and secure database management. Understanding and implementing this architecture is crucial for designing robust and scalable database systems.

2.4.2 Data Independence

Data independence is a key concept in the realm of Database Management Systems (DBMS). It refers to the capacity to change the schema at one level of the database system without necessitating changes to the schema at the next higher level. This concept is pivotal in ensuring that the database system remains flexible and manageable over time.

Types of Data Independence

Data independence is broadly categorized into two types: logical data independence and physical data independence.

- **Logical Data Independence:** Ability to change the conceptual schema without altering the external schemas.

Examples of Changes:

- Adding or removing a new attribute (column) in a table.
- Changing the data type of an existing attribute.
- Merging two records into one or splitting one record into two.
- Adding new relationships or altering existing relationships between tables.

Importance:

- Enhances flexibility and adaptability of the database.
- Ensures that application programs do not need to be rewritten when changes are made to the logical structure of the database.

Example:

```
CREATE TABLE Customers
  (CustomerID INT PRIMARY KEY,
  Name VARCHAR(100),
  Email VARCHAR(100));
```

If we decide to add a new attribute Phone Number, logical data independence ensures that user applications accessing Customers table don't need to change:

```
ALTER TABLE Customers ADD Phone Number VARCHAR(15);
```

❖ **Physical Data Independence:** Physical data independence is the ability to change the internal schema without needing to alter the conceptual schema. This means that changes to the physical storage of data do not impact the logical structure or the applications that interact with the database.

Examples of Changes:

- Changing the file organization or storage structures.
- Using different storage devices.
- Adding or modifying indexes to improve performance.
- Changing the data compression techniques or storage paths.

Importance:

- Provides a layer of abstraction between the physical storage and the logical structure.
- Allows for performance tuning and optimization without affecting the logical data model or the applications.

Example:

- Suppose we want to improve the performance of a Customers table by adding an index on the Email column:

```
CREATE INDEX idx_email ON Customers(Email);
```

Physical data independence ensures that this change does not affect the logical view or the applications accessing the Customers table.

Data independence is a foundational principle in the design and management of DBMS, ensuring that databases remain flexible, manageable, and adaptable to changing requirements. By separating the logical and physical aspects of the database, data independence allows for efficient updates and maintenance, enhancing the overall robustness and functionality of the

database system. Understanding and implementing data independence is crucial for database administrators and developers to create resilient and scalable database environments.

2.5 DATABASE LANGUAGES AND INTERFACES

2.5.1 Database Languages

Database systems support various languages and interfaces for defining, manipulating, and querying data. Database languages are specialized languages used to define, manipulate, control, and manage data in a database. Each type of database language serves a specific purpose in the database management process. The primary categories of database languages include Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), and Transaction Control Language (TCL).

❖ **Data Definition Language (DDL)**
- Used to define database schemas.
- Example: CREATE TABLE, ALTER TABLE.

```
CREATE TABLE Customers
  (CustomerID INT PRIMARY KEY,
   Name VARCHAR(100),
   Email VARCHAR(100));

ALTER TABLE Customers ADD Phone Number VARCHAR(15);
DROP TABLE Customers;
TRUNCATE TABLE Customers;
```

Establishes the framework and structure of the database, enabling efficient data storage and retrieval.

❖ **Data Manipulation Language (DML)**
- Used for data manipulation.
- Example: SELECT, INSERT, UPDATE, DELETE.

```
SELECT * FROM Customers;
INSERT INTO Customers (CustomerID, Name, Email) VALUES (1, 'John Doe', 'john.doe@example.com');
UPDATE Customers SET Email = 'john.new@example.com' WHERE CustomerID = 1;
DELETE FROM Customers WHERE CustomerID = 1;
```

Facilitates the manipulation and management of data, allowing users to perform various operations on the stored data.

❖ **Data Control Language (DCL)**
- Used to control access to data.
- Example: GRANT, REVOKE.

```
GRANT SELECT ON Customers TO user1;
REVOKE SELECT ON Customers FROM user1;
```

Ensures data security and integrity by managing user permissions and access levels.

❖ **Transaction Control Language (TCL)**
- Used to manage transactions.
- Example: COMMIT, ROLLBACK.

2.5.2 Interfaces

Database Management Systems (DBMS) offer various interfaces that allow users to interact with the database. These interfaces are designed to cater to different user requirements, ranging from database administrators and developers to end-users and application programs.

Here are the primary types of interfaces provided by DBMS:

❖ **Command-Line Interface (CLI):**
A Command-Line Interface allows users to interact with the DBMS by typing commands in a text-based environment. This interface is powerful for experienced users who need precise control over database operations. Text-based interaction with the DBMS.

Features:
- Direct command execution.
- Scripting capabilities for automated tasks.
- Access to all DBMS functionalities.

Example:

```
mysql> SELECT * FROM Customers;
```

❖ **Graphical User Interface (GUI):**
A Graphical User Interface provides a visual and user-friendly way to interact with the DBMS. It uses graphical elements such as windows, icons, and menus to simplify database operations.

Features:
- Visual representation of database schema.
- Drag-and-drop functionalities.
- Wizards and tools for database design, query building, and data management.

Example:
- Tools like MySQL Workbench, Microsoft SQL Server Management Studio (SSMS), and Oracle SQL Developer.

❖ **Application Program Interface (API):**

An API allows applications to interact with the DBMS programmatically. It provides a set of functions and protocols for accessing and manipulating the database.

Features:
- Language-specific libraries and drivers (e.g., JDBC for Java, ODBC for multiple languages).
- Seamless integration with applications.
- Support for various database operations like querying, updating, and transaction management.

Example:

- Using Python's SQLite3 library:

```
import sqlite3

conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute('SELECT * FROM Customers')
rows = c.fetchall()
for row in rows:
    print(row)
conn.close()
```

Programming interfaces for database interaction.

❖ **Natural Language Interface:**

A natural language interface allows users to interact with the DBMS using natural language queries. This interface aims to make database interactions more intuitive and accessible to non-technical users.

Features:

- Natural language processing to interpret user queries.
- Conversational interaction style.
- Integration with virtual assistants and chatbots.

2.6 SUMMARY

In this chapter, we explored the fundamental concepts of data modeling and database structures, which serve as the foundation for understanding how data is logically represented and organized in a database system. We discussed several types of data models, including the Hierarchical Model, which organizes data in a tree-like structure; the Network Model, which allows complex many-to-many relationships; the Relational Model, which represents data in tables and uses keys to establish relationships; and the Object-Oriented Model, which integrates object-oriented programming concepts into databases. Additionally, the Entity-Relationship (ER) Model was introduced as a high-level conceptual model used for database design through entities, attributes, and relationships.

The chapter also distinguished between schemas and instances, where the schema represents the database structure (blueprint), and the instance represents the actual data stored at a given time. Furthermore, we reviewed database languages such as DDL (Data Definition Language), DML (Data Manipulation Language), and DCL (Data Control Language), along with various user interfaces like graphical, form-based, and natural language interfaces that facilitate user interaction with databases. Overall, this chapter provides a clear understanding of how data is modeled, stored, accessed, and managed within modern database systems, forming a conceptual bridge between real-world entities and their digital representation.

2.7 TECHNICAL TERMS

1. Data Model
2. Hierarchical Data Model
3. Network Data Model
4. Relational Data Model
5. Object-Oriented Data Model
6. Entity-Relationship (ER) Model
7. Schema
8. Instance
9. Data Independence
10. Conceptual Schema

2.8 SELF-ASSESSMENT QUESTIONS

Essay Questions

1. Explain different types of data models with suitable examples.
2. Compare and contrast the hierarchical, network, and relational data models.
3. Describe the components of an Entity-Relationship (ER) model.
4. Discuss the difference between schema and instance with suitable examples.
5. Explain the various types of database languages and their purposes.

Short Questions

1. What is a data model?
2. List any four types of data models.
3. What is the difference between hierarchical and network data models?
4. Define the relational data model.
5. What are the main features of an object-oriented data model?
6. What is an entity in the ER model?

2.9 SUGGESTED READINGS

1. Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, 5th Edition, Pearson Education, 2007.
2. Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, *Database System Concepts*, 6th Edition, McGraw Hill, 2011.
3. C. J. Date, *An Introduction to Database Systems*, 8th Edition, Addison Wesley, 2003.
4. Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*, 3rd Edition, McGraw Hill, 2003.
5. Peter Rob and Carlos Coronel, *Database Systems: Design, Implementation, and Management*, Cengage Learning, 2009.

Dr. Neelima Guntupalli

LESSON- 03

DATABASE ARCHITECTURE

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Database Architecture. The chapter began with understanding of Three Schema architecture. Later understand about Centralized and Client/Server Architecture for DBMS, Classification of Database Management Systems. After completing this chapter, the student will understand the complete idea about Database architecture.

STRUCTURE

3.1. INTRODUCTION

Databases are integral to modern information systems, providing structured ways to store, retrieve, and manage data. A database is a collection of related data organized to be easily accessed, managed, and updated. Databases support a wide range of applications, from small personal projects to vast enterprise systems. This chapter explores the fundamentals of databases, the database management system (DBMS) approach, and the various roles involved in managing and using databases.

The chapter first covered the Characteristics of the Database Approach, Actors on the Scene, Workers behind the scene, Advantages of the using the DBMS Approach and etc.

3.1.1 THREE-SCHEMA ARCHITECTURE

The Three-Schema Architecture is a framework used in Database Management Systems (DBMS) to separate the user applications from the physical database. This separation provides a way to manage the complexity of data and ensures data independence. The architecture is divided into three levels: the internal schema, the conceptual schema, and the external schema.

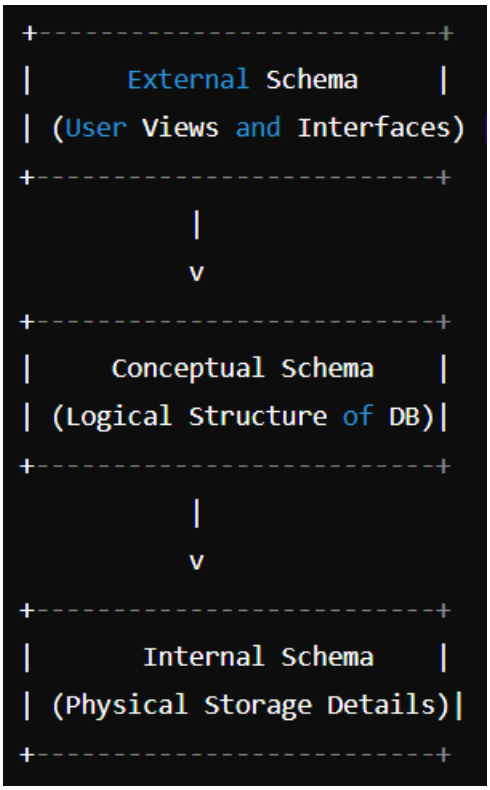


Fig 3.1 Three-tier Architecture

- **Presentation Tier:**
  This is the topmost layer that interacts with the user. It displays information and collects input using interfaces like web pages, mobile apps, or forms.
- **Application / Logic Tier:**
  This middle layer contains the application's business logic and processing rules. It acts as a mediator between the user interface and the database, sending queries and processing results.
- **Data Tier:**
  The lowest layer consists of the database and DBMS where actual data is stored, managed, and retrieved. It ensures data consistency, integrity, and security.

In a university database system:

- The **Presentation Tier** is the student portal (browser interface).
- The **Application Tier** is the web server handling registration and grade processing.
- The **Data Tier** is the database storing student and course information.

This **three-tier architecture** enables easy updates, secure data access, and efficient distribution of workload between client and server components.

❖ **Internal Schema**

Description:

- The internal schema defines the physical storage structure of the database. It describes how data is stored in the database and includes data structures, indexing methods, and file organization techniques.

Characteristics:

- **Storage Details:** Includes details about physical storage, such as data files, indexes, and data blocks.
- **Optimization:** Focuses on optimizing storage and access speed.
- **Data Independence:** Provides physical data independence by allowing changes to the internal schema without affecting the conceptual schema.

Example:

- A table may be stored as a B-tree index for efficient retrieval.

❖ **Conceptual Schema**

Description:

- The conceptual schema provides a unified and logical view of the entire database. It describes the structure of the whole database for a community of users, hiding the details of physical storage.

Characteristics:

- **Unified View:** Represents all entities, relationships, and constraints.
- **Logical Structure:** Independent of how data is physically stored.
- **Data Independence:** Provides logical data independence by allowing changes to the conceptual schema without affecting the external schemas.

Example:

- Defines entities like Customers and Orders, their attributes, and relationships between them

```
CREATE TABLE Customers (
   CustomerID INT PRIMARY KEY,
   Name VARCHAR(100),
   Email VARCHAR(100)
);
CREATE TABLE Orders (
   OrderID INT PRIMARY KEY,
   OrderDate DATE,
   CustomerID INT,
   FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

❖ **External Schema**

Description:

- The external schema, also known as the view level, defines how individual users or user groups interact with the database. It provides a customized view of the database tailored to the needs of different users.

Characteristics:

- **User Views:** Can have multiple external schemas, each tailored to different user requirements.
- **Security:** Helps in providing different access levels to different users.
- **Simplified Interaction:** Allows users to interact with the database without needing to know its complete structure.

Example:

- An external schema for a salesperson might include only the customer names and contact information.

```
CREATE VIEW SalespersonView AS
SELECT Name, Email
FROM Customers;
```

Benefits of Three-Schema Architecture

1. **Data Abstraction:**
   - Separates the user applications from the physical data storage, providing a higher level of abstraction and simplifying database management.
2. **Data Independence:**
   - Enhances both logical and physical data independence, making the database system more flexible and easier to maintain.
3. **Security:**
   - Provides a mechanism to define multiple user views, enhancing data security by restricting access to sensitive data.
4. **Consistency:**
   - Ensures that different user views are consistent with the overall conceptual schema, maintaining data integrity.

The Three-Schema Architecture is a powerful framework in DBMS that provides a structured approach to data abstraction, independence, and security. By separating the internal, conceptual, and external schemas, it allows for more flexible, efficient, and secure database management. Understanding and implementing this architecture is crucial for designing robust and scalable database systems.

3.2.2 Data Independence

Data independence is a key concept in the realm of Database Management Systems (DBMS). It refers to the capacity to change the schema at one level of the database system without necessitating changes to the schema at the next higher level. This concept is pivotal in ensuring that the database system remains flexible and manageable over time.

Types of Data Independence

Data independence is broadly categorized into two types: logical data independence and physical data independence.

❖ **Logical Data Independence:** Ability to change the conceptual schema without altering the external schemas.

Examples of Changes:

- Adding or removing a new attribute (column) in a table.
- Changing the data type of an existing attribute.
- Merging two records into one or splitting one record into two.
- Adding new relationships or altering existing relationships between tables.

Importance:

- Enhances flexibility and adaptability of the database.
- Ensures that application programs do not need to be rewritten when changes are made to the logical structure of the database.

Example:

```
CREATE TABLE Customers (
   CustomerID INT PRIMARY KEY,
   Name VARCHAR(100),
   Email VARCHAR(100)
);
```

If we decide to add a new attribute PhoneNumber, logical data independence ensures that user applications accessing Customers table don't need to change:

```
ALTER TABLE Customers ADD PhoneNumber VARCHAR(15);
```

❖ **Physical Data Independence:** Physical data independence is the ability to change the internal schema without needing to alter the conceptual schema. This means that changes to the physical storage of data do not impact the logical structure or the applications that interact with the database.

Examples of Changes:

- Changing the file organization or storage structures.
- Using different storage devices.
- Adding or modifying indexes to improve performance.
- Changing the data compression techniques or storage paths.

Importance:

- Provides a layer of abstraction between the physical storage and the logical structure.
- Allows for performance tuning and optimization without affecting the logical data model or the applications.

Example:

- Suppose we want to improve the performance of a Customers table by adding an index on the Email column:

CREATE INDEX idx_email ON Customers(Email);
Physical data independence ensures that this change does not affect the logical view or the applications accessing the Customers table.

Data independence is a foundational principle in the design and management of DBMS, ensuring that databases remain flexible, manageable, and adaptable to changing requirements. By separating the logical and physical aspects of the database, data independence allows for efficient updates and maintenance, enhancing the overall robustness and functionality of the database system. Understanding and implementing data independence is crucial for database administrators and developers to create resilient and scalable database environments.

3.3 CENTRALIZED AND CLIENT/SERVER ARCHITECTURE FOR DBMS

3.3.1 Centralized Architecture

Centralized architecture in Database Management Systems (DBMS) refers to a system where all database functionalities, including storage, processing, and management, are performed on a single central server. This server is responsible for handling all database requests and operations, serving as the main point of access for all users and applications.

Key Characteristics

1. **Single Server System:**
   - All data storage and processing are managed by one central server.
   - The central server handles all database operations, including querying, updating, and managing transactions.
2. **Centralized Control:**
   - Database administration and management tasks are centralized, simplifying maintenance and oversight.
   - Consistent enforcement of security policies and data integrity rules.
3. **Unified Data Storage:**
   - All data is stored in a single location, reducing redundancy and ensuring consistency.
   - Simplifies backup and recovery processes.
4. **Direct User Interaction:**
   - Users and applications interact directly with the central server for all database operations.
   - Simplifies the client-side configuration as there is only one server to connect to.

Advantages

1. **Simplified Management:**
   - Easier to manage and maintain as all database operations are centralized.
   - Simplified backup, recovery, and security management.
2. **Consistent Performance:**
   - Predictable performance characteristics as all operations are handled by a single server.

- Easier to monitor and optimize performance centrally.

3. **Reduced Data Redundancy:**
   - Single storage location reduces data duplication and ensures data consistency.
   - Easier to enforce data integrity and validation rules.
4. **Enhanced Security:**
   - Centralized control makes it easier to implement and manage security policies.
   - Simplifies access control and auditing.

Disadvantages

1. **Scalability Limitations:**
   - Limited by the capacity of the central server, making it challenging to scale as data volume and user load increase.
   - Upgrading the central server can be costly and disruptive.
2. **Single Point of Failure:**
   - The central server is a single point of failure; if it goes down, the entire database system becomes unavailable.
   - Requires robust backup and disaster recovery plans.
3. **Performance Bottlenecks:**
   - High demand on the central server can lead to performance bottlenecks.
   - All user requests must be processed by the central server, potentially leading to congestion and delays.
4. **Geographical Limitations:**
   - Users located far from the central server may experience latency issues.
   - May not be suitable for applications requiring high-speed access from multiple geographic locations.

Example:

Consider a **university management system** where all student records, faculty data, course schedules, and examination results are stored in a **single central server** located in the university's data center. All administrative departments (like Admissions, Accounts, and Examination Cell) access the same database through connected terminals.

- If the Admissions Office updates a student's record, the same change is instantly visible to the Accounts or Examination Cell.
- The **central server** ensures that all users work with the same, up-to-date information.
- However, if the server goes down, all users lose access, which is a major limitation of centralized systems.

Advantages:

- Easier database management and maintenance.
- High data consistency and security.
- Centralized backup and recovery control.

Disadvantages:

- Single point of failure (server outage affects all users).
- Scalability issues with many simultaneous users.
- Performance depends on network and server capacity.

Centralized architecture in DBMS offers a simplified approach to database management, with all operations controlled by a single central server. This architecture is beneficial for small to medium-sized organizations or applications with moderate performance and scalability needs. However, it comes with limitations in terms of scalability, potential performance bottlenecks, and a single point of failure. Understanding the advantages and disadvantages of centralized architecture helps in determining its suitability for specific applications and organizational needs.

3.3.2 Client/Server Architecture

Client/Server architecture in Database Management Systems (DBMS) is a distributed application structure that partitions tasks between clients, which request services, and servers, which provide those services. This architecture enhances the efficiency, scalability, and manageability of database systems by distributing the workload across multiple machines.

Key Characteristics

1. **Two-Tier Architecture:**
   - **Client Tier:** Consists of client machines that run applications and user interfaces. Clients send requests to the server and present the results to the user.
   - **Server Tier:** Consists of a central server that processes requests from clients, performs database operations, and manages data storage.
2. **Three-Tier Architecture (Enhanced Client/Server):**
   - **Presentation Tier:** The client-side interface where users interact with the application.
   - **Application Logic Tier:** The middle layer that processes business logic and communicates between the presentation and data tiers.
   - **Data Tier:** The server-side where the database is hosted and managed.
3. **Distributed Processing:**
   - Workload is distributed between client and server, optimizing performance and resource utilization.
   - Clients handle presentation logic, while servers handle data processing and management.
4. **Network Communication:**
   - Clients and servers communicate over a network using standardized protocols (e.g., TCP/IP).

Advantages

1. **Scalability:**
   - Easily scalable by adding more clients or servers as needed.
   - Supports large numbers of simultaneous users and high transaction volumes.
2. **Improved Performance:**
   - Distributes processing load between clients and servers, reducing bottlenecks.
   - Clients handle user interfaces and local processing, while servers manage data-intensive tasks.
3. **Flexibility:**
   - Different clients (desktop, web, mobile) can interact with the same server.
   - Servers can be upgraded or replaced independently of clients.

4. **Centralized Data Management:**
   - Centralized control over data ensures consistency and integrity.
   - Easier to implement security, backup, and recovery policies.

Disadvantages

1. **Complexity:**
   - More complex to design, implement, and maintain compared to centralized architectures.
   - Requires robust network infrastructure and management.
2. **Network Dependency:**
   - Performance and reliability depend on the underlying network.
   - Network failures can disrupt access to the database.
3. **Cost:**
   - Higher initial setup and maintenance costs due to the need for multiple servers and network infrastructure.

Example:

Consider a **banking system** that uses a **client/server architecture**.

- The **client-side application** (installed on teller or ATM terminals) allows users to perform operations such as balance inquiry, fund transfer, or cash withdrawal.
- The **server-side system** (at the bank's data center) runs the **DBMS** that stores and processes all account details, transactions, and authentication requests.
- When a teller checks a customer's account, the client sends an SQL query like:
- SELECT balance FROM accounts WHERE account_no = 12345;

The **server** executes this query and returns the result to the client for display.

Advantages:

- Efficient sharing of database resources among multiple users.
- Reduced client workload since data processing occurs on the server.
- Easier maintenance and scalability — new clients can be added without affecting the central database.

Disadvantages:

- Requires reliable network connectivity.
- Security concerns if communication between client and server is not encrypted.
- Higher initial setup and maintenance costs compared to centralized systems.

Client/Server architecture in DBMS offers a robust and scalable solution for managing complex and distributed database systems. By dividing the workload between clients and servers, this architecture enhances performance, flexibility, and centralized data management. While it introduces some complexity and network dependency, the benefits make it suitable for a wide range of applications, especially in large enterprises and web-based environments. Understanding the client/server model is crucial for designing efficient and scalable database solutions.

3.3.3 CLASSIFICATION OF DATABASE MANAGEMENT SYSTEMS

- **Based on Data Model**
  - Relational DBMS (RDBMS): MySQL, PostgreSQL.
  - Object-Oriented DBMS (OODBMS): ObjectDB, db4o.
  - NoSQL DBMS: MongoDB, Cassandra.
- **Based on Number of Users**
  - Single-user DBMS: Microsoft Access.
  - Multi-user DBMS: Oracle, SQL Server.
- **Based on Database Distribution**
  - Centralized DBMS: Data stored in a single location.
  - Distributed DBMS: Data distributed across multiple locations.
  - Federated DBMS: Manages multiple autonomous databases.
- **Based on Cost**
  - Open-source DBMS: MySQL, PostgreSQL.
  - Commercial DBMS: Oracle, SQL Server.
- **Based on Access Method**
  - Navigational DBMS: Uses pointers to navigate between data.
  - SQL DBMS: Uses SQL for data access.

Table 3.1: Classification of Database Management Systems

Classification Criteria	Type of DBMS	Description / Example
Based on Data Model	Relational DBMS (RDBMS)	Stores data in tables with rows and columns. *Examples:* MySQL, PostgreSQL
	Object-Oriented DBMS (OODBMS)	Stores data in the form of objects, as in object-oriented programming. *Examples:* ObjectDB, db4o
	NoSQL DBMS	Designed for unstructured or semi-structured data; schema-free. *Examples:* MongoDB, Cassandra
Based on Number of Users	Single-User DBMS	Supports one user at a time; simpler systems. *Example:* Microsoft Access
	Multi-User DBMS	Allows multiple users to access the database simultaneously. *Examples:* Oracle, SQL Server
Based on Database Distribution	Centralized DBMS	Entire database stored in one central location.
	Distributed DBMS	Database distributed across multiple physical locations connected by a network.
	Federated DBMS	Manages multiple autonomous databases under a single interface.

Based on Cost	Open-Source DBMS	Free to use and modify; community-supported. *Examples:* MySQL, PostgreSQL
	Commercial DBMS	Proprietary and licensed software with vendor support. *Examples:* Oracle, SQL Server
Based on Access Method	Navigational DBMS	Accesses data using pointers or predefined paths.
	SQL DBMS	Uses Structured Query Language (SQL) for defining and manipulating data.

3.4 SUMMARY

In this chapter, we explored the fundamental architectural concepts of Database Management Systems (DBMS). The three-schema architecture provides a clear separation between users' views, the conceptual design, and the physical storage of data. It is organized into three levels — internal, conceptual, and external — which together help to manage complexity and ensure better data abstraction. A key benefit of this architecture is data independence, allowing changes at one level (such as storage structure) without affecting higher levels like application programs or user views. This separation ensures flexibility, maintainability, and scalability in modern database systems.

The chapter also introduced different types of DBMS architectures, including centralized systems, where all data and DBMS software reside on a single server, and client/server systems, which divide tasks between clients (users) and a central server for better efficiency and concurrent access. Finally, we classified DBMSs based on various criteria such as data models (relational, object-oriented, hierarchical, etc.), number of users, and distribution of data. These architectural and classification principles form the foundation for understanding how databases are organized, managed, and accessed in both traditional and modern computing environments.

3.5 TECHNICAL TERMS

Data Models, Schema, Instances, Three schema architecture and Data Independence

3.6 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Illustrate about data models.
2. Describe about Data Independence
3. Explain about three tier architectures of DBMS

Short Notes:

1. Write about Schema and Instances
2. Define Data Interface
3. List out benefits of Clint Server Architecture

3.7 SUGGESTED READINGS

1. "Database System Concepts" by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
2. "Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe
3. "Database Management Systems" by Raghu Ramakrishnan and Johannes Gehrke
4. "An Introduction to Database Systems" by C.J. Date
5. "SQL and Relational Theory: How to Write Accurate SQL Code" by C.J. Date

Dr. Neelima Guntupalli

LESSON- 04

DATA MODELING USING THE ER MODEL

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Data Modeling Using the ER Model. The chapter began with Conceptual Data models, Entity Types, Entity Sets, Attributes and Keys, Relationship types, Relationship sets, roles and structural Constraints, Weak Entity types, Relationship Types of Degree Higher than Two, Refining the ER Design for the COMPANY Database. After completing this chapter, the student will understand Data Modeling Using the ER Model.

4.1 INTRODUCTION

Data modeling is a fundamental step in designing a database. It involves creating a visual representation of the data structures and their relationships, ensuring the database will efficiently support the required data management tasks. One of the most widely used techniques for data modeling is the Entity-Relationship (ER) Model.

The ER Model was introduced by Peter Chen in 1976 and provides a high-level conceptual framework for database design. It uses a diagrammatic approach to represent data entities, their attributes, and the relationships between them. The primary components of the ER Model include entities, attributes, and relationships, each playing a critical role in defining the structure and constraints of the data.

The chapter first covered began with understanding Conceptual Data models, Entity Types, Entity Sets, Attributes and Keys, Relationship types, Relationship sets, roles and structural Constraints, Weak Entity types, Relationship Types of Degree Higher than Two, Refining the ER Design for the COMPANY Database.

4.2 ENTITY TYPES AND ENTITY SETS

Entity Types and Entity Sets are foundational elements in the ER Model for DBMS. Entity Types provide the blueprint for defining the properties and structure of data objects, while Entity Sets represent the actual data instances in the database. By effectively utilizing these concepts, database designers can create structured, efficient, and scalable databases that accurately represent the real-world entities and their relationships. Understanding the distinction between entity types and entity sets is crucial for successful database modeling and implementation. Entities can be tangible, such as 'Customer' or 'Product,' or intangible, such as 'Order' or 'Transaction. Entities are represented by rectangles in ER diagrams. For example, in a library system, entities might include 'Book,' 'Member,' and 'Loan.'

4.2.1 Entity Types

An entity type is a collection of entities that share common properties or characteristics. It represents a category or class of objects in the real world with the same attributes.

Characteristics:

- **Attributes:** Properties that describe the entity type. Each entity within the type will have the same set of attributes, but the attribute values will differ.
- **Primary Key:** An attribute or a set of attributes that uniquely identify each entity in the entity type.
- **Representation:** In an ER diagram, an entity type is represented by a rectangle containing the entity type name.
- **Example:** In a university database, an entity type could be Student, with attributes such as StudentID, Name, DateOfBirth, and Major.

4.2.2 Entity Sets

An entity set is a collection of all entities of a particular entity type at any point in time. It is essentially the table in a relational database where rows represent individual entities.

Characteristics:

- **Homogeneous Collection:** Contains entities of the same entity type.
- **Dynamic:** The number of entities in an entity set can change over time as entities are added, modified, or removed.

Representation:

- In an ER diagram, the entity set is represented by the same rectangle used for the entity type.
- **Example:**
- The Student entity set in a university database would include all current students, each represented by a unique combination of attribute values.

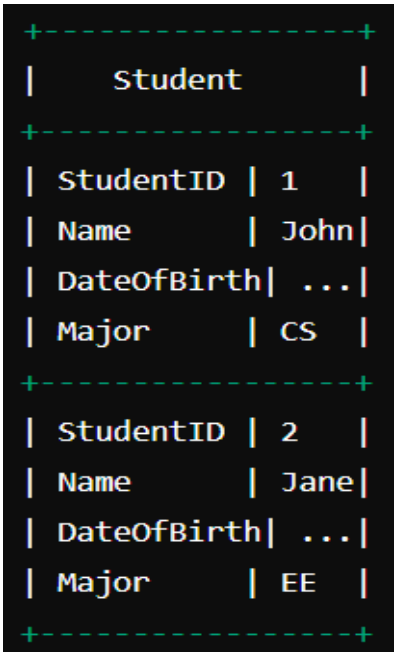


Fig 4.1 Entity Set of Student Object

4.3 ATTRIBUTES AND KEYS

Attributes and keys are fundamental concepts in Database Management Systems (DBMS) that help define and manage the structure of data within a database. Attributes provide the details about the data entities, while keys ensure the uniqueness and establish relationships between the data entities.

4.3.1 Attributes

Attributes are properties or characteristics that describe an entity in a database. Each attribute represents a data field and holds a value for every entity instance.

Types of Attributes:

- **Simple Attribute:** An attribute that cannot be divided into smaller components. For example, FirstName and LastName are simple attributes.

- **Composite Attribute:** An attribute that can be subdivided into smaller components. For example, Address can be subdivided into Street, City, State, and ZIP Code.
- **Single-Valued Attribute:** An attribute that holds a single value for a given entity instance. For example, DateOfBirth is typically single-valued.
- **Multi-Valued Attribute:** An attribute that can hold multiple values for a given entity instance. For example, PhoneNumbers can store multiple phone numbers for a person.
- **Derived Attribute:** An attribute whose value can be derived from other attributes. For example, Age can be derived from the DateOfBirth.
- **Stored Attribute:** An attribute that is stored in the database and not derived from other attributes. For example, EmployeeID.

Example:

- For the entity Student, attributes might include StudentID, Name, DateOfBirth, Address, and PhoneNumbers.

ER Diagram Representation:

- Attributes are represented by ovals connected to their respective entities by lines.

4.3.2 Keys

Keys are special types of attributes or combinations of attributes that are used to uniquely identify records in a table and establish relationships between tables.

Types of Keys:

1. Primary Key:

- **Definition:** A unique attribute or a combination of attributes that uniquely identifies each record in a table.
- **Characteristics:**
  - Must contain unique values.
  - Cannot contain NULL values.
  - There can be only one primary key per table.

Example: StudentID in the Student table.

2. Composite Key:

- **Definition:** A primary key that consists of two or more attributes to uniquely identify a record.
- **Example:** OrderID and ProductID together can form a composite key for an OrderDetails table.

3. Candidate Key:

- **Definition:** An attribute or a set of attributes that can uniquely identify a record and could potentially be chosen as the primary key.
- **Example:** Both Email and PhoneNumber in a Customer table can be candidate keys.

4. Alternate Key:

- **Definition:** A candidate key that is not chosen as the primary key.
- **Example:** If Email is chosen as the primary key, then PhoneNumber would be an alternate key.

5. Foreign Key:

- **Definition:** An attribute or a set of attributes in one table that refers to the primary key in another table to establish a relationship between the two tables.
- **Characteristics:**
  - Can contain duplicate values.
  - Can contain NULL values.
- **Example:** StudentID in the Enrollment table can be a foreign key referencing StudentID in the Student table.

6. Super Key:

- **Definition:** A set of one or more attributes that can uniquely identify a record in a table.
- **Characteristics:** Can contain additional attributes that are not necessary for unique identification.
- **Example:** StudentID alone is a super key, and StudentID along with Name is also a super key.

7. Unique Key:

- **Definition:** An attribute or a set of attributes that ensures all values in a column or a group of columns are unique across the database.
- **Characteristics:**
  - Similar to the primary key but can accept a single NULL value.
- **Example:** Email in the Student table can be a unique key if each student has a unique email address.

ER Diagram Representation:

- Primary keys are underlined in entity representations.
- Foreign keys are represented with a dashed line connecting the two related entities.

Example Scenarios

University Database:

- **Entity:** Student
  - **Attributes:** StudentID (Primary Key), Name, DateOfBirth, Address, Email (Unique Key), PhoneNumbers
- **Entity:** Course
  - **Attributes:** CourseID (Primary Key), CourseName, Credits
- **Entity:** Enrollment
  - **Attributes:** EnrollmentID (Primary Key), StudentID (Foreign Key), CourseID (Foreign Key), EnrollmentDate

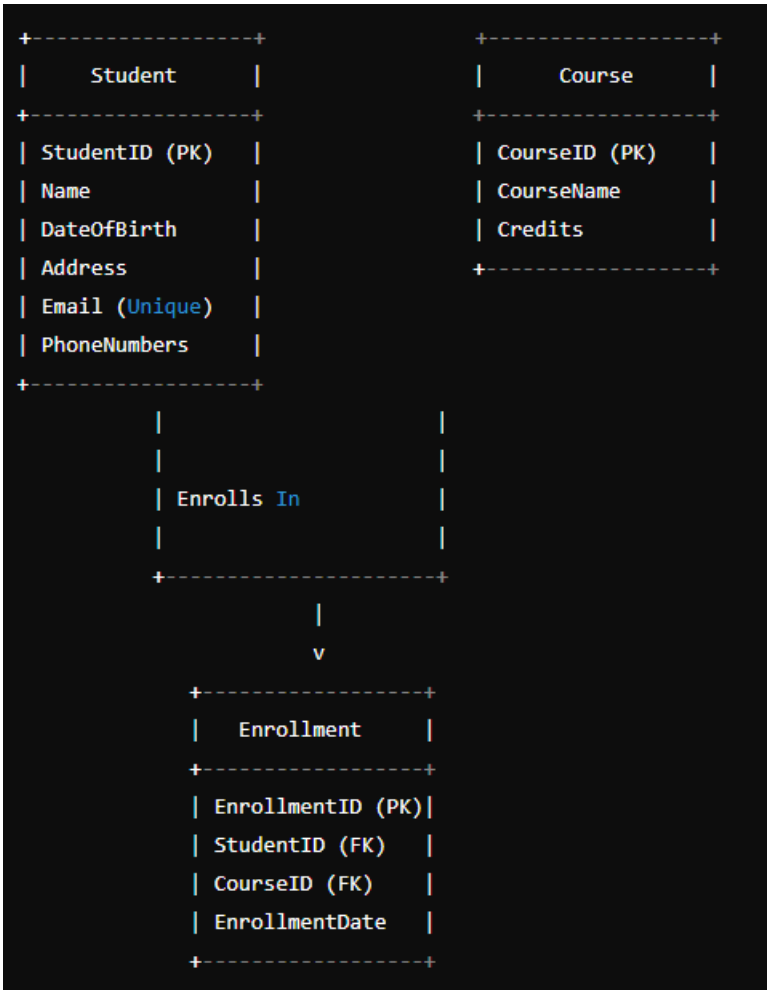


Fig 4.2 University Database Example

Attributes and keys are integral to defining the structure and relationships of data in a database. Attributes describe the properties of an entity, while keys ensure the uniqueness and integrity of data, facilitating efficient data retrieval and management. Understanding these concepts is crucial for designing robust and efficient databases that meet the requirements of complex data-driven applications.

4.4 OTHER TYPES OF INDEXES

In addition to the commonly used B-tree and hash indexes, modern database systems support several specialized indexing techniques tailored for specific data types and query patterns. These indexes improve performance for queries that involve low-cardinality attributes, text search, or spatial/geometric data. The following are three important types of specialized indexes.

4.4.1 BITMAP INDEXES

A **bitmap index** uses bit vectors (bitmaps) to represent the presence or absence of a value in each row.

It is especially efficient when:

- The attribute has **low cardinality** (few distinct values), such as gender, status, category, or department code.
- Queries involve **complex Boolean conditions** (AND, OR, NOT).

How it works

- For each distinct value of the attribute, a bitmap (array of bits) is created.
- Bit = 1 → row contains that value
- Bit = 0 → row does not contain that value
- Bitwise operations make query evaluation very fast.

Advantages

- Very compact storage for low-cardinality columns.
- Extremely fast for combining conditions using bitwise operations.
- Good for read-heavy, analytical workloads (e.g., data warehouses).

Limitations

- Not suitable for high-cardinality columns.
- Bitmap indexes increase overhead on updates (due to bitmaps needing modification).

4.4.2 FULL-TEXT INDEXES

A **full-text index** enables efficient searching of large text fields, documents, and unstructured text.

Traditional indexes are inefficient for matching keywords, phrases, or natural-language queries—full-text indexes solve this.

Features

- Break text into *tokens* (words, stems).
- Build an **inverted index** mapping terms → list of documents/rows containing them.
- Support advanced text-search operations like:
  - Keyword search
  - Phrase search
  - Boolean text search
  - Relevance ranking
  - Stemming and stop-word filtering

Applications

- Searching in fields such as product descriptions, articles, emails, comments, etc.
- Required in information retrieval systems, search engines, and content-heavy databases.

Advantages

- Very fast text-search queries.
- Supports ranking and relevance-based search.

Limitations

- Index creation and maintenance can be expensive.
- Performance depends on tokenizer and language-specific processing.

4.4.3 SPATIAL INDEXES

A **spatial index** is designed for storing and querying geometric and geographic data—such as points, lines, polygons, and regions.

Examples of spatial data

- GIS coordinates (latitude/longitude)
- Maps, boundaries, routes
- Locations of services (ATMs, hospitals)
- Shapes and geometric figures

Common spatial index structures

- **R-tree (most widely used)**
- R*-tree, Quad-trees, KD-trees (depending on the DBMS)

Why spatial indexes are needed
Spatial queries often involve operations such as:

- "Find all points within a radius"
- "Find regions that overlap this polygon"
- "Locate the nearest neighbor"

B-tree indexes cannot handle multi-dimensional data efficiently, whereas spatial indexes group nearby objects and improve search performance enormously.

Advantages

- Efficient multi-dimensional search (2D/3D).
- Fast spatial operations (range queries, intersection, containment).

Limitations

- More complex structure than B-trees.
- Maintenance overhead for frequently updated spatial data.

4.5. RELATIONSHIP TYPES, RELATIONSHIP SETS, AND ROLES

In Database Management Systems (DBMS), relationships between entities are crucial for representing how data interacts and is associated within the database. Understanding relationship types, relationship sets, and roles helps in accurately modeling these interactions within an Entity-Relationship (ER) model.

4.5.1 Relationship Types

- A relationship type defines the association between two or more entity types. It describes how entities of different types are related to each other.
- **Characteristics:**
- **Degree of Relationship:** Indicates the number of entity types involved in the relationship.
  - **Unary Relationship:** Involves one entity type (e.g., an employee supervises other employees).
  - **Binary Relationship:** Involves two entity types (e.g., students enroll in courses).
  - **Ternary Relationship:** Involves three entity types (e.g., a supplier supplies products to a warehouse).

- **Cardinality Constraints:** Specifies the number of instances of one entity type that can be associated with an instance of another entity type.

  - **One-to-One (1:1):** One instance of an entity is associated with one instance of another entity (e.g., each person has one passport).
  - **One-to-Many (1):** One instance of an entity is associated with multiple instances of another entity (e.g., a teacher teaches many students).
  - **Many-to-Many (M):** Multiple instances of an entity are associated with multiple instances of another entity (e.g., students enroll in multiple courses, and each course has multiple students).

Example:

- A Student entity type and a Course entity type can have an Enrolls relationship type indicating that students enroll in courses.

ER Diagram Representation:

- Relationships are represented by diamonds connecting the involved entities.

4.5.2 Relationship Sets

A relationship set is a collection of relationships of the same type. It represents the set of associations between instances of one entity set and instances of another (or the same) entity set.

Characteristics:

- **Instance Collection:** Contains all instances of a particular relationship type at any given time.
- **Dynamic:** The number of relationships in the set can change over time as entities are added, modified, or removed.
- **Example:** The Enrolls relationship set would include all instances where students have enrolled in courses.
- **ER Diagram Representation:** Represented by the same diamond as the relationship type, with lines connecting to the involved entities.

4.5.3 Roles

- Roles specify the function that an entity plays in a relationship. Roles are especially important in relationships involving the same entity type more than once (recursive relationships).

Characteristics:

- **Role Names:** Identify the purpose of an entity within the relationship. Role names help clarify the participation of an entity in the relationship.
- **Recursive Relationships:** Used to define roles in relationships where the same entity type participates more than once.

Example:

- In a Supervises relationship between the Employee entity type, roles can be Supervisor and Subordinate.
- **ER Diagram Representation:**
- Roles are often labeled on the connecting lines in the ER diagram to specify the function of each entity in the relationship.

Employee Database:

Entities:

- Employee (EmployeeID, Name, Position)
- **Relationships:**
- Supervises (between Employee and Employee)
  - **Roles:** Supervisor, Subordinate
  - **Cardinality:** One-to-Many (1)

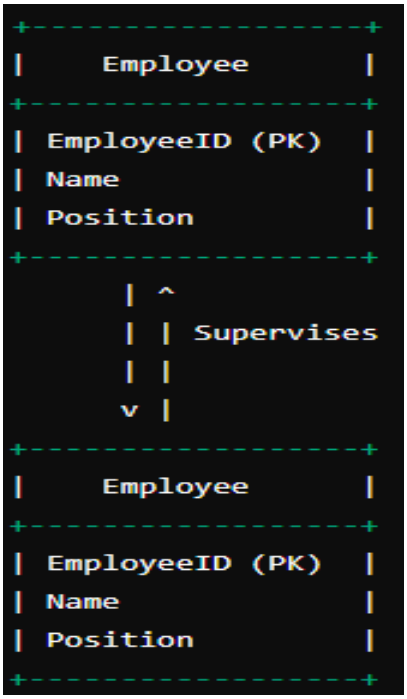


Fig 4.3 ER Diagram of Employee representation of Relationship

Understanding relationship types, relationship sets, and roles is essential for accurately modeling the interactions between data entities in a database. These concepts ensure that the relationships among entities are correctly represented, facilitating efficient data management and retrieval. Proper use of these elements in ER modeling leads to well-structured databases that accurately reflect real-world scenarios and support the required data operations.

4.6. STRUCTURAL CONSTRAINTS

Structural constraints in a Database Management System (DBMS) are rules that enforce restrictions on the relationships between entities to ensure the integrity and consistency of the data. These constraints play a crucial role in defining how entities interact with each other and what kind of relationships are permissible.

4.6.1 Domain Constraints

Definition: Restrictions on the permissible values for a given attribute.

Example: An attribute age should only accept integer values between 0 and 120.

Implementation: Data types and value ranges

```
CREATE TABLE      Person (
   age INT CHECK (age >= 0 AND age <= 120)
);
```

4.6.2 Entity Integrity Constraints

Definition: Ensure each entity (row) is uniquely identifiable.

Example: Every table should have a primary key, and no primary key value can be null.

Implementation: Primary key constraints

```
CREATE TABLE Employee (
   employee_id INT PRIMARY KEY,
   name VARCHAR(50)
);
```

4.6.3 Referential Integrity Constraints

Definition: Ensure that a foreign key value always points to an existing, valid record in another table.

Example: An order table's customer_id must match a valid customer_id in the customers table.

Implementation: Foreign key constraints

```
CREATE TABLE Orders (
   order_id INT PRIMARY KEY,
   customer_id INT,
   FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

4.6.4 Unique Constraints

Definition: Ensure that all values in a column or a set of columns are unique.

Example: Email addresses in a users table must be unique.
Implementation: Unique constraints.

```
CREATE TABLE Users (
   user_id INT PRIMARY KEY,
   email VARCHAR(100) UNIQUE
);
```

4.6.5 Check Constraints

Definition: Specify a condition that each row must satisfy.

Example: An employee's salary must be greater than the minimum wage.

Implementation: Check constraints

```
CREATE TABLE Employees (
   employee_id INT PRIMARY KEY,
   salary DECIMAL(10, 2),
   CHECK (salary >= 1500)
);
```

4.6.6 Not Null Constraints

Definition: Ensure that a column cannot have null values.
Example: An employee's name cannot be null.
Implementation: Not null constraints

```
CREATE TABLE Employees (
   employee_id INT PRIMARY KEY,
   name VARCHAR(50) NOT NULL
);
```

4.6.7 Default Constraints

Definition: Provide a default value for a column when no value is specified.

Example: The status of an order should default to 'pending' if not specified.

Implementation: Default constraints.

```
CREATE TABLE Orders (
   order_id INT PRIMARY KEY,
   status VARCHAR(20) DEFAULT 'pending'
);
```

4.7 WEAK ENTITY TYPES

In a Database Management System (DBMS), a weak entity type is an entity type that cannot be uniquely identified by its own attributes alone. Instead, it relies on a "strong" or "owner" entity to ensure its unique identification. Weak entities typically have a partial key and are associated with a strong entity through a relationship.

4.7.1 Key Characteristics of Weak Entity Types

- **Dependency on Strong Entity**:
  - Weak entities do not have a primary key that can uniquely identify their instances independently.
  - They depend on the primary key of a strong entity for unique identification.
- **Partial Key (Discriminator)**:
  - A weak entity has a partial key, also known as a discriminator, which, when combined with the primary key of the strong entity, uniquely identifies each instance of the weak entity.
- **Existence Dependency**:
  - Weak entities are existence-dependent on the strong entity. They cannot exist without being associated with an instance of the strong entity.
- **Identifying Relationship**:
  - The relationship between a weak entity and its strong entity is called an identifying relationship. This relationship helps in linking the weak entity to the strong entity and ensures that the weak entity can be uniquely identified.

4.7.2 Example of Weak Entity Type

Consider a database for a university where each student (strong entity) can have multiple dependents (weak entity). The dependents cannot be uniquely identified without referencing the student.

Strong Entity: Student

- Attributes: StudentID (Primary Key), Name, DateOfBirth

Weak Entity: Dependent

- Attributes: DependentName, Age, Relationship
- Partial Key: DependentName
- Identifying Relationship: Each dependent is associated with a specific student.

Weak Entity: Dependent representation in SQL:

```
CREATE TABLE Dependents (
   DependentName VARCHAR(100),
   Age INT,
   Relationship VARCHAR(50),
   StudentID INT,
   PRIMARY KEY (DependentName, StudentID),
   FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

By understanding and correctly implementing weak entities, you can ensure that your database accurately models real-world relationships and maintains data integrity.

4.8 RELATIONSHIP TYPES OF DEGREE HIGHER THAN TWO IN DBMS

In a Database Management System (DBMS), relationships are used to establish associations between different entity types. Most relationships are binary, involving two entities, but there are cases where relationships involve three or more entities. These are called n-ary relationships, where "n" represents the degree of the relationship. Here are the key aspects and examples of relationship types of degree higher than two:

Ternary Relationships (Degree 3)

A ternary relationship involves three different entity types. It is used when a relationship cannot be decomposed into binary relationships without losing some essential semantics.

Example: Supplier-Product-Project

Consider a scenario where a company manages suppliers, products, and projects. A ternary relationship might be used to represent which supplier supplies which product to which project.

Entities:

- Supplier (SupplierID, SupplierName)
- Product (ProductID, ProductName)
- Project (ProjectID, ProjectName)

Ternary Relationship: Supplies

- Attributes: Quantity, Date

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

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

CREATE TABLE Projects (
   ProjectID INT PRIMARY KEY,
   ProjectName VARCHAR(100)
);

CREATE TABLE Supplies (
   SupplierID INT,
   ProductID INT,
   ProjectID INT,
```

```
    Quantity INT,
    Date DATE,
    PRIMARY KEY (SupplierID, ProductID, ProjectID),
    FOREIGN KEY (SupplierID) REFERENCES Suppliers(SupplierID),
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID),
    FOREIGN KEY (ProjectID) REFERENCES Projects(ProjectID)
);
```

4.9 SUMMARY

Data modelling using the Entity-Relationship (ER) model involves creating a conceptual representation of a database's structure, focusing on entities (real-world objects or concepts), their attributes (properties), and the relationships between them. The ER model uses diagrams with rectangles for entities, ovals for attributes, and diamonds for relationships, connected by lines to illustrate the data structure clearly. This method helps in visualizing, communicating, and documenting the database design, ensuring that it accurately represents real-world scenarios and serves as a blueprint for creating an efficient and scalable database.

4.10 TECHNICAL TERMS

Entity, Attributes, Entity Type, Weak Entity, Relationship, Relationship Type, Constraint, ER model

4.11 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Illustrate about Entity and Entity Types
2. Describe about Relationship Types
3. Explain about Company Database ER Model

Short questions:

1. Write about Weak Entity
2. Define Key Constraints

4.12 SUGGESTED READINGS

1. "Database System Concepts" by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
2. "Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe
3. "Database Management Systems" by Raghu Ramakrishnan and Johannes Gehrke
4. "An Introduction to Database Systems" by C.J. Date
5. "SQL and Relational Theory: How to Write Accurate SQL Code" by C.J. Date

Dr. Neelima Guntupalli

LESSON- 05

THE ENHANCED ENTITY-RELATIONSHIP MODEL

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of The Enhanced Entity-Relationship Model. The chapter began with Sub classes, Super classes and Inheritance, Specialization and Generalization, Constraints and Characteristics of Specialization and Generalization Hierarchies, Modeling of Union Types using Categories, An Example University ERR Schema, Design Choices and Formal Definitions. After completing this chapter, the student will understand The Enhanced Entity-Relationship Model.

5.1 INTRODUCTION

The Enhanced Entity-Relationship (EER) model extends the original Entity-Relationship (ER) model to support more complex data representations and constraints. It introduces additional concepts like subclasses, superclasses, inheritance, specialization, generalization, and union types, making it a powerful tool for advanced database design.

The chapter first covered began with understanding Sub classes, Super classes and Inheritance, Specialization and Generalization, Constraints and Characteristics of Specialization and Generalization Hierarchies, Modeling of Union Types using Categories, An Example University ERR Schema, Design Choices and Formal Definitions.

5.2 SUBCLASSES, SUPERCLASSES, AND INHERITANCE

5.2.1 Subclasses

In the context of the Enhanced Entity-Relationship (EER) model, a subclass is a specialized form of an entity that inherits attributes and relationships from a parent entity, known as the superclass. The subclass can also have its own unique attributes and relationships that differentiate it from other subclasses and the superclass. This concept is essential for modeling complex data structures in database management systems (DBMS).

Example: A Person entity can be specialized into Student and Teacher subclasses. While both Student and Teacher inherit attributes like Name and Date of Birth from Person, Student might have additional attributes such as StudentID and Major, and Teacher might have EmployeeID and Department.

When creating subclasses in an EER diagram, it is essential to clearly define the superclass and identify the distinguishing characteristics that justify the creation of subclasses.

Steps:

1. **Identify the Superclass:** Determine the general entity that will serve as the superclass.
2. **Define Attributes and Relationships:** List the attributes and relationships common to all subclasses.
3. **Identify Specializations:** Determine the specific entities (subclasses) that will be derived from the superclass based on unique attributes or relationships.
4. **Draw Inheritance Arcs:** Use arcs or lines to connect subclasses to the superclass, indicating inheritance.

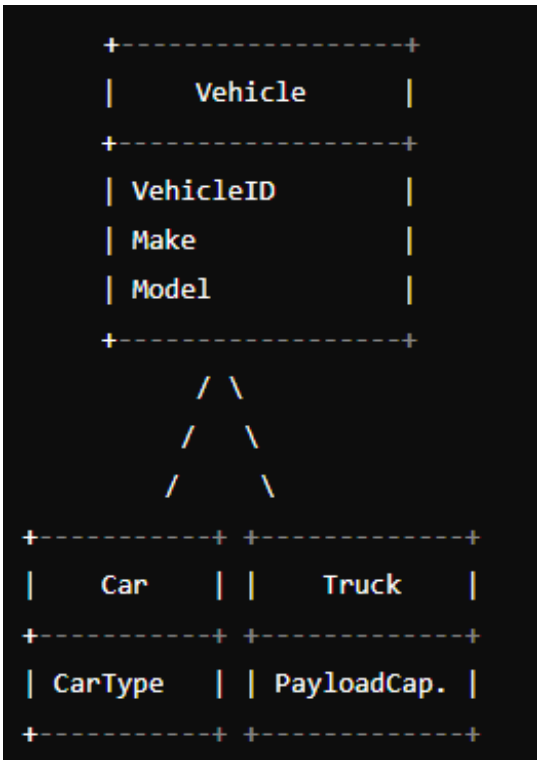


Fig 5.1 Subclass in EER Diagram

Advantages of Using Subclasses

1. **Reusability:** Common attributes and relationships are defined once in the superclass and inherited by subclasses.
2. **Reduced Redundancy:** Inheritance reduces the need to duplicate attributes across multiple entities.
3. **Clearer Modeling:** Subclasses allow for more precise modeling of real-world entities and their unique characteristics.

5.2.2 Superclasses

A superclass is a higher-level entity that contains common attributes and relationships shared by one or more subclasses A superclass is a generalized entity from which subclasses are derived.

Example:

- Person is the superclass for Student and Teacher.
- Vehicle as a superclass. It includes attributes common to all types of vehicles, such as license_plate_number, manufacturer, and model.

5.2.3 Inheritance

Inheritance allows a subclass to inherit attributes and relationships from its superclass. This promotes reusability and consistency within the data model.

Example:

- Attributes Name and DateOfBirth in Person are inherited by both Student and Teacher.

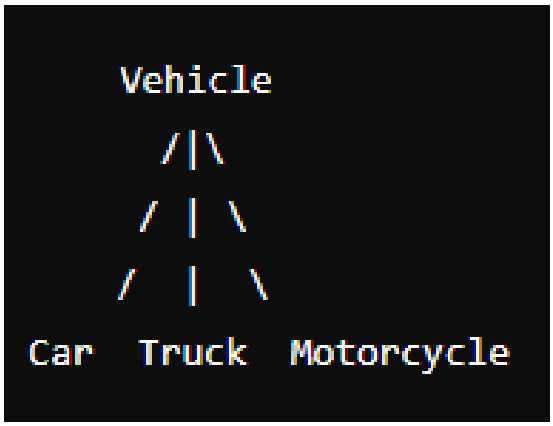


Fig 5.2 Inheritance in EER Diagram

Step-by-Step Process

1. **Identify Common Entities and Attributes**:
    - Common entity: Vehicle
    - Common attributes: vehicle_id, license_plate_number, manufacturer, model
2. **Define the Superclass**:
    - Superclass: Vehicle
    - Attributes: vehicle_id, license_plate_number, manufacturer, model
3. **Define the Subclasses**:
    - Car: Attributes: number_of_doors, trunk_capacity
    - Truck: Attributes: payload_capacity, number_of_axles
    - Motorcycle: Attributes: type_of_handlebars, engine_capacity
4. **Establish Inheritance Relationships**:
    - Draw Vehicle as the superclass.
    - Connect Vehicle to Car, Truck, and Motorcycle using lines.
    - Use a triangle symbol to represent the inheritance relationship, with the apex pointing towards Vehicle.
5. **Specify Constraints**:
    - **Disjoint Constraint**: A vehicle can be either a Car, Truck, or Motorcycle, but not more than one (disjoint).
    - **Total Participation**: Every vehicle must be classified as one of the subclasses (total participation).

By following these steps, you can effectively create an EER diagram that represents the inheritance of vehicles, ensuring a clear and structured data model.

5.3 SPECIALIZATION AND GENERALIZATION

5.3.1 Specialization

Specialization in EER (Extended Entity-Relationship) modeling involves creating lower-level entity types (subclasses) from a higher-level entity type (superclass) based on some distinguishing characteristics. This process is a top-down approach where a general entity is divided into more specific entities.

Example: Person can be specialized into Student and Teacher based on roles within an educational institution.

Specialization Constraints

- **Disjoint Constraint**: Ensures that a Vehicle can be only one of the subclasses (Car, Truck, or Motorcycle).
- **Total Participation**: Ensures that every Vehicle instance must be a member of one of the subclasses.

Specialization in DBMS allows for the creation of a more structured and organized database by breaking down general entities into more specific entities based on distinguishing characteristics. This approach helps in managing and querying data more efficiently while maintaining data integrity and avoiding redundancy.

5.3.2 GENERALIZATION

Generalization in Database Management Systems (DBMS) is the process of combining multiple lower-level entity types into a higher-level entity type based on common attributes or relationships. It is the opposite of specialization and follows a bottom-up approach. This method is useful for simplifying and abstracting complex databases by identifying commonalities among various entities and representing them in a generalized manner.

Key Concepts of Generalization

- **Higher-Level Entity (Superclass)**: The generalized entity that encompasses the shared attributes and relationships of the lower-level entities.
- **Lower-Level Entities (Subclasses)**: The specific entities that are combined to form the higher-level entity. Each subclass may have its own unique attributes and relationships.

Example: Consider a scenario where you have different types of vehicles: cars, trucks, and motorcycles. Each type of vehicle has its own specific attributes, but they also share some common attributes.

Step-by-Step Process

1. **Identify Common Attributes**:
   - Common attributes: vehicle_id, license_plate_number, manufacturer, model
2. **Define the Higher-Level Entity (Superclass)**:
   - Superclass: Vehicle
   - Attributes: vehicle_id, license_plate_number, manufacturer, model

3. **Define Lower-Level Entities (Subclasses)**:
   - Car: Attributes: number_of_doors, trunk_capacity
   - Truck: Attributes: payload_capacity, number_of_axles
   - Motorcycle: Attributes: type_of_handlebars, engine_capacity
4. **Establish Generalization Relationships**:
   - Connect Car, Truck, and Motorcycle to Vehicle using lines.
   - Use a triangle symbol to represent the generalization relationship, with the base pointing towards Car, Truck, and Motorcycle, and the apex pointing towards Vehicle.
5. **Specify Constraints**:
   - **Disjoint Constraint**: Indicate that a Vehicle can be either a Car, Truck, or Motorcycle, but not more than one (disjoint).
   - **Total Participation**: Indicate that every Vehicle must be one of the specialized types (total participation).

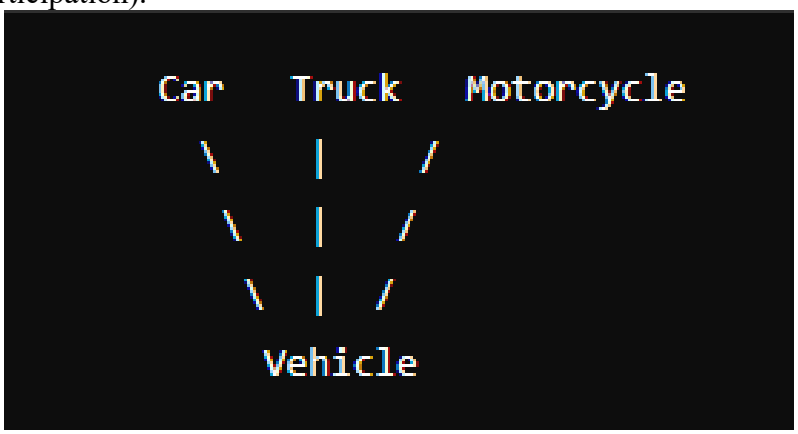


Fig 5.3Generalization in EER Diagram

Generalization in DBMS is a powerful modeling technique that helps in abstracting and simplifying complex data structures by identifying commonalities among different entities and representing them as a generalized entity. This approach enhances data organization, reduces redundancy, and improves database maintainability and query efficiency.

5.4 CONSTRAINTS AND CHARACTERISTICS OF SPECIALIZATION AND GENERALIZATION HIERARCHIES

5.4.1 Completeness Constraint

The Completeness Constraint in the Extended Entity-Relationship (EER) model specifies whether every instance of a higher-level entity (superclass) must belong to at least one lower-level entity (subclass). It ensures that all possible entity instances are accounted for in the specialization/generalization hierarchy.

There are two types of completeness constraints:

1. **Total Completeness (Total Participation)**:
   - Every instance of the superclass must be a member of at least one subclass.
   - This is represented by a double line connecting the superclass to the subclasses in the EER diagram.
   - Example: If every Vehicle must be either a Car, Truck, or Motorcycle, then the specialization is totally complete.

2. **Partial Completeness (Partial Participation)**:
   - Some instances of the superclass may not belong to any of the subclasses.
   - This is represented by a single line connecting the superclass to the subclasses in the EER diagram.
   - Example: If some Vehicles might not be classified as Car, Truck, or Motorcycle, then the specialization is partially complete.

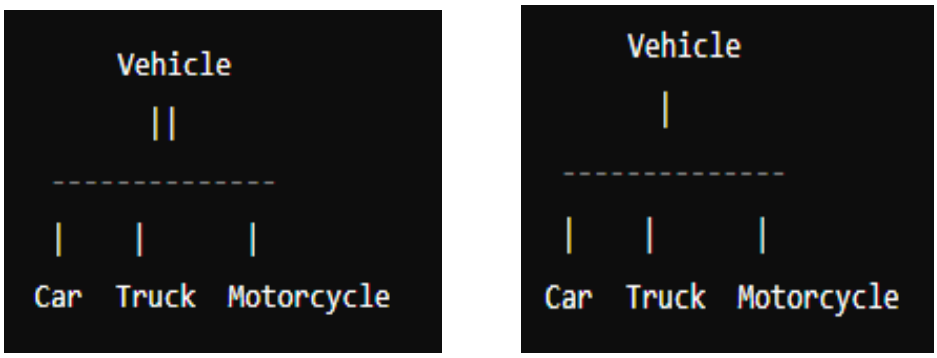


Fig 5.4 (a) Total Specialization (b) Partial Specialization

The choice between total and partial specialization depends on the specific requirements of the domain being modeled.

5.4.2 Disjointness Constraint

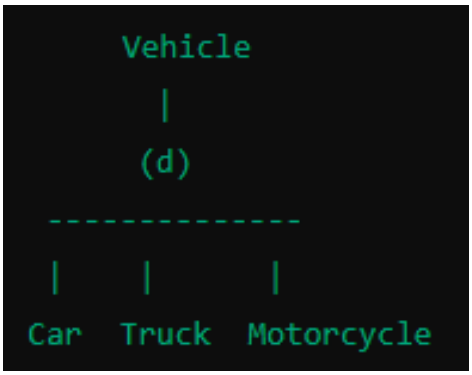
This specifies whether an instance of a superclass can be a member of more than one subclass.

- **Disjoint Specialization:** An instance of the superclass can belong to only one subclass.
- **Overlap Specialization:** An instance of the superclass can belong to multiple subclasses.

Example
Consider an EER diagram for vehicles:
- **Superclass:** Vehicle
- **Subclasses:** Car, Truck, Motorcycle

Disjoint (Exclusive) Constraint
If a vehicle can be either a Car, Truck, or Motorcycle, but not more than one of these at the same time, the disjointness constraint is disjoint. This is shown by a "d" in the diagram.

Overlapping Constraint
If a vehicle can be classified as more than one subclass (e.g., a vehicle that is both a Car and a Truck), the disjointness constraint is overlapping. This is shown by an "o" in the diagram.



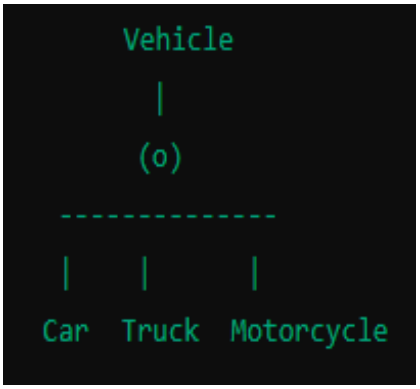


Fig 5.5 (a) Disjoint (Exclusive) Constraint (b) Overlapping Constraint

5.4.3 Combining Completeness and Disjointness Constraints

In practice, both completeness and disjointness constraints can be combined to provide a more precise definition of how instances of a superclass relate to its subclasses.

Example with Total Completeness and Disjoint Constraint:

If every vehicle must be either a Car, Truck, or Motorcycle, and cannot be more than one at the same time, it would be represented as:

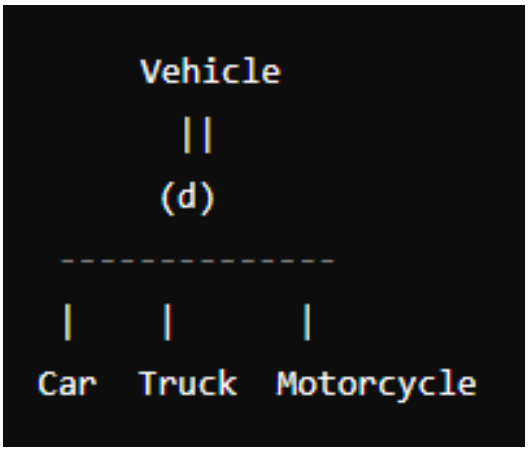


Fig 5.6 Total Completeness and Disjoint Constraint

Example with Partial Completeness and Overlapping Constraint:

If some vehicles may not be categorized as Car, Truck, or Motorcycle, but those that are can belong to more than one subclass, it would be represented as:

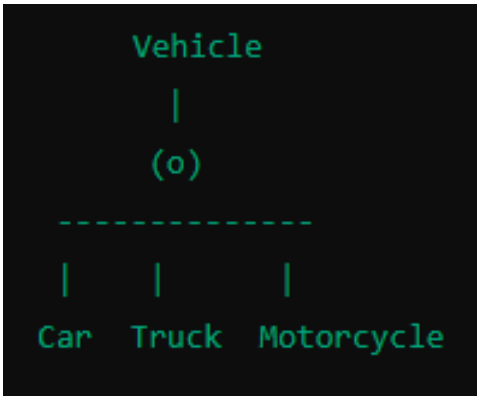


Fig 5.7 Partial Completeness and Overlapping Constraint

These constraints help in accurately modeling real-world scenarios and ensuring data integrity in the database design.

5.5 MODELING OF UNION TYPES USING CATEGORIES

Union types (or categories) are used to represent a single superclass that is derived from multiple distinct superclasses.

Example: A Member entity could be a union of Student, Teacher, and Alumni, allowing the Member to inherit attributes from any of these entities.
To model union types using categories for the Vehicle entity in an Enhanced Entity-Relationship (EER) diagram, we will follow a structured approach. We'll define the superclass Vehicle and its potential categories (subclasses) and establish a union entity to represent the different types of vehicles.

Steps to Model Union Types Using Categories in EER:

1. **Identify the Superclass:** Vehicle
2. **Identify the Subclasses:** Car, Truck, Motorcycle
3. **Define the Union Entity:** Create a union entity to represent the categories.
4. **Set Constraints:** Specify completeness and disjointness constraints.

Union Entity:

Create a Vehicle Type entity that represents the union of Car, Truck, and Motorcycle.

Modeling union types using categories for the Vehicle entity in an EER diagram involves defining the superclass Vehicle, its subclasses Car, Truck, and Motorcycle, and specifying the union entity with appropriate constraints. This method ensures a robust and flexible database design that can handle complex categorization requirements for vehicles.

5.6 EXAMPLE UNIVERSITY EER SCHEMA

Consider an EER schema for a university database, including entities such as Person, Student, Teacher, Course, and relationships like Enrolls, Teaches.

5.5.1 Entities and Relationships

Entities:

- Student (StudentID, Name, DateOfBirth, Major, Year, GPA)
- Professor (ProfessorID, Name, Department, Title, OfficeNumber)
- Course (CourseID, CourseName, Credits, Department)
- Department (DepartmentID, DepartmentName, Building)
- Classroom (RoomNumber, Building, Capacity)
- Enrollment (EnrollmentID, Grade)
- Teaches (Semester, Year)

Relationships:

- Student 1:M Enrollment (Enrolls)
- Course 1:M Enrollment (Contains)
- Professor 1:M Teaches (Teaches)
- Course 1:M Teaches (IsTaughtIn)
- Department 1:M Course (Offers)
- Department 1:M Professor (Has)
- Classroom 1:M Course (Hosts)

5.5.2 Schema Diagram

A diagram representing the entities, their attributes, and relationships can be used to visualize the EER model.

A **schema diagram** is a visual representation of the entities, their attributes, relationships, and the constraints that define the structure of the Enhanced Entity–Relationship (EER) model.

It provides a high-level overview of the database design, showing how different entities are related and how specialization, generalization, and inheritance are represented.

The schema diagram helps database designers and users to:

- Understand **the organization of data** in the system.
- Identify **entity types, attributes, and relationships**.
- Visualize **hierarchical structures** formed through subclasses and superclasses.
- Observe **constraints** such as disjointness, completeness, and participation.

Components of an EER Schema Diagram

1. **Entities:**
    - Represented by rectangles.
    - Each entity includes its attributes, and the **primary key** is underlined.

Example:
Student(StudentID, Name, Age, Major)

2. **Attributes:**
   - Shown as ovals connected to their entity.
   - Multivalued attributes are represented by double ovals, and derived attributes by dashed ovals.
3. **Relationships:**
   - Depicted by diamonds connecting participating entities.
   - Cardinality ratios (1:1, 1:N, M:N) and participation constraints are shown on the connecting lines.
4. **Specialization and Generalization:**
   - Represented using a **triangle** symbol.
   - A downward triangle indicates specialization (from superclass to subclasses).
   - An upward triangle indicates generalization (from subclasses to a superclass).
   - Disjointness and completeness constraints are labeled near the triangle:
     - d for disjoint
     - o for overlapping
     - T for total
     - P for partial
5. **Subclasses and Superclasses:**
   - **Superclasses** appear as general entities from which common attributes are inherited.
   - **Subclasses** branch out and may have additional attributes or relationships.
6. **Categories (Union Types):**
   - Represented by a **circle** connected to multiple superclasses.
   - Indicates that the subclass is formed as a union of different entity sets.

Example: University EER Schema Diagram

A **University EER schema** may include the following entities and relationships:

- **Entities:**
  - Person(PersonID, Name, Address)
  - Student(StudentID, Program)
  - Faculty(FacultyID, Department)
  - Course(CourseID, Title, Credits)
  - Department(DeptID, DeptName)
- **Relationships:**
  - EnrolledIn(Student, Course)
  - Teaches(Faculty, Course)
  - Advises(Faculty, Student)
- **Specialization:**
  - Person is a superclass.
  - Student and Faculty are subclasses derived from Person.
  - Further specialization of Student into UndergraduateStudent and GraduateStudent.
- **Constraints:**
  - The specialization of Person into Student and Faculty is **total and disjoint** — every person must be either a student or faculty, but not both.

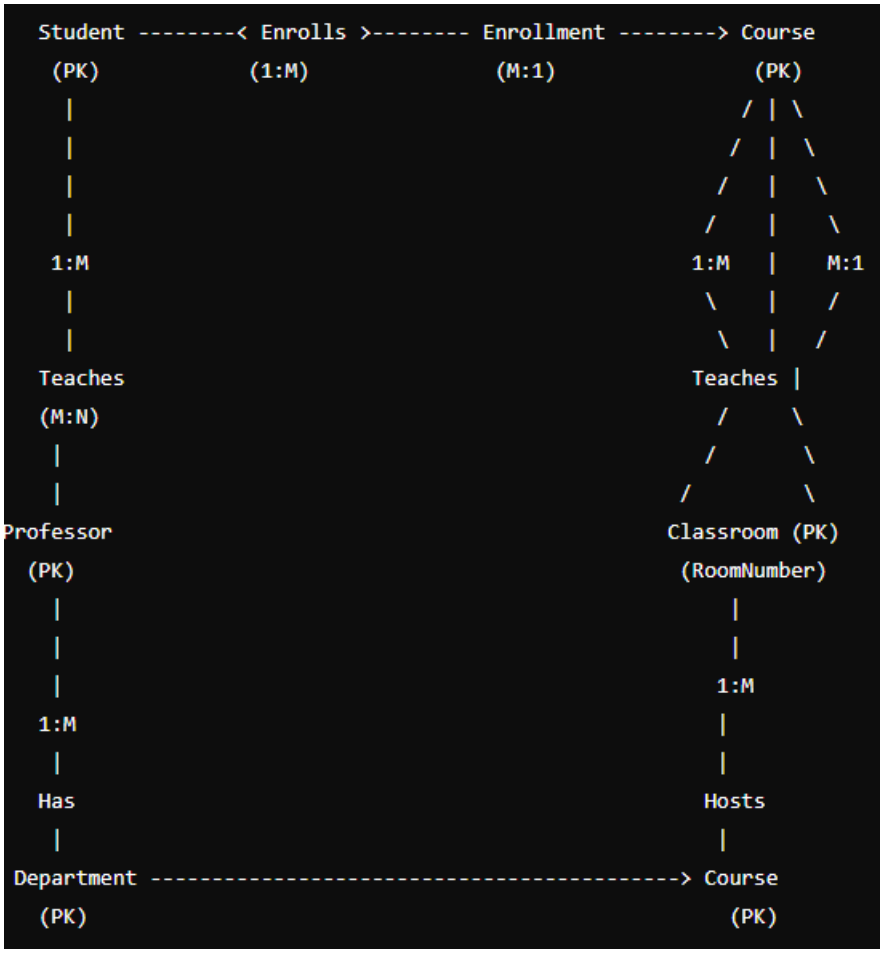


Fig 5.8 EER Diagram University Database

In the diagram:

- Rectangles represent entities such as *Person*, *Student*, and *Course*.
- Diamonds represent relationships like *Teaches* and *EnrolledIn*.
- Triangles represent specialization/generalization hierarchies.
- Lines connecting entities show participation and cardinality.
- Such a diagram allows database designers to visually validate:
- Attribute inheritance from superclasses to subclasses.
- Relationship participation across hierarchical levels.
- Constraint enforcement between entities.

Advantages of Schema Diagrams

- Provides a clear and intuitive visual representation of complex EER designs.
- Facilitates communication between designers, developers, and stakeholders.
- Helps identify redundancies and inconsistencies early in the design process.
- Serves as a blueprint for converting the conceptual model into a relational schema.

5.7 DESIGN CHOICES AND FORMAL DEFINITIONS

5.7.1 Design Choices

Design choices involve decisions about which entities to include, how to structure them, and how to implement constraints and relationships.

Example: Deciding whether to use a total or partial specialization for Person can impact the flexibility and complexity of the database schema.

5.7.2 Formal Definitions

Formal definitions provide precise specifications for entities, attributes, relationships, and constraints within the EER model.

Example: The definition of the Person entity might include formal specifications for attributes like PersonID, Name, and DateOfBirth.

5.8 SUMMARY

The Enhanced Entity-Relationship (EER) model extends the traditional ER model by incorporating more advanced features like subclasses, superclasses, inheritance, specialization, generalization, and union types. These enhancements enable more precise and flexible data modeling, making it suitable for complex database designs. By understanding and applying these concepts, database designers can create robust and efficient database schemas that accurately reflect real-world scenarios.

This chapter provides a comprehensive overview of the Enhanced Entity-Relationship model, including its components, constraints, and applications in database design. The example university EER schema illustrates how these concepts can be practically applied to create a detailed and functional database schema.

5.9 TECHNICAL TERMS

Enhanced Entity, Relationship, Complete Constraint, Disjoint Constraint, Subclass, Super class , Inheritance, Specialization, Generalization.

5.10 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Illustrate about Specialization and Generalization
2. Describe about University EER Schema
3. Explain about Completeness Constraint

Short questions:

1. Write about Inheritance
2. Define Disjointness Constraint
3. Explain about Entities and Relationships

5.11 SUGGESTED READINGS

1. "Database System Concepts" by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
2. "Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe
3. "Database Management Systems" by Raghu Ramakrishnan and Johannes Gehrke
4. "An Introduction to Database Systems" by C.J. Date
5. "SQL and Relational Theory: How to Write Accurate SQL Code" by C.J. Date
6. Elmasri, R., & Navathe, S. B. (2010). Fundamentals of Database Systems (6th ed.). Addison-Wesley.
7. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). Database System Concepts (6th ed.). McGraw-Hill.

Dr. Kampa Lavanya

LESSON- 06

THE RELATIONAL MODEL CONCEPTS

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of The Enhanced Entity-Relationship Model. The chapter began with Sub classes, Super classes and Inheritance, Specialization and Generalization, Constraints and Characteristics of Specialization and Generalization Hierarchies, Modeling of Union Types using Categories, An Example University ERR Schema, Design Choices and Formal Definitions. After completing this chapter, the student will understand The Enhanced Entity-Relationship Model.

6.1. INTRODUCTION

In any relational database, maintaining data accuracy, consistency, and inter-table relationships is essential. The relational data model achieves this through a set of well-defined rules and operations that govern how data is stored, manipulated, and related. Concepts such as integrity constraints ensure that only valid and consistent data is entered into the database, while relational operations provide powerful ways to query and transform data based on mathematical set theory and logic.

Furthermore, real-world databases often originate from conceptual designs built using Entity–Relationship (ER) models. To implement these models effectively, designers use ER-to-Relation mapping to convert entities, attributes, and relationships into relational tables. More advanced concepts like modelling union types using categories and depicting keys and foreign keys through diagrams help in visualizing data dependencies and enforcing referential integrity across relations. Together, these topics form the foundation for reliable, structured, and logically consistent database design.

In this chapter, we will explore the essential components of the relational data model — relations, attributes, tuples, schemas, and keys. We will also examine how integrity constraints are enforced and introduce relational algebra, the formal language for querying relational databases.

6.2. RELATIONAL DATA MODEL CONCEPTS

The relational data model, introduced by E. F. Codd in 1970, is the most widely used model for database design and implementation. It organizes data into relations, which are conceptualized as two-dimensional tables consisting of rows and columns. Each table represents a specific entity or relationship type, where rows (tuples) correspond to individual records and columns (attributes) represent properties of those records. This simple yet powerful structure makes the relational model easy to understand, flexible to use, and mathematically rigorous.

The relational model emphasizes data independence, data integrity, and ease of manipulation through well-defined operations based on relational algebra. Its foundation lies in set theory, ensuring that all data manipulations follow consistent and predictable rules. In this section, we will discuss the fundamental components of the relational data model — relations, attributes, tuples, domains, schemas, and keys — along with their properties and constraints. These concepts provide the logical basis for building, querying, and maintaining structured databases in real-world applications.

6.2.1 Relation: Structure and Properties

A relation is the fundamental building block of the relational data model. It represents a table of data consisting of rows and columns, where each row corresponds to a unique record and each column corresponds to an attribute of that record. In formal terms, a relation is a set of tuples (rows) sharing the same set of attributes (columns). The structure of a relation provides a systematic and uniform way to store and organize data.

Each relation in a database has a unique name and a fixed set of attributes, with each attribute having a distinct name and a defined domain of values. The order of attributes and tuples in a relation is not significant, since the relation is based on the mathematical concept of a set. However, the values contained in each tuple are atomic, meaning that every attribute holds a single, indivisible value—ensuring that no lists, arrays, or nested records are stored within a single field.

Example Relation 'Student'

StudentID	Name	Age	Major
1001	Alice	21	Computer Science
1002	Bob	22	Mathematics
1003	Carol	20	Physics

Characteristics of a Relation:

- Each attribute has a unique name within the relation.
- Each tuple represents a distinct record or data instance.
- Tuple order does not affect the meaning of the relation.
- No duplicate tuples are allowed in a valid relation.
- Attribute values are atomic, meaning they cannot be divided further.
- The relation name distinguishes one relation from another within the database.

Thus, a relation provides a structured and consistent method of representing data in tabular form. This tabular organization simplifies data retrieval and manipulation, enabling relational databases to maintain clarity, flexibility, and logical integrity.

6.2.2 Attributes and Domains

Each attribute in a relation describes one property or characteristic of the entity represented by the relation. Attributes are represented as columns in a table, and each attribute has an associated domain — the set of all possible or allowable values that the attribute can take. The domain defines the data type, format, and range of values, ensuring that all data stored in the attribute is valid and meaningful.

A domain acts as a constraint on attribute values, preventing inconsistent or incorrect data from being entered into the database. For example, an *Age* attribute might be restricted to integer values between 16 and 100, while a *Name* attribute may only accept alphabetic strings.

Example Attributes and Domains in the 'Student' Relation

Attribute	Domain	Example Values
StudentID	Integer	1001, 1002, 1003
Name	String	"Alice", "Bob", "Carol"
Age	Integer (16–100)	20, 21, 22
Major	Set of Strings	"Physics", "Mathematics", …

Key Points:

- Each attribute has a clearly defined domain that specifies acceptable values.
- Domains may restrict both the type (e.g., integer, string) and the range or format (e.g., valid age range or date format).
- Domains promote data integrity by ensuring uniformity and preventing invalid entries.
- When a value does not conform to its domain, the database system rejects it as an error.

Hence, domains play a crucial role in maintaining the accuracy, consistency, and reliability of data stored in a relational database.

6.2.3 Tuples

A tuple represents a single record or instance in a relation. It is an ordered set of attribute values, where each value corresponds to one attribute of the relation. In tabular form, a tuple appears as a row in the relation table. Each tuple provides a complete description of an entity by combining values from each attribute's domain.

In the relational model, every tuple must contain exactly one value for each attribute of the relation, and these values must be atomic, meaning they cannot be further divided into smaller components. This ensures that the data remains consistent and easy to process during query operations.

Example A Tuple from the 'Student' Relation

StudentID	Name	Age	Major
1001	Alice	21	Computer Science

In this example, the tuple *(1001, Alice, 21, Computer Science)* represents one student entity, where each value belongs to the domain of its corresponding attribute.

Characteristics of a Tuple:

- Contains one value for each attribute of the relation.
- All attribute values are atomic (indivisible).
- Each tuple is unique — no two tuples in the same relation are identical.
- Represents a single, complete record within the relation.

Thus, tuples serve as the fundamental data units in the relational model, capturing individual instances of entities and ensuring that all stored information is well-structured and logically consistent.

6.2.4 Schema and Instance

A relation schema defines the structure of a relation — it specifies the relation's name and the names and domains of all its attributes. The schema acts as a blueprint or template that describes how data is organized in the relation. It remains relatively stable over time and forms part of the database's overall design.

Example Relation Schema

Student(StudentID: Integer, Name: String, Age: Integer, Major: String)

In this schema,

- Student is the name of the relation.
- StudentID, Name, Age, and Major are attributes.
- Each attribute is associated with a domain (e.g., Integer, String).

The relation instance, on the other hand, refers to the actual data stored in the relation at a particular point in time. It represents a collection of tuples that conform to the structure defined by the relation schema. The instance of a relation is dynamic — it changes whenever tuples are inserted, deleted, or updated.

Example Relation Instance of 'Student'

StudentID	Name	Age	Major
1001	Alice	21	Computer Science
1002	Bob	22	Mathematics
1003	Carol	20	Physics

Key Points:

- The schema defines the structure and constraints of a relation.
- The instance represents the actual content or data of the relation at a given time.
- While the schema is fixed, the instance changes as the database is modified.

Hence, the relation schema provides a logical definition, and the relation instance provides a snapshot of data, together forming the foundation for representing information in a relational database.

6.2.5 Characteristics of Relations

A relation in the relational data model exhibits several important characteristics that define its structure and behavior. These characteristics ensure that data stored in relational databases remains well-organized, consistent, and compatible with the principles of set theory upon which the model is based.

Key Characteristics:

- **Attribute names are unique within a relation:**

Each attribute (column) in a relation has a distinct name to avoid ambiguity when referring to data items. For example, a relation cannot contain two attributes both named *ID*.

- **Tuples are unordered sets (not lists):**

The order of tuples (rows) in a relation has no significance. Whether a tuple appears first or last in the table does not affect the meaning of the data. Each tuple is identified by the values it contains, not by its position.

- **Attribute values are atomic:**

Every attribute in a relation must contain only a single, indivisible value from its domain. Complex or composite values such as lists or arrays are not permitted, ensuring simplicity and consistency in data representation.

- **No duplicate tuples are allowed:**

A valid relation cannot contain two identical tuples. This uniqueness property ensures that each record represents a distinct real-world entity or fact. Relations are sets; hence, operations on relations obey set theory principles:

Since a relation is defined as a set of tuples, all operations such as union, intersection, and difference follow the rules of set theory. This mathematical foundation enables relational algebra to provide powerful and logically sound data manipulation capabilities.

6.2.6 Keys

In a relational database, keys are essential for uniquely identifying tuples and establishing relationships between relations. A key is one or more attributes that together ensure that each tuple in a relation is unique. Keys not only prevent duplicate records but also play a vital role in enforcing data integrity and defining relationships among tables.

There are several types of keys used in the relational model, each serving a specific purpose.

Key Type	Description	Example
Super Key	Any set of one or more attributes that uniquely identifies a tuple in a relation. A relation may have multiple super keys.	{StudentID, Name}
Candidate Key	A minimal super key — that is, a super key from which no attribute can be removed without losing the property of uniqueness. Each relation can have one or more candidate keys.	{StudentID}
Primary Key	The chosen candidate key that uniquely identifies each tuple in a relation. The primary key must have unique values and cannot contain NULL entries.	StudentID
Foreign Key	An attribute (or set of attributes) in one relation that refers to the primary key of another relation, thereby establishing a link between the two relations.	Course.StudentID → Student.StudentID

Example Use of Keys
Consider two relations:

Student

StudentID	Name	Age	Major
1001	Alice	21	Computer Science
1002	Bob	22	Mathematics

Course

CourseID	Title	StudentID
C101	**DBMS**	**1001**
C102	**Data Science**	**1002**

Here:

- StudentID is the Primary Key in the *Student* relation.
- StudentID in the *Course* relation is a Foreign Key that refers to *Student.StudentID*, establishing a relationship between the two tables.

Key Points:

- Every relation must have a primary key to uniquely identify its tuples.
- A foreign key ensures referential integrity between related tables.
- The use of keys maintains uniqueness, consistency, and linkage across database relations.

Hence, keys form the foundation of relational database design by uniquely identifying data, preventing duplication, and connecting related entities through well-defined relationships.

6.3 INTEGRITY CONSTRAINTS

Integrity constraints are rules that ensure the accuracy, consistency, and validity of data stored in a relational database. They enforce logical conditions on the data to prevent invalid or inconsistent entries and maintain the overall correctness of the database state. In the relational model, integrity constraints are crucial for preserving data reliability across different relations.

The main types of integrity constraints are as follows:

- **Domain Constraints:**

These ensure that the value of each attribute lies within its defined domain. Each attribute in a relation is associated with a specific data type and permissible range or format. Any value that does not conform to its domain is considered invalid. *Example:* The attribute *Age* may have a domain of integer values between 16 and 100. Entering *Age = 'Twenty'* would violate the domain constraint.

- **Entity Integrity:**

This constraint ensures that every tuple in a relation can be uniquely identified. The primary key of a relation must contain unique and non-null values. *Example:* In the *Student* relation, the *StudentID* (primary key) must have a value for every student record; a NULL *StudentID* would violate entity integrity.

- **Referential Integrity:**

This constraint maintains consistency among related relations.
If a foreign key in one relation refers to a primary key in another, the foreign key value must either:

1. Match an existing primary key value in the referenced relation, or
2. Be NULL, if the relationship is optional.

Example: In the *Course* relation, *StudentID* (foreign key) must correspond to a valid *StudentID* in the *Student* relation.

6.4 RELATIONAL OPERATIONS

Relational operations are the fundamental manipulative tools of the relational data model. They are used to retrieve, combine, and transform data stored in one or more relations. These operations are derived from relational algebra, a formal mathematical system that defines a set of operations on relations (tables) to produce new relations as results.

Relational algebra operations enable users to query data precisely and efficiently without altering the underlying database structure. Each operation takes one or more relations as input and returns another relation as output, ensuring closure — one of the key properties of the relational model.

Basic Relational Algebra Operations

- **Selection (σ):**

Selects tuples (rows) from a relation that satisfy a specified condition.
Notation: σ_{condition}(Relation)
Example: σ_{Age > 20}(Student) → selects students older than 20.

- **Projection (π):**

Selects specific attributes (columns) from a relation, eliminating duplicates.
Notation: π_{attribute-list}(Relation)
Example: π_{Name, Major}(Student) → displays only student names and majors.

- **Join (⋈):**

Combines related tuples from two relations based on a common attribute.
Notation: Relation1 ⋈ Relation2
Example: Student ⋈ Enrollment → merges records of students and their course enrollments based on matching *StudentID* values.

- **Set Operations:**

Since relations are based on set theory, traditional set operations apply:

  - Union (∪): Combines tuples from two relations, removing duplicates.
  - Intersection (∩): Returns tuples common to both relations.
  - Difference (−): Returns tuples present in one relation but not in the other.
  - Cartesian Product (×): Produces all possible combinations of tuples from two relations.

Example: Relation 'Student'

StudentID	Name	Age	Major
1001	Alice	21	Computer Science
1002	Bob	22	Mathematics
1003	Carol	20	Physics

Illustrations:

- Selection Example: σ_{Major = 'Mathematics'}(Student) → retrieves only Bob's record.
- Projection Example: π_{Name, Age}(Student) → returns columns *Name* and *Age* for all students.
- Join Example: Student ⋈ Enrollment → merges student information with enrollment data using *StudentID*.

6.5 ER MODEL TO RELATION MAPPING

The Entity–Relationship (ER) model provides a high-level conceptual view of the data, while the relational model represents data in the form of tables (relations). The process of ER-to-Relation mapping (also called schema transformation) involves converting entities, attributes, and relationships from an ER diagram into corresponding relational schemas. This step ensures that the conceptual database design is accurately translated into an implementable relational structure.

In this mapping process:

- Each entity in the ER model becomes a relation (table).
- Each attribute of the entity becomes a column (field) in the relation.
- The primary key of the entity becomes the primary key of the relation.
- Relationships between entities are represented using foreign keys that reference primary keys in related tables.

Example: ER Diagram — Entities and Relationships

Consider an ER diagram with three entities: Student, Course, and Enrollment, where *Enrollment* represents the many-to-many relationship between *Student* and *Course*.
Entities:

- Student (*StudentID*, Name, Age, Major)
- Course (*CourseID*, Title, Credits)
- Enrollment (represents a relationship between Student and Course)

Mapping to Relations

1. Student Relation:

Each entity becomes a table with its attributes.

- Primary Key: StudentID

StudentID	Name	Age	Major
1001	Alice	21	Computer Science
1002	Bob	22	Mathematics

2. Relational Schema:

Student(StudentID, Name, Age, Major)

2. Course Relation:

Represents the *Course* entity.

- Primary Key: CourseID

CourseID	Title	Credits
C101	Database Systems	4
C102	Data Science	3

3. Relational Schema:

Course(CourseID, Title, Credits)

3. Enrollment Relation:

Represents the *Enrollment* relationship between *Student* and *Course*.

- Primary Key: Combination of StudentID and CourseID (composite key).
- Foreign Keys:
  - StudentID → references *Student.StudentID*
  - CourseID → references *Course.CourseID*

StudentID	CourseID	Grade
1001	C101	A
1002	C102	B

4. Relational Schema:

Enrollment(StudentID, CourseID, Grade)
Foreign Key Constraints:

- Enrollment.StudentID → Student.StudentID
- Enrollment.CourseID → Course.CourseID

The Student and Course entities map directly to relations, each with a primary key. The Enrollment relation serves as a bridge table that establishes connections between students and the courses they take using foreign keys. This mapping maintains referential integrity and accurately represents real-world relationships in a relational database.

6.6 KEYS AND FOREIGN KEYS

In a relational database, keys play a crucial role in ensuring the uniqueness of data and defining relationships between tables.
A primary key uniquely identifies each record within a relation, while a foreign key establishes a connection between related relations by referencing the primary key of another table. Together, they maintain entity integrity and referential integrity within the database system.

Primary Key

The primary key is an attribute (or a combination of attributes) that uniquely identifies each tuple in a relation.

- It must contain unique and non-null values.
- Each table should have exactly one primary key, which is underlined in schema representations.
- The primary key ensures that no two tuples in a table can have the same identifier.

Example: Student Table

StudentID	**Name**	**Age**	**Major**
1001	Alice	21	Computer Science
1002	Bob	22	Mathematics
1003	Carol	20	Physics

Here, StudentID is the Primary Key, uniquely identifying each student. It is often highlighted or underlined in the schema as:

Student(*StudentID*, Name, Age, Major)

Foreign Key

A foreign key is an attribute in one relation that refers to the primary key of another relation.

- It defines a referential relationship between two tables.
- The foreign key value in the referencing table must either match an existing primary key value in the referenced table or be NULL (if the relationship is optional).
- This constraint ensures consistency across tables, preventing orphaned or invalid references.

Example: Enrollment Table

StudentID	CourseID	Grade
1001	C101	A
1002	C102	B

In this example:

- StudentID is a Foreign Key referencing Student.StudentID.
- CourseID is a Foreign Key referencing Course.CourseID.

The Enrollment relation connects students to the courses they are enrolled in. Its relational schema can be expressed as:

Enrollment(*StudentID*, *CourseID*, Grade)

Foreign Keys:

- **StudentID → Student(StudentID)**
- **CourseID → Course(CourseID)**

6.7 SUMMARY

The **relational data model** provides a simple, logical, and mathematically sound framework for representing and managing data in databases. In this model, all data is organized into **relations (tables)** consisting of **tuples (rows)** and **attributes (columns)**, which together form the foundation of modern database systems.

- **Relations represent data** in a tabular structure, making it easy to understand, manipulate, and query. Each relation corresponds to an entity or a relationship type in the real world.**Keys** such as primary keys, candidate keys, and foreign keys ensure **uniqueness**, **identification**, and **linkages** among relations. They play a central role in maintaining **entity integrity** and **referential integrity**.
- **Integrity constraints** (domain, entity, and referential) preserve the **accuracy and consistency** of data, ensuring that only valid values are stored in the database.**Relational algebra** provides a set of formal operations—such as selection, projection, join, and set operations—that enable **powerful and flexible querying** of data.

- In conclusion, the relational model's structure, constraints, and operations together offer a robust foundation for building reliable, consistent, and efficient database systems, forming the basis for the widely used **Structured Query Language (SQL)** and modern relational database technologies.

6.8 TECHNICAL TERMS

Relation, Tuple, Attribute, Domain, Schema, Instance, Primary Key, Foreign Key, Super Key, Candidate Key, Selection, Projection, Join, Union.

6.9 SELF-ASSESSMENT QUESTIONS

Short-Answer Questions

1. Define a relation in the relational data model.
2. What is the difference between a relation schema and a relation instance?
3. How does a primary key differ from a candidate key?
4. What are domain constraints? Give one example.
5. Explain selection and join operations in relational algebra with simple examples.

Essay-Type Questions

1. Describe the structure and properties of a relation. Illustrate with an example.
2. Explain in detail the different types of keys used in a relational model. Provide suitable examples for each.
3. Discuss the integrity constraints in the relational model. How do they ensure data consistency?
4. Explain the basic relational algebra operations — selection, projection, join, and set operations — with examples.
5. Illustrate the process of mapping an ER model to a relational schema using the entities *Student*, *Course*, and *Enrollment*.

6.10 SUGGESTED READINGS

1. Elmasri, Ramez, and Shamkant B. Navathe. Fundamentals of Database Systems. Pearson Education.
2. C. J. Date. An Introduction to Database Systems. Addison-Wesley.
3. Korth, Henry F., Abraham Silberschatz, and S. Sudarshan. Database System Concepts. McGraw-Hill Education.
4. Ramakrishnan, Raghu, and Johannes Gehrke. Database Management Systems. McGraw-Hill Education.
5. Connolly, Thomas M., and Carolyn E. Begg. Database Systems: A Practical Approach to Design, Implementation, and Management. Pearson Education.

Dr. Kampa Lavanya

LESSON- 07

RELATIONAL DATABASE CONSTRAINTS

AIM

To understand the types of constraints that govern the validity and consistency of data in a relational database, and to explore how update operations and transactions interact with these constraints to maintain database integrity.

OBJECTIVES

After completing this topic, students should be able to:

1. Explain the concept and purpose of relational model constraints in maintaining data integrity.
2. Identify and describe different types of constraints, including domain constraints, entity integrity, and referential integrity.
3. Understand how relational database schemas define structure, relationships, and constraint specifications.
4. Discuss the role of update operations (insert, delete, and modify) in relational databases and their effect on constraints.
5. Analyze how transactions ensure database consistency through atomicity, consistency, isolation, and durability (ACID properties).
6. Recognize and handle constraint violations that occur during update operations using appropriate database mechanisms.
7. Apply theoretical concepts to design database schemas that enforce data integrity and consistency across multiple relations.

STRUCTURE

7.1 INTRODUCTION

The reliability of a relational database depends on its ability to maintain data accuracy, consistency, and validity at all times. This is achieved through a set of well-defined rules known as relational database constraints. These constraints ensure that the data stored in the database remains logically correct and consistent with the real-world entities it represents. Without such rules, databases could easily contain inconsistent, incomplete, or meaningless information.

In the relational model, constraints are an integral part of the database schema — they specify the conditions that data must satisfy to be considered valid. The most common types of constraints include domain constraints, entity integrity constraints, and referential integrity constraints. Together, they prevent invalid data entry, duplication, and broken relationships between tables.

In addition to these constraints, update operations (such as insert, delete, and modify) and transactions play a crucial role in maintaining database integrity. Each update must be checked for possible constraint violations, and transactions must ensure that all operations are executed in a consistent and reliable manner. When violations occur, the database management system (DBMS) must take corrective actions — such as rejecting the operation, cascading changes, or rolling back the transaction — to preserve the correctness of the data.

Thus, this lesson explores the nature of relational constraints, their enforcement through schemas, the impact of update operations, and the role of transactions in maintaining the overall integrity of a relational database.

7.2 Relational Model Constraints

In the **relational data model**, constraints are fundamental rules that ensure the **validity and consistency** of data stored in a database. They are part of the logical design of the database

and are enforced automatically by the **Database Management System (DBMS)**. These constraints define the conditions that every relation (table) must satisfy, thereby preventing invalid or inconsistent data from entering the system.

A **constraint** may apply to:

- Individual attributes (columns),
- Tuples (rows),
- Or relationships between multiple relations.

Relational model constraints can be broadly classified into three categories:
(1) Domain Constraints,
(2) Entity Integrity Constraints, and
(3) Referential Integrity Constraints.

7.2.1 Domain Constraints

Domain constraints specify that each attribute in a relation must take its value from a **predefined set of permissible values** — known as the *domain* of that attribute.
Each domain is associated with a particular data type (e.g., Integer, String, Date) and may include restrictions such as valid ranges or specific formats.

Example 7.1: Domain Constraint in the 'Student' Relation

Attribute	Domain	Valid Examples	Invalid Examples
StudentID	Integer (1000–9999)	1001, 1002	12, "A123"
Name	String (A–Z)	"Alice", "Bob"	1234, " "
Age	Integer (16–100)	20, 22	–5, 200
Major	String (Valid Majors)	"Physics", "Math"	"XYZ" (invalid major)

If a value entered for *Age* is 200 or *Name* is numeric, the **domain constraint** is violated. Such violations are automatically detected by the DBMS, which rejects the operation.

Key Points:

- Each attribute has a specific **domain** of allowed values.
- Domain constraints prevent **invalid, incomplete, or inconsistent data**.
- These are the most basic integrity rules in the relational model.

7.2.2 Entity Integrity Constraints

The **entity integrity constraint** ensures that every tuple (row) in a relation is **uniquely identifiable**.

This is achieved through the use of a **primary key** — an attribute (or combination of attributes) whose value must be:

- **Unique** for each tuple.
- **Non-null** (no missing values allowed).

This constraint guarantees that no two rows in a table represent the same real-world entity and that each record can be distinctly referenced.

Example 7.2: Entity Integrity in the 'Student' Relation

StudentID	Name	Age	Major
1001	Alice	21	Computer Science
1002	Bob	22	Mathematics
NULL ✗	Carol	20	Physics

Here, the third tuple violates **entity integrity** because the **primary key (StudentID)** is NULL — meaning the record cannot be uniquely identified.
The DBMS would reject this insertion.

Key Points:

- Primary key values must be **unique and non-null**.
- Ensures each entity (tuple) is distinguishable from others.
- Protects the logical integrity of the relation.

7.2.3 Referential Integrity Constraints

The **referential integrity constraint** ensures **consistency among related tables**. It states that a **foreign key** in one relation must either:

- Match a **primary key value** in another relation, or
- Be **NULL** (if the relationship is optional).

This rule ensures that references between relations remain valid — that is, there are **no "orphan" records** pointing to nonexistent entities.

Example 7.3: Referential Integrity Between Student and Enrollment Relations
Student Relation

StudentID	Name	Major
1001	Alice	Computer Science
1002	Bob	Mathematics

Enrollment Relation

StudentID	CourseID	Grade
1001	C101	A
1002	C102	B
1003 ✗	C103	A

Here, the *StudentID = 1003* in the **Enrollment** relation violates referential integrity because there is **no matching StudentID** in the **Student** relation. The DBMS will detect this inconsistency and reject the operation or raise an error.

Enforcement Rules:

When a referenced record (parent) is deleted or updated, the DBMS can respond in one of several ways:

1. **Reject** the operation (default action).
2. **Cascade** the changes to dependent tuples (e.g., delete related records automatically).
3. **Set NULL** for the foreign key in dependent tuples.
4. **Set Default** to a predefined key value.

Key Points:

- Ensures that relationships between tables remain valid.
- Prevents the creation of orphan records.
- Maintains **consistency** across related data.

7.2.4 Importance of Relational Constraints

Relational constraints play a central role in maintaining **data correctness** and **logical integrity** across the database.
They:

- Prevent entry of incorrect, missing, or contradictory data.
- Enforce valid relationships between entities.
- Enable **trustworthy data retrieval** and **accurate query results**.
- Form the foundation of **transaction integrity** and **database reliability**.

Summary of Constraint Types

Constraint Type	Purpose	Example of Violation
Domain	Attribute value must belong to its domain.	Age = 200
Entity Integrity	Primary key must be unique and non-null.	StudentID = NULL
Referential Integrity	Foreign key must match a valid primary key.	Enrollment.StudentID = 1003 (not in Student)

7.3 Relational Database Schemas

A **relational database schema** defines the **logical structure** of the entire database. It describes how data is organized into relations (tables), what attributes each relation contains, and the various **integrity constraints** that must hold true. Essentially, the schema serves as a **blueprint** for how data is stored, related, and managed in a relational database system.

The schema is specified during the **database design phase** and forms the foundation for creating and maintaining the actual database instance. Once defined, it helps ensure consistency, standardization, and integrity across all stored data.

7.3.1 Definition of a Database Schema

A **database schema** is a collection of relation schemas, each defining the structure of a relation.

A **relation schema** specifies:

- The **name** of the relation.
- The **names and domains** of its attributes.
- The **keys** and **constraints** associated with it.

Formally, a relation schema can be represented as:

$R(A_1, A_2, A_3, ..., A_n)$
where **R** is the name of the relation, and **$A_1, A_2, A_3, ..., A_n$** are the attributes.

Example 7.4: Relation Schemas for a University Database

1. Student(StudentID: Integer, Name: String, Age: Integer, Major: String)
2. Course(CourseID: String, Title: String, Credits: Integer)
3. Enrollment(StudentID: Integer, CourseID: String, Grade: String)

Here, the **Student**, **Course**, and **Enrollment** relations together form the **database schema** for the university system.

7.3.2 Database Schema vs. Database Instance

It is important to distinguish between a **schema** and an **instance**:

Aspect	Database Schema	Database Instance
Definition	The logical description or structure of the database.	The actual data stored in the database at a given point in time.
Nature	Relatively permanent (changes infrequently).	Changes frequently as data is inserted, deleted, or updated.
Example	Student(StudentID, Name, Age, Major)	(1001, "Alice", 21, "Computer Science")
Purpose	Defines data organization and constraints.	Represents the current state of the data.

Thus, the **schema** provides a stable design framework, while the **instance** represents the changing content of the database.

7.3.3 Role of Schemas in Constraint Enforcement

Database schemas not only define the **structure** of relations but also specify the **constraints** that maintain data integrity. These constraints are expressed as part of the schema definition using the **Data Definition Language (DDL)** in SQL.

Example 7.5: Schema with Constraints in SQL

```
CREATE TABLE Student (
  StudentID INTEGER PRIMARY KEY,
  Name VARCHAR(50) NOT NULL,
  Age INTEGER CHECK (Age BETWEEN 16 AND 100),
  Major VARCHAR(40)
);

CREATE TABLE Course (
  CourseID CHAR(5) PRIMARY KEY,
  Title VARCHAR(50),
  Credits INTEGER CHECK (Credits > 0)
);

CREATE TABLE Enrollment (
  StudentID INTEGER,
  CourseID CHAR(5),
  Grade CHAR(2),
  PRIMARY KEY (StudentID, CourseID),
  FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
  FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

In this example:

- **Entity integrity** is ensured through primary keys (*StudentID*, *CourseID*).
- **Domain constraints** restrict valid values for *Age* and *Credits*.
- **Referential integrity** is maintained through foreign keys linking *Enrollment* to *Student* and *Course*.

Thus, the schema enforces **structural correctness** and **logical consistency** within the database.

7.3.4 Schema Diagrams

A **schema diagram** is a graphical representation of the database schema that shows:

- Relations (tables),
- Attributes (columns),
- Primary keys (underlined or bolded),
- Foreign key relationships (shown using arrows).

Example: University Schema Diagram

```
[STUDENT] ----------------------< [ENROLLMENT] >---------------------- [COURSE]
 StudentID (PK)                    StudentID (FK)
 Name                              CourseID (FK)
 Age                               Grade
 Major
```

Explanation:

- The **Student** table has *StudentID* as its **primary key**.
- The **Course** table has *CourseID* as its **primary key**.
- The **Enrollment** table acts as a **bridge relation**, linking students and courses through **foreign keys**.
- Arrows indicate **referential dependencies** — *Enrollment.StudentID* → *Student.StudentID*, and *Enrollment.CourseID* → *Course.CourseID*.

Schema diagrams provide an at-a-glance view of how entities relate and how constraints are applied across relations.

7.3.5 Importance of Database Schemas

Relational database schemas are vital for the following reasons:

- **Define Structure:** Describe how data is logically organized into relations.
- **Ensure Integrity:** Enforce constraints and maintain consistency.
- **Provide Clarity:** Serve as a clear reference for database designers, developers, and users.
- **Support Modularity:** Enable changes to be made at the schema level without affecting applications directly.
- **Facilitate Query Design:** Help users understand data relationships and write accurate queries.

7.4 UPDATE OPERATIONS AND CONSTRAINTS

In a relational database, data stored in relations is not static — it constantly changes as new records are added, existing ones are modified, or obsolete data is removed. These changes are performed through **update operations**, which form the core of data manipulation in the relational model.

Each update operation must maintain the **integrity constraints** defined in the database schema. The **Database Management System (DBMS)** automatically checks these constraints whenever an update is attempted and either executes, modifies, or rejects the operation based on whether it preserves data consistency.

7.4.1 Types of Update Operations

The relational model supports three basic update operations:
INSERT, **DELETE**, and **UPDATE (MODIFY)**.
Each of these operations affects the data stored in relations and can potentially cause **constraint violations**.

(a) INSERT Operation

The **INSERT** operation adds a new tuple (record) into a relation.
When a new record is inserted, the DBMS must ensure that:

- All attribute values conform to **domain constraints**.
- The **primary key** value is **unique** and **non-null** (entity integrity).
- All **foreign key** values, if present, **match existing primary key values** in the referenced tables (referential integrity).

Example 7.6: Inserting a Tuple

INSERT INTO Student VALUES (1003, 'Carol', 20, 'Physics');
This operation will succeed only if:

- The *Age* value is within its valid domain.
- The *StudentID* does not already exist in the *Student* table.
- The *Major* value conforms to allowed entries.

Possible Violations:

- Inserting a *NULL* in a primary key field.
- Using an invalid *Age* value (outside domain).
- Referencing a non-existent record in another table.

If any constraint is violated, the DBMS **rejects** the insertion.

(b) DELETE Operation

The **DELETE** operation removes one or more tuples from a relation.
While simple in concept, it can cause **referential integrity violations** if the deleted tuple is referenced by foreign keys in other relations.

Example 7.7: Deleting a Tuple

DELETE FROM Student WHERE StudentID = 1001;
If *StudentID = 1001* is referenced in the *Enrollment* table, deleting this record would create an **orphaned reference**, violating referential integrity.

Possible Solutions:
To handle such situations, most DBMSs provide options:

1. **RESTRICT / NO ACTION:** Reject the deletion if it causes a violation.
2. **CASCADE:** Automatically delete all related tuples in dependent tables.
3. **SET NULL:** Replace the foreign key value in dependent records with *NULL*.
4. **SET DEFAULT:** Assign a predefined default value to the foreign key.

Example of CASCADE Deletion:

```
ALTER TABLE Enrollment
ADD CONSTRAINT FK_Student
FOREIGN KEY (StudentID)
REFERENCES Student(StudentID)
ON DELETE CASCADE;
```

This ensures that when a student record is deleted, all related enrollment records are also automatically removed.

(c) UPDATE (MODIFY) Operation

The **UPDATE** operation modifies attribute values in existing tuples.
Although it does not add or remove records, it can still violate **domain**, **entity**, or **referential** integrity constraints.

Example 7.8: Updating a Record
UPDATE Student
SET Age = 150
WHERE StudentID = 1002;
This violates the **domain constraint** since *Age = 150* falls outside the allowed range (16–100).

Another example:
UPDATE Enrollment
SET StudentID = 2000
WHERE CourseID = 'C101';

This may violate **referential integrity** if *StudentID = 2000* does not exist in the *Student* table.

Possible Violations:

- Updating a **primary key** may break links with foreign keys.
- Updating a **foreign key** may reference a non-existent record.
- Modifying a **domain attribute** may introduce invalid values.

The DBMS checks all constraints before applying the update and rejects any operation that causes inconsistency.

7.4.2 Effects of Update Operations on Constraints

Each update operation must preserve the following integrity rules:

Operation	Possible Violation	Constraint Affected
INSERT	Inserting duplicate or NULL primary key; inserting invalid domain value; inserting foreign key not in referenced table.	Entity Integrity, Domain, Referential Integrity
DELETE	Deleting a tuple referenced by another relation.	Referential Integrity
UPDATE	Modifying primary key or foreign key values; updating with invalid domain values.	Entity Integrity, Domain, Referential Integrity

The **DBMS** automatically checks each constraint and enforces them by rejecting the violating operation, modifying related records, or triggering user-defined actions.

7.4.3 Handling Constraint Violations
When a constraint violation occurs, the DBMS can respond in one of the following ways:

1. **Reject the Operation (Default):**

2. The update is canceled, and an error message is displayed to the user.
3. **Cascade the Changes:**
4. Changes are automatically propagated to maintain referential integrity. (e.g., deleting a student deletes all their enrollments.)
5. **Set NULL or Default:**
6. The foreign key values in dependent records are set to *NULL* or to a predefined default.
7. **Trigger Custom Action:**
   Database triggers can be defined to perform additional operations or validations before completing an update.

Example 7.9: Handling Violations using ON UPDATE CASCADE

```
ALTER TABLE Enrollment
ADD CONSTRAINT FK_Student
FOREIGN KEY (StudentID)
REFERENCES Student(StudentID)
ON UPDATE CASCADE;
```

If a student's *StudentID* changes, the corresponding *StudentID* values in *Enrollment* are automatically updated.

7.4.4 Importance of Enforcing Constraints During Updates

Maintaining integrity during update operations ensures:

- **Data consistency** across all relations.
- **Accuracy and validity** of the information stored.
- **Reliability** of query results and reports.
- **Protection** against accidental data loss or orphaned records.

Without constraint enforcement, the database may become inconsistent — leading to incorrect or misleading information that compromises decision-making and system reliability.

7.5 TRANSACTIONS AND DATABASE INTEGRITY

A **transaction** in a database is a **logical unit of work** that consists of one or more operations (such as insert, delete, or update) performed as a single, indivisible sequence. Transactions are essential for maintaining the **consistency and reliability** of the database, especially in multi-user environments where concurrent access occurs.

Each transaction must preserve the **integrity constraints** of the database and leave it in a **consistent state** — whether the transaction completes successfully or fails midway.

7.5.1 ACID Properties of Transactions

To ensure reliability, every transaction must satisfy the **ACID** properties:

1. **Atomicity:**
All operations within a transaction are executed completely or not at all.
If any operation fails, the entire transaction is rolled back.
2. **Consistency:**
The transaction must transform the database from one **valid state** to another, maintaining all integrity constraints.
3. **Isolation:**
Concurrent transactions must not interfere with each other.
Each transaction behaves as if it is executed independently.
4. **Durability:**
Once a transaction is committed, its effects are **permanent**, even in the event of a system failure.

7.5.2 Role in Maintaining Database Integrity

- Transactions ensure that **multiple related updates** are treated as a single consistent unit.
- If a constraint violation or system error occurs, the **rollback** mechanism restores the previous consistent state.
- Together with constraints, transactions safeguard **data correctness**, **consistency**, and **recoverability**.

Example:

When a student enrolls in a course, two operations occur:
1. Insert a record in *Enrollment*.
2. Update the student's record in *Student*.

If either fails, both must be undone — preserving consistency across relations.

7.6 Dealing with Constraint Violations

A **constraint violation** occurs when a database operation (such as insert, delete, or update) attempts to store or modify data in a way that breaks one or more **integrity rules** defined in the schema.

These violations can compromise the **consistency and reliability** of the database, so the **DBMS** must detect and handle them automatically.

7.6.1 Causes of Constraint Violations

Common causes include:
Inserting values **outside the defined domain** (domain violation).
Attempting to insert or update a **NULL or duplicate primary key** (entity integrity violation).
Deleting or modifying a tuple referenced by another table (referential integrity violation).
Updating a **foreign key** to a non-existent value in the parent table.

7.6.2 Handling Constraint Violations
When a violation occurs, the DBMS can respond in several ways:

1. **Reject the Operation:**
   The default action — the DBMS cancels the operation and reports an error.
2. **Cascade the Changes:**
   Automatically applies the same action to dependent tuples (e.g., deleting all related records).

3. Set NULL or Default:
Replaces invalid foreign key values with NULL or a default value to maintain consistency.

4. Rollback the Transaction:
Reverses all changes made by the transaction to restore the database to its previous valid state.

7.6.3 Importance of Proper Handling

Effective handling of constraint violations ensures that:

- The database remains **accurate and logically consistent**.
- **Data dependencies** across tables are preserved.
- Users and applications receive **predictable, reliable behavior** from the database system.

7.7 SUMMARY

The **relational database constraints** play a crucial role in ensuring that the data stored within a relational database is **accurate, consistent, and logically valid**. These constraints define the rules that govern data entry, update, and deletion, thereby maintaining the overall **integrity** of the database.

- **Relational model constraints**—such as domain, entity integrity, and referential integrity—ensure the correctness of data values and relationships between tables.
- **Database schemas** define the logical structure of relations and embed constraints to enforce data validity at the structural level.
- **Update operations** (insert, delete, modify) can affect data integrity; therefore, the DBMS checks and enforces constraints during these operations.
- **Transactions**, governed by ACID properties, ensure that complex or multi-step operations maintain database consistency even in the presence of errors or concurrent access.
- Proper handling of **constraint violations**—through rejection, cascading, setting defaults, or rollback—prevents inconsistencies and data loss.

In conclusion, relational constraints, update operations, and transaction management together ensure that a database remains **trustworthy, consistent, and dependable**, providing a solid foundation for all data-driven applications.

7.8 TECHNICAL TERMS

- Constraint
- Domain Constraint
- Entity Integrity
- Referential Integrity
- Schema
- Instance
- Transaction
- ACID Properties
- Constraint Violation
- Rollback

7.9 SELF-ASSESSMENT QUESTIONS

Short-Answer Questions

1. Define a **relational model constraint** and give one example.
2. What is the purpose of a **domain constraint** in a relation?
3. State the difference between **entity integrity** and **referential integrity**.
4. List the three basic **update operations** in a relational database.
5. What does **atomicity** mean in the context of transactions?

Essay-Type Questions

1. Explain the **different types of relational model constraints** with suitable examples.
2. Describe how **update operations** can cause constraint violations. How can these violations be handled by the DBMS?
3. Discuss the **ACID properties** of transactions and their importance in maintaining database integrity.
4. Explain the **role of schemas** in defining and enforcing database constraints.
5. Write short notes on **constraint violations** and describe how cascading, setting NULL, or rollback actions maintain data consistency.

7.10 SUGGESTED READINGS

1. "Database System Concepts" by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
2. "Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe
3. "Database Management Systems" by Raghu Ramakrishnan and Johannes Gehrke
4. "An Introduction to Database Systems" by C.J. Date
5. "SQL and Relational Theory: How to Write Accurate SQL Code" by C.J. Date
6. Elmasri, R., & Navathe, S. B. (2010). Fundamentals of Database Systems (6th ed.). Addison-Wesley.
7. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). Database System Concepts (6th ed.). McGraw-Hill.

Dr. Kampa Lavanya

LESSON- 08

THE RELATIONAL ALGEBRA

AIMS AND OBJECTIVES

The aim of this chapter is to provide a comprehensive understanding of the **formal query languages** used in relational database systems in terms of **Relational Algebra** . These languages form the theoretical backbone of relational data manipulation and query processing, offering a foundation for understanding and applying practical query languages like SQL. By exploring the principles of relational databases, the chapter seeks to bridge the gap between theoretical concepts and their real-world applications in database systems.

At the end of the lesson student will be able to,

1. Understand Formal Languages in terms of Relational Algebra
2. describe and apply unary operations like SELECT and PROJECT and binary operations like JOIN, UNION, and DIVISION.
3. demonstrate how relational algebra operations translate into SQL constructs

STRUCTURE

8.1. INTRODUCTION

Relational databases are at the heart of modern data storage and retrieval systems, and their theoretical foundations lie in **Relational Algebra** . These formal query languages provide a mathematical framework for representing and manipulating data in a structured format. **Relational Algebra** is a procedural language that defines the step-by-step process to retrieve data.

This chapter delves into the essential operations of relational algebra, including unary and binary operations such as SELECT, PROJECT, JOIN, and DIVISION, as well as set-based operations like UNION and INTERSECTION. By understanding these theoretical constructs,

database practitioners can design more efficient systems and queries, bridging the gap between abstract mathematical concepts and real-world database implementations.

8.2 UNARY RELATIONAL OPERATIONS

❖ **SELECT (σ):** SELECT operation is used to select a subset of the tuples from a relation that satisfy a selection condition. It is a filter that keeps only those tuples that satisfy a qualifying condition – those satisfying the condition are selected while others are discarded.

- **Example-1:**
  - σ SALARY > 30,000 (EMPLOYEE) retrieves employees earning more than 30,000.
  - σDNO = 4 (EMPLOYEE)
- Can handle complex conditions using logical operators (AND, OR, NOT).
- In general, the select operation is denoted by σ < selection condition > (R) where the symbol σ (sigma) is used to denote the select operator, and the selection condition is a Boolean expression specified on the attributes of relation R
- **Example-2:**
  - σ (DNO=4 AND SALARY > 25000) OR (DNO=5 AND SALARY > 30000) (EMPLOYEE)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
Jennifer		Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
Ramesh		Narayan	666884444	1962-09-15	975 FireOak,Humble,TX	M	38000	333445555	5

Fig 8.1 The query result of Example 2

❖ **PROJECT (π):** This operation selects certain columns from the table and discards the other columns. The PROJECT creates a vertical partitioning – one with the needed columns (attributes) containing results of the operation and other containing the discarded Columns.

- **Example-3:** To list each employee's first and last name and salary, the following is used:
  - π FNAME, LNAME, SALARY(EMPLOYEE) retrieves names and salaries of employees.
- The general form of the project operation is π<attribute list >(R) where π (pi) is the symbol used to represent the project operation and <attribute list > is the desired list of attributes from the attributes of relation R.
- The project operation removes any duplicate tuples, so the result of the project operation is a set of tuples and hence a valid relation.
- **Example-4:** π SEX, SALARY(EMPLOYEE)

LNAME	FNAME	SALARY
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

SEX	SALARY
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Fig 8.2 The query result of Example 3 & 4

❖ **RENAME (ρ)**: We may want to apply several relational algebra operations one after the other. Either we can write the operations as a single relational algebra expression by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results.

Example: ρ(NEWNAME(EMPLOYEE)) renames the Employee relation to NewName.

8.3 RELATIONAL ALGEBRA OPERATIONS FROM SET THEORY

Relational algebra incorporates operations derived from set theory to manipulate and query relational data. These operations operate on relations (tables) treated as sets of tuples. Key set-theoretic operations include:

1. UNION Operation
2. INTERSECTION Operation
3. Set Difference (or MINUS) Operation
4. CARTESIAN Operation

❖ UNION Operation

The result of this operation, denoted by R ∪ S, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
DEP5_EMPS ← σDNO=5 (EMPLOYEE)
RESULT1 ← π SSN(DEP5_EMPS)
RESULT2(SSN) ← π SUPERSSN(DEP5_EMPS)
RESULT ← RESULT1 ∪ RESULT2

The union operation produces the tuples that are in either RESULT1 or RESULT2 or both. The two operands must be ―type compatible‖.

STUDENT	FN	LN
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

INSTRUCTOR	FNAME	LNAME
	John	Smith
	Ricardo	Browne
	Susan	Yao
	Francis	Johnson
	Ramesh	Shah

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

Fig 8.1 Union Operation among STUDENT and INSTRUCTOR Relations

❖ INTERSECTION Operation

The result of this operation, denoted by R ∩ S, is a relation that includes all tuples that are in both R and S. The two operands must be "type compatible".

FN	LN
Susan	Yao
Ramesh	Shah

Fig 8.2 Intersection Operation among STUDENT and INSTRUCTOR Relations

- **Set Difference (or MINUS) Operation**

The result of this operation, denoted by R - S, is a relation that includes all tuples that are in R but not in S. The two operands must be "type compatible‖.

Example: The figure shows the names of students who are not instructors, and the names of instructors who are not students.

STUDENT	FN	LN
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

STUDENT-INSTRUCTOR

FNAME	LNAME
John	Smith
Ricardo	Browne
Francis	Johnson

INSTRUCTOR-STUDENT

Fig 8.3 Set Difference Operation among STUDENT and INSTRUCTOR Relations

- Notice that both union and intersection are commutative operations; that is

  $R \cup S = S \cup R$, and $R \cap S = S \cap R$
- Both union and intersection can be treated as n-ary operations applicable to any number of relations as both are associative operations; that is

  $R \cup (S \cup T) = (R \cup S) \cup T$, and $(R \cap S) \cap T = R \cap (S \cap T)$
- The minus operation is not commutative; that is, in general

  $R - S \neq S - R$

❖ **CARTESIAN (or cross product) Operation**

This operation is used to combine tuples from two relations in a combinatorial fashion.

- In general, the result of R (A1 , A2 , . . ., An) x S(B1 , B2 , . . ., Bm) is a relation Q with degree n + m attributes Q(A1 , A2 , . . ., An , B1 , B2 , . . ., Bm), in that order.
- The resulting relation Q has one tuple for each combination of tuples—one from R and one from S.
- Hence, if R has nR tuples (denoted as |R| = nR), and S has nS tuples, then | R x S | will have nR * nS tuples.
- The two operands do NOT have to be "type compatible".

Example:
FEMALE_EMPS ← σ SEX='F'(EMPLOYEE)
EMPNAMES ← π FNAME, LNAME, SSN (FEMALE_EMPS)
EMP_DEPENDENTS ← EMPNAMES x DEPENDENT

FEMALE_EMPS

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle,Spring,TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice,Houston,TX	F	25000	333445555	5

EMPNAMES

FNAME	LNAME	SSN
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28

RESULT

FNAME	LNAME	DEPENDENT_NAME
Jennifer	Wallace	Abner

Fig 8.4 CATESIAN Product Operation Example.

8.4 BINARY RELATIONAL OPERATIONS: JOIN AND DIVISION

Binary relational operations work on two relations to produce a new relation. Among these, JOIN and DIVISION are crucial for relational algebra due to their unique roles in querying and manipulating data.

❖ JOIN Operation

The sequence of cartesian product followed by select is used quite commonly to identify and select related tuples from two relations, a special operation, called JOIN. This operation is very important for any relational database with more than a single relation, because it allows us to process relationships among relations.

The general form of a join operation on two relations R(A1 , A2 , . . ., An) and S(B1 , B2 , . . ., Bm) is:

R ⋈ <join condition> S

where R and S can be any relations that result from general relational algebra expressions.

Example: Suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each DEPARTMENT tuple with the EMPLOYEE tuple whose SSN value matches the MGRSSN value in the department tuple.

$$\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \underset{\text{MGRSSN=SSN}}{\bowtie} \text{EMPLOYEE}$$

DEPT_MGR	DNAME	DNUMBER	MGRSSN	. . .	FNAME	MINIT	LNAME	SSN	. . .
	Research	5	333445555	. . .	Franklin	T	Wong	333445555	. . .
	Administration	4	987654321	. . .	Jennifer	S	Wallace	987654321	. . .
	Headquarters	1	888665555	. . .	James	E	Borg	888665555	. . .

Fig 8.5 Result of Join Operation of above example.

- **EQUIJOIN Operation**

The most common use of join involves join conditions with equality comparisons only. Such a join, where the only comparison operator used is =, is called an EQUIJOIN. In the result of an EQUIJOIN we always have one or more pairs of attributes (whose names need not be identical) that have identical values in every tuple. The JOIN seen in the previous example was EQUIJOIN.

- **NATURAL JOIN Operation**

Because one of each pair of attributes with identical values is superfluous, a new operation called natural join—denoted by *—was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition. The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes, have the same name in both relations. If this is not the case, a renaming operation is applied first.

Example: To apply a natural join on the DNUMBER attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write:

DEPT_LOCS	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	LOCATION
	Headquarters	1	888665555	1981-06-19	Houston
	Administration	4	987654321	1995-01-01	Stafford
	Research	5	333445555	1988-05-22	Bellaire
	Research	5	333445555	1988-05-22	Sugarland
	Research	5	333445555	1988-05-22	Houston

Fig 8.6 Result of Natural Join Operation of above example.

Outer Join:

- Extends JOIN by including tuples from one or both relations that do not satisfy the join condition, filling unmatched attributes with NULL.

- **Left Outer Join**: Includes unmatched tuples from the left relation.
  $R \bowtie S=(R \bowtie S) \cup (R - \pi A,B(R \bowtie S))$

TEMP ←− (EMPLOY EE $\supset\bowtie_{SSN=MGRSSN}$ DEPARTMENT)
RESULT ←− $\pi_{F\ NAME, MINIT, DNAME}$ (TEMP)

- **Right Outer Join**: Includes unmatched tuples from the right relation.
  $R \bowtie S=(R \bowtie S) \cup (S - \pi B,C(R \bowtie S))$
- **Full Outer Join**: Includes unmatched tuples from both relations.
  $R \bowtie S=(R \bowtie S) \cup (R - \pi A,B(R \bowtie S)) \cup (S - \pi B,C(R \bowtie S))$

❖ DIVISION Operation

The **DIVISION** operation is used when a relation RRR (dividend) is divided by another relation SSS (divisor). It returns a relation containing tuples from RRR that are associated with all tuples in SSS.

Conditions for Division:

- RRR must have all attributes of SSS, and may have additional attributes.
- The result contains these additional attributes, retaining tuples that satisfy the division.

Example for DIVISION operation:

- "Retrieve the names of employees who work on all the projects that 'John Smith' works on. JSMITH_SSN(ESSN) ←− πSSN (σF NAME='John' AND LNAME='Smith' (EMPLOYEE)) JSMITH_PROJ ←− πP NO (JSMITH SSN ∗ WORKS_ON)
WORKS_ON2 ←− πESSN, P_NO (WORKS_ON)
DIV_HERE(SSN) ←− WORKS_ON2 ÷ JSMITH PROJ
RESULT ←− πF NAME, LNAME (EMPLOYEE ∗ DIV HERE)

UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Fig 8.7 Relational Algebra Operations

8.5 ADDITIONAL RELATIONAL OPERATIONS

Beyond the core operations (SELECT, PROJECT, UNION, JOIN, etc.), relational algebra includes **additional operations** that enhance querying and manipulation capabilities in relational databases.

❖ Aggregate Functions

A type of request that cannot be expressed in the basic relational algebra is to specify mathematical aggregate functions on collections of values from the database. – Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples. These functions are used in simple statistical queries that summarize information from the database tuples.

Common functions applied to collections of numeric values include

- SUM: Calculates the sum of values in a column.
- AVG: Computes the average of values.
- COUNT: Counts the number of tuples.
- MIN: Finds the minimum value in a column.
- MAX: Finds the maximum value in a column.

(a)

R	DNO	NO_OF_EMPLOYEES	AVERAGE_SAL
	5	4	33250
	4	3	31000
	1	1	55000

(b)

DNO	COUNT_SSN	AVERAGE_SALARY
5	4	33250
4	3	31000
1	1	55000

(c)

COUNT_SSN	AVERAGE_SALARY
8	35125

Fig 8.8 Result of AVG Aggregate Operation

❖ OUTER JOIN

- **LEFT OUTER JOIN:** R3(A1, A2, . . . , An, B1, B2, . . . , Bm) ←− R1(A1, A2, . . . , An) ⟕ <JOIN COND> R2(B1, B2, . . . , Bm)

✓ This operation keeps every tuple t in left relation R1 in R3, and fills "NULL" for attributes B1, B2, . . . , Bm if the join condition is not satisfied for t.

✓ **Example,**

TEMP ←− (EMPLOYEE ⟕ SSN=MGRSSN DEPARTMENT)

RESULT ←− π F NAME, MINIT, DNAME (TEMP)

- **RIGHT OUTER JOIN**: similar to LEFT OUTER JOIN, but keeps every tuple t in right relation R2 in the resulting relation R3.

$R \bowtie S=(R \bowtie S) \cup (S - \pi_{B,C}(R \bowtie S))$

- **FULL OUTER JOIN**: Includes unmatched tuples from both relations.

$R \bowtie S=(R \bowtie S) \cup (R - \pi_{A,B}(R \bowtie S)) \cup (S - \pi_{B,C}(R \bowtie S))$

❖ **The OUTER UNION Operation**

- **OUTER UNION:** make union of two relations that are partially compatible.
  - R3(A1, A2, . . . , An, B1, B2, . . . , Bm, C1, C2, . . . , Cp) ←− R1(A1, A2, . . . , An, B1, B2, . . . , Bm) OUTER UNION R2(A1, A2, . . . , An, C1, C2, . . . , Cp)
  - The list of compatible attributes are represented only once in R3.
  - Tuples from R1 and R2 with the same values on the set of compitable attributes are represented only once in R3
  - In R3, fill "NULL" if necessary
  - **Example** STUDENT (NAME, SSN, DEPT, ADVISOR) and FACULTY(NAME, SSN, DEPT, RANK)

The resulting relation schema after OUTER UNION will be R 3(NAME, SSN, DEPT, ADVISOR, RANK)

8.6 EXAMPLES OF QUERIES IN RELATIONAL ALGEBRA

- Retrieve the name and address of all employees who work for the 'Research' department.
- For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.
- Find the names of employees who work on all the projects controlled by department
- number 5.
- Make a list of project numbers for projects that involve an employee whose last name
- is 'Smith', either as a worker or as a manager of the department that controls the
- project.
- List the names of all employees who have two or more dependents.
- Retrieve the names of employees who have no dependents.
- List the names of managers who have at least one dependent.
- Retrieve the name of each employee who has a dependent with the same first name
- and same sex as the employee.
- Retrieve the names of all employees who do not have supervisors.
- Find the sum of salary of all employees, the maximum salary, the minimum salary, and
- the average salary for each department.

- Retrieve the name and address of all employees who work for the 'Research' department.

  $RESEARCH\_DEPT \longleftarrow \sigma_{DNAME='Research'}(DEPARTMENT)$

  $RESEARCH\_EMPS \longleftarrow (RESEARCH\_DEPT \bowtie_{DNUMBER=DNO} (EMPLOYEE))$

  $RESULT \longleftarrow \pi_{FNAME,LNAME,ADDRESS}(RESEARCH\_EMPS)$

- For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

  $STAFFORD\_PROJS \longleftarrow \sigma_{PLOCATION='Stafford'}(PROJECT)$

  $CONTR\_DEPT \longleftarrow (STAFFORD\_PROJS \bowtie_{DNUM=DNUMBER} (DEPARTMENT))$

  $PROJ\_DEPT\_MGR \longleftarrow (CONTR\_DEPT \bowtie_{MGRSSN=SSN} (EMPLOYEE))$

  $RESULT \longleftarrow \pi_{PNUMBET,DNUM,LNAME,ADDRESS,BDATE}(PROJ\_DEPT\_MGR)$

- Find the names of employees who work on all the projects controlled by department number 5.

  $DEPT5\_PROJS(PNO) \longleftarrow \pi_{PNUMBER}(\sigma_{DNUM=5}(PROJECT))$

  $EMP\_PROJ(SSN, PNO) \longleftarrow \pi_{ESSN,PNO}(WORKS\_ON)$

  $RESULT\_EMP\_SSNS \longleftarrow EMP\_PROJ \div DEPT\_PROJS$

  $RESULT \longleftarrow \pi_{LNAME,FNAME}(RESULT\_EMP\_SSNS * EMPLOYEE)$

- Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

  $SMITHS(ESSN) \longleftarrow \pi_{SSN}(\sigma_{LNAME='Smith'}(EMPLOYEE))$

  $SMITH\_WORKER\_PROJ \longleftarrow \pi_{PNO}(WORKS\_ON * SMITHS)$

  $MGRS \longleftarrow \pi_{LNAME,DNUMBER}(EMPLOYEE \bowtie_{SSN=MGRSSN} DEPARTMENT)$

  $SMITH\_MANAGED\_DEPTS(DNUM) \longleftarrow \pi_{DNUMBER}(\sigma_{LNAME='Smith'}(MGRS))$

  $SMITH\_MGR\_PROJS(PNO) \longleftarrow \pi_{PNUMBER}(SMITH\_MANAGED\_DEPTS * PROJECT)$

  $RESULT \longleftarrow (SMITH\_WORKER\_PROJS \cup SMITH\_MGR\_PROJS)$

- List the names of all employees who have two or more dependents.

  $T_1(SSN, NO\_OF\_DEPTS) \longleftarrow_{ESSN} \Im_{COUNT\ DEPENDENT\_NAME}(DEPENDENT)$

  $T_2 \longleftarrow \sigma_{NO\_OF\_DEPTS \geq 2}(T_1)$

  $RESULT \longleftarrow \pi_{LNAME,FNAME}(T_2 * EMPLOYEE)$

- Retrieve the names of employees who have no dependents.

$$ALL\_EMPS \longleftarrow \pi_{SSN}(EMPLOYEE)$$
$$EMPS\_WITH\_DEPS(SSN) \longleftarrow \pi_{ESSN}(DEPENDENT)$$
$$EMPS\_WITHOUT\_DEPS \longleftarrow (ALL\_EMPS - EMPS\_WITH\_DEPS)$$
$$RESULT \longleftarrow \pi_{LNAME,FNAME}(EMPS\_WITHOUT\_DEPS * EMPLOYEE)$$

- List the names of managers who have at least one dependent.

$$MGRS(SSN) \longleftarrow \pi_{MGRSSN}(DEPARTMENT)$$
$$EMPS\_WITH\_DEPS(SSN) \longleftarrow \pi_{ESSN}(DEPENDENT)$$
$$MGRS\_WITH\_DEPS \longleftarrow (MGRS \cap EMPS\_WITH\_DEPS)$$
$$RESULT \longleftarrow \pi_{LNAME,FNAME}(MGRS\_WITH\_DEPS * EMPLOYEE)$$

- Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

$$EMPS\_DEPS \longleftarrow$$
$$(EMPLOYEE \bowtie_{SSN=ESSN \text{ AND } SEX=SEX \text{ AND } FNAME=DEPENDENT\_NAME} DEPENDENT)$$
$$RESULT \longleftarrow \pi_{LNAME,FNAME}(EMPS\_DEPS)$$

- Retrieve the names of all employees who do not have supervisors.

$$RESULT \longleftarrow \pi_{LNAME,FNAME}(\sigma_{SUPERSSN=NULL}(EMPLOYEE))$$

- Find the sum of salary of all employees, the maximum salary, the minimum salary, and the average salary for each department.

$$RESULT \longleftarrow$$
$$_{DNO}\Im_{SUM\ SALARY,\ MAXIMUM\ SALARY,\ MINIMUM\ SALARY,\ AVERAGE\ SALARY}(EMPLOYEE)$$

8.7 SUMMARY

Relational Algebra explores foundational query languages in relational database theory. It begins with unary relational operations, SELECT and PROJECT, which allow filtering rows and choosing specific attributes, respectively. Set-theoretic operations like UNION, INTERSECTION, and DIFFERENCE are covered for combining and comparing relations. Binary relational operations such as JOIN (including natural joins and outer joins) and DIVISION enable complex queries by relating to tuples across tables. Additional operations like aggregation and renaming further enhance query capabilities.

8.8 TECHNICAL TERMS

- Relational Algebra
- SELECT
- PROJECT

- JOIN
- RENAME
- Existential Quantifier
- Universal Quantifier

8.9 SELF-ASSESSMENT QUESTIONS

Short Questions

1. Define the SELECT operation in relational algebra and its purpose.
2. What is the difference between PROJECT and SELECT in relational algebra?
3. Explain the purpose of the CARTESIAN PRODUCT operation in relational algebra.
4. Describe the division operation in relational algebra and give one practical use case.

Long Questions

1. Explain with examples how set-theoretic operations (UNION, INTERSECTION, and DIFFERENCE) are used in relational algebra.
2. Describe binary relational operations with a focus on different types of joins (natural join, equijoin, and outer join) and their applications.
3. Discuss the role of additional relational operations such as aggregation and renaming in query optimization and simplification.
4. Write a detailed query using relational algebra to find customers who have accounts in every branch of a bank and explain each step.

8.10 SUGGESTED READINGS

1. Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM, 13(6), 377-387.
2. Date, C. J. (2003). "An Introduction to Database Systems." 8th Edition. Addison-Wesley.
3. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). "Database System Concepts." 6th Edition. McGraw-Hill.
4. Ullman, J. D., & Widom, J. (2008). "A First Course in Database Systems." 3rd Edition. Pearson.

Dr. Kampa Lavanya

LESSON- 09

THE RELATIONAL CALCULUS

AIMS AND OBJECTIVES

The aim of this chapter is to provide a comprehensive understanding of the **formal query languages** used in relational database systems in terms of **Relational Calculus**. Relational calculus is a non-procedural query language used in database management systems (DBMS). Its objectives include:

1. **Declarative Query Specification**:
   - Provide a means to describe what data to retrieve without specifying how to retrieve it.
   - Focus on the "what" rather than the "how" by defining desired results using logical predicates.

2. **Foundation for Query Languages**:
   - Serve as a theoretical basis for SQL and other high-level query languages.
   - Ensure that query languages are both expressive and robust.

3. **Support for Logical Reasoning**:
   - Allow users to express queries using logical expressions and constraints.
   - Enable reasoning about data relationships and structures.

4. **Abstraction**:
   - Hide the complexities of query execution by focusing on the end result.
   - Enable users to write queries without needing to understand the underlying physical database design.

5. **Data Integrity and Consistency**:
   - Facilitate querying in a way that respects the database's logical consistency.
   - Ensure that queries operate within the constraints and relationships defined by the database schema.

These languages form the theoretical backbone of relational data manipulation and query processing, offering a foundation for understanding and applying practical query languages like SQL. By exploring the principles of relational databases, the chapter seeks to bridge the gap between theoretical concepts and their real-world applications in database systems.

At the end of the lesson students will be able to:

1. Understand Ability to Formulate Complex Queries
2. Enhanced Understanding of Query Languages
3. Know Logical and Formal Thinking
4. Work on Validation of Query Equivalence
5. develop the ability to write and analyze queries in Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC), emphasizing logical expressions.

STRUCTURE

9.1. INTRODUCTION

Relational calculus is a non-procedural query language in database management systems (DBMS) that allows users to specify what data they want to retrieve without dictating how to retrieve it. Rooted in mathematical logic, relational calculus uses declarative expressions to define queries, focusing on logical relationships and constraints within the data. It contrasts with relational algebra, which is procedural, by emphasizing the "what" rather than the "how" of query formulation. Relational calculus serves as a theoretical foundation for high-level query languages like SQL, making it an essential concept in database theory and design. Two primary forms of relational calculus are Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC). TRC focuses on specifying queries using variables that represent tuples in a relation, with logical predicates to filter the desired tuples. In contrast, DRC uses variables that represent individual domain values rather than entire tuples, allowing for a more granular approach to query definition. Both forms rely heavily on logical expressions, such as existential and universal quantifiers, to describe data constraints and retrieval criteria. Together, TRC and DRC provide powerful frameworks for expressing complex database queries in a logical and non-procedural manner.

Relational Calculus is declarative, focusing on specifying what data to retrieve without detailing the process. This chapter, additionally, it explores the principles of **Tuple Relational Calculus (TRC)** and **Domain Relational Calculus (DRC)**, emphasizing logical expressions to define queries. By understanding these theoretical constructs, database practitioners can design more efficient systems and queries, bridging the gap between abstract mathematical concepts and real-world database implementations.

9.2 THE TUPLE RELATIONAL CALCULUS

Tuple Calculus and Domain Calculus are two formal query languages for relational databases. They are declarative, meaning they specify *what* data to retrieve without defining *how* to retrieve it, unlike procedural languages such as relational algebra.

❖ **The Tuple Relational Calculus**

- **Nonprocedural Language:** Specify what to do; Tuple (Relational) Calculus, Domain (Relational) Calculus.

- **Procedural Language:** Specify how to do; Relational Algebra.
- The expressive power of Relational Calculus and Relational Algebra is identical.
- A relational query language L is considered relationally complete if we can express in L any query that can be expressed in Relational Calculus.
- Most relational query language is relationally complete but have more expressive power than relational calculus (algebra) because of additional operations such as aggregate functions, grouping, and ordering.

Queries in TRC are expressions of the form:

$$\{t \mid P(t)\}$$

where:

- t is a tuple variable.
- P(t) is a predicate that describes the conditions the tuples must satisfy.

A general expression form:

- $\{t_1.A_1, t_2.A_2, \ldots, t_n.A_n \mid COND(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$

Where t1,t2, . . . ,tn,tn+1,tn+2, . . . ,tn+m are tuple variables, each Ai is an attribute of the relation on which ti ranges, and COND is a condition or formula

Example:
Find all employees working in the "HR" department:

$$\{t \mid t \in EMPLOYEE \wedge t.dept =' HR'\}$$

A formula is made up one or more atoms connected via the logical operators and, or, not and is defined recursively as follows.

- Every atom is a formula.
- If F_1 and F_2 are formulas, then so are $(F_1 \; and \; F_2)$, $(F_1 \; or \; F_2)$, $not(F_1)$, $not(F_2)$.

  And

  * $(F_1 \; and \; F_2)$ is TRUE if both F_1 and F_2 are TRUE; otherwise, it is FALSE.
  * $(F_1 \; or \; F_2)$ is FALSE if both F_1 and F_2 are FALSE; otherwise, it is TRUE.
  * $not(F_1)$ is TRUE if F_1 is FALSE; it is FALSE if F_1 is TRUE.
  * $not(F_2)$ is TRUE if F_2 is FALSE; it is FALSE if F_2 is TRUE.

➢ **The Existential and Universal Quantifiers**

In relational calculus, **existential** and **universal quantifiers** are essential components of logical expressions used to define query conditions. These quantifiers enable precise specification of constraints and relationships in database queries. The **existential quantifier (∃)** asserts the existence of at least one value or tuple that satisfies a given condition. It is commonly used to verify if there is some data in the database that meets specific criteria. Conversely, the **universal quantifier (∀)** specifies that a condition must hold true for all values or tuples in a given domain or relation. By incorporating these quantifiers, relational calculus

allows for expressive and flexible query formulations, enabling users to retrieve data with high precision while adhering to logical principles.

Existential (∃) and universal (∀) quantifiers are fundamental concepts in logic and relational calculus. They are used to specify conditions for sets of values in queries or logical expressions.

- **Existential Quantifier (∃):** Used to assert that *at least one* element in a domain satisfies a given condition.

Syntax:

$$\exists x\ (P(x))$$

This means there is at least one xxx for which the predicate P(x)P(x)P(x) is true.

Example in Tuple Relational Calculus (TRC):
Find employees working in the "HR" department:

$$\{t \mid t \in EMPLOYEE \land \exists d\ (t.dept = d \land d =' HR')\}$$

- **Universal Quantifier (∀) :** Used to assert that a condition applies to all elements in a domain.

Syntax:

$$\forall x\ (P(x))$$

This means the predicate P(x)P(x)P(x) is true for every xxx in the domain.

Example in Tuple Relational Calculus (TRC):
Find employees who are in all departments:

$$\{t \mid t \in EMPLOYEE \land \forall d\ (d \in DEPARTMENT \implies t.dept = d)\}$$

➢ **Comparison Between Existential and Universal Quantifiers**

Aspect	Existential Quantifier (∃)	Universal Quantifier (∀)
Definition	Asserts the existence of at least one element that satisfies a condition.	States that a condition must be true for all elements in a domain.
Symbol	Represented by `∃` (e.g., ∃x P(x))	Represented by `∀` (e.g., ∀x P(x))
Usage	Used when querying if there is at least one match in a dataset.	Used to ensure a condition holds for every element in a dataset.
Logical Meaning	"There exists" or "at least one."	"For all" or "every."
Example in TRC	∃t (t ∈ Employee ∧ t.salary > 50000): Checks if there exists an employee earning more than $50,000.	∀t (t ∈ Employee → t.age > 18): Ensures all employees are older than 18.
Nature	Verifies the presence of specific instances.	Verifies the universality of a condition.
Common Scenarios	Finding whether a particular condition is met at least once.	Ensuring that a rule or constraint is consistently followed.
Negation Relation	The negation of ∃x P(x) is ∀x ¬P(x).	The negation of ∀x P(x) is ∃x ¬P(x).
Complexity	Simpler to evaluate as it stops upon finding one matching instance.	Requires checking every instance, making it computationally more intensive.

Both quantifiers are critical in relational calculus, enabling the formulation of comprehensive and logically precise queries. They complement each other, with existential quantifiers being ideal for selective queries and universal quantifiers ensuring universal conditions are met.

➢ **Example Queries Using the Existential Quantifier**

- Retrieve the name and address of all employees who work for the 'Research' department.
  $\{t.FNAME, t.LNAME, t.ADDRESS \mid EMPLOYEE(t) \text{ and } (\exists d)$
  $(DEPARTMENT(d) \text{ and } d.DNAME = '\text{Research}' \text{ and } d.DNUMBER = t.DNO)\}$
- For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birthdate, and address.
  $\{p.PNUMBER, p.DNUM, m.LNAME, m.BDATE, m.ADDRESS \mid PROJECT(p)$
  $\text{and } EMPLOYEE(m) \text{ and } p.PLOCATION = '\text{Stafford}' \text{ and}$
  $((\exists d)(DEPARTMENT(d) \text{ and } p.DNUM = d.DNUMBER \text{ and } d.MGRSSN = m.SSN))\}$
- For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.
  $\{e.FNAME, e.LNAME, s.FNAME, s.LNAME \mid EMPLOYEE(e) \text{ and}$
  $EMPLOYEE(s) \text{ and } e.SUPERSSN = s.SSN\}$
- Find the name of each employee who works on some project controlled by department number 5.
  $\{e.LNAME, e.FNAME \mid EMPLOYEE(e) \text{ and } ((\exists x)(\exists w)$
  $(PROJECT(x) \text{ and } WORKS\_ON(w) \text{ and } x.DNUM = 5 \text{ and } w.ESSN = e.SSN \text{ and}$
  $x.PNUMBER = w.PNO))\}$

9.3 THE DOMAIN RELATIONAL CALCULUS

Domain Relational Calculus (DRC) is a non-procedural query language used to express queries in a relational database management system (DBMS). It is a declarative language, meaning users specify what data they want to retrieve without describing the steps for data retrieval. In DRC, queries are constructed using domain variables, which represent individual values in the attributes (or domains) of a relation, rather than entire tuples (rows). This allows DRC to provide a more granular and flexible way to express data constraints and relationships. The query results in DRC are sets of domain values that satisfy specific conditions expressed in logical predicates.

- Rather than having variables range over tuples in relations, the domain variables range over single values from domains of attributes, general form is :

$\{x_1, x_2, \ldots, x_n \mid COND(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m})\}$

- A formula is made up of atoms.

- An atom of the form $R(x_1, x_2, \ldots, x_j)$ (or simply $R(x_1 x_2 \ldots x_j)$), where R is the name of a relation of degree j and each x_i, $1 \leq i \leq j$, is a domain variable.
  This atom defines that $< x_1, x_2, \ldots, x_j >$ must be a tuple in R, where the value of x_i is the value of the i^{th} attribute of the tuple.
  If the domain variables $x_1, x_2, \ldots, x_j$ are assigned values corresponding to a tuple of R, then the atom is TRUE.
- An atom of the form x_i **op** x_j, where **op** is one of the comparison operators $\{=, >, \leq, <, \geq, \neq\}$.
  If the domain variables x_i and x_j are assigned values that satisfy the condition, then the atom is TRUE.
- An atom of the form x_i **op** c or c **op** x_j, where c is a constant value.
  If the domain variables x_i (or x_j) is assigned a value that satisfies the condition, then the atom is TRUE.

DRC makes use of logical operators such as conjunction (∧), disjunction (∨), negation (¬), and quantifiers like existential (∃) and universal (∀) to define constraints on the data. Unlike Tuple Relational Calculus (TRC), which uses tuple variables, DRC operates on domain variables and is more focused on individual attribute values. This characteristic makes DRC suitable for queries that require conditions on specific attributes rather than entire tuples. Although relational calculus in general is not commonly used in practical applications, its theoretical foundation is crucial for understanding the principles of query languages like SQL, which incorporate many of the ideas from relational calculus.

Examples

Consider a database with the following relations:

1. **Employee:**

emp_id	name	salary
E1	John	50000
E2	Alice	60000
E3	Bob	55000

2. **Department:**

dept_id	dept_name
D1	HR
D2	IT
D3	Finance

3. **Works:**

emp_id	dept_id
E1	D1
E1	D2
E2	D2
E3	D3

Query: Find the names of employees who work in the "IT" department.
{ e.name | ∃e_id ∃d_id (Employee(e_id, e.name, e.salary) ∧ Works(e_id, d_id) ∧ d_id = 'D2') }

Explanation:

- e.name: The result we want to retrieve, which is the employee's name.
- ∃e_id ∃d_id: The existential quantifiers specifying that there exist some e_id (employee ID) and d_id (department ID).
- Employee(e_id, e.name, e.salary): A condition that checks if the employee with ID e_id exists in the Employee relation.
- Works(e_id, d_id): A condition that ensures there is a record in the Works relation linking the employee e_id to a department d_id.
- d_id = 'D2': This specifies that the department ID should be 'D2', which corresponds to the "IT" department.

Result:

- The query returns the name of the employee(s) who work in the "IT" department.
- **Output**: Alice

- In this DRC query, we are using domain variables (e_id and d_id) to represent the employee and department.
- The query looks for employees whose e_id matches a record in the Works relation where the department ID (d_id) is 'D2' (IT department).
- The result includes the name of the employee who satisfies this condition.
- This example demonstrates how DRC uses logical expressions with domain variables to filter and retrieve specific data from the relations.

Additional Queries:

Retrieve the birthdate and address of the employee whose name is 'John B Smith'.

$$\{uv \mid (\exists q)(\exists r)(\exists s)(\exists t)(\exists w)(\exists x)(\exists y)(\exists z)$$
$$(EMPLOYEE(qrstuvwxyz) \text{ and } q =' John' \text{ and } r =' B' \text{ and } s =' Smith')\}$$

Retrieve the name and address of all employees who work for the 'Research' department.

$$\{qsv \mid (\exists z)(\exists l)(\exists m)(EMPLOYEE(qrstuvwxyz) \text{ and}$$
$$DEPARTMENT(lmno) \text{ and } l =' Research' \text{ and } m = z)\}$$

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birthdate, and address.

$$\{iksuv \mid (\exists j)(\exists m)(\exists n)(\exists t)(PROJECT(hijk) \text{ and } EMPLOYEE(qrstuvwxyz)$$
$$\text{and } DEPARTMENT(lmno) \text{ and } k = m \text{ and } n = t \text{ and } j =' Stafford')\}$$

Find the names of employees who have no dependents.

$$\{qs \mid (\exists t)(EMPLOYEE(qrstuvwxyz) \text{ and } (not(\exists l)(DEPENDENT(lmnop)$$
$$\text{and } t = l)))\}$$

IS EQUIVALENT TO:

$$\{qs \mid (\exists t)(EMPLOYEE(qrstuvwxyz) \text{ and } ((\forall l)(not(DEPENDENT(lmnop))$$
$$\text{or } not(t = l))))\}$$

List the names of managers who have at least one dependent.

$$\{sq \mid (\exists t)(\exists j)(\exists l)(EMPLOYEE(qrstuvwxyz) \text{ and } DEPARTMENT(hijk)$$
$$\text{and } DEPENDENT(lmnop) \text{ and } t = j \text{ and } l = t)\}$$

➢ **Advantages of Domain Relational Calculus (DRC)**

1. **Declarative Query Language:**
   - Focuses on specifying what data to retrieve rather than how to retrieve it, making it more intuitive and user-friendly for non-technical users.
2. **Expressive Power:**
   - Allows for complex queries involving logical operators (e.g., ∧, ∨, ¬) and quantifiers (∃, ∀), providing significant flexibility in data retrieval.
3. **Foundation for SQL:**
   - Serves as a theoretical basis for SQL, helping in understanding and developing advanced database query languages.
4. **Granularity:**
   - Operates at the domain level, enabling more precise data retrieval by focusing on individual attribute values rather than entire tuples.
5. **Logical Consistency:**
   - Encourages logical and structured query design, ensuring queries are consistent with the database schema and relationships.

➢ **Disadvantages of Domain Relational Calculus (DRC)**

1. **Complexity:**
   - Writing queries can be challenging for beginners due to its reliance on formal logic, making it less accessible to users without a background in mathematical reasoning.
2. **Non-Procedural Nature:**
   - While being declarative is an advantage, the lack of procedural constructs can make it harder to visualize how data will be retrieved.
3. **Performance Issues:**
   - Queries written in DRC may not directly translate into efficient execution plans, potentially leading to performance bottlenecks during data retrieval.
4. **Limited Practical Use:**
   - Unlike SQL, DRC is not widely used in real-world applications, which limits its practical utility and adoption.
5. **Potential for Ambiguity:**
   - Misuse of quantifiers or logical expressions can lead to unintended results, especially in complex queries, making it prone to errors.
6. **Steeper Learning Curve:**
   - Understanding and applying DRC requires familiarity with formal logic and database theory, which can deter its adoption by casual users.

While DRC provides a strong theoretical framework for querying databases, its practical limitations and complexity make it less popular for everyday use compared to more user-friendly query languages like SQL.

9.4 SUMMARY

Relational calculus is a non-procedural query language in database management systems (DBMS) that focuses on defining what data to retrieve rather than detailing how to retrieve it. Rooted in formal logic, it allows users to express queries through logical expressions and constraints, leveraging variables, predicates, and quantifiers. Relational calculus is divided into two main forms: Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC).

TRC uses tuple variables to represent entire rows in a relation, while DRC operates at a more granular level, using domain variables to represent individual attribute values. Both forms utilize existential (∃) and universal (∀) quantifiers, along with logical operators like conjunction (∧), disjunction (∨), and negation (¬), to construct expressive and precise queries. As a theoretical foundation for query languages like SQL, relational calculus bridges the gap between formal database theory and practical database querying. It emphasizes logical consistency and declarative expression, allowing users to focus on specifying their desired outcomes without worrying about execution details. However, the complexity of its formal syntax and reliance on mathematical logic can pose a learning curve, making it more suitable for academic and theoretical purposes than widespread practical use. Nevertheless, its role in shaping the development of modern database query languages highlights its importance in understanding the principles of database management.

9.5 TECHNICAL TERMS

- Non-Procedural Query Language
- Relational Calculus
- Domain Relational Calculus
- Tuple Relational Calculus
- Existential Quantifier
- Universal Quantifier

9.6 SELF-ASSESSMENT QUESTIONS

Short Questions

1. What is the difference between Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC)?
2. Define existential and universal quantifiers in relational calculus.
3. What are free and bound variables in relational calculus?
4. How does relational calculus differ from relational algebra?
5. What is meant by "safety" in relational calculus queries?

Long Questions

1. Explain the concept of relational calculus and its significance in database management systems.
2. Describe Tuple Relational Calculus (TRC) with an example query and explain its components.
3. Compare and contrast Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC), highlighting their differences and use cases.
4. Write a relational calculus query to find employees who work in all departments and explain each part of the query.
5. Discuss how relational calculus serves as a foundation for SQL and other high-level query languages, with examples illustrating its influence.

9.7 SUGGESTED READINGS

1. Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM, 13(6), 377-387.
2. Date, C. J. (2003). "An Introduction to Database Systems." 8th Edition. Addison-Wesley.
3. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). "Database System Concepts." 6th Edition. McGraw-Hill.
4. Ullman, J. D., & Widom, J. (2008). "A First Course in Database Systems." 3rd Edition. Pearson.

Dr. Vasantha Rudramalla

LESSON- 10
SQL-99

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Structure Query Language. The chapter began SQL Data Definitions and Data Types, Specifying Constraints in SQL, Schema Change Statements on SQL, Basic Queries in SQL, More Complex SQL Queries, INSERT, DELETE and UPDATE statements in SQL, Triggers and Views.After completing this chapter, the student will understand Structure Query Language.

10.1 INTRODUCTION

SQL-99, also known as SQL3, is a significant update to the SQL standard that introduced several advanced features for schema definition, constraints, queries, and views. In terms of schema definition, SQL-99 expanded the capabilities for creating and modifying database structures, including more sophisticated data types and table constructs. It introduced comprehensive support for defining constraints, such as primary keys, foreign keys, unique constraints, and check constraints, enhancing data integrity and consistency. SQL-99 also improved query capabilities with new features like common table expressions (CTEs), recursive queries, and enhanced set operations, enabling more complex and efficient data retrieval. Additionally, SQL-99 provided robust support for views, allowing users to create virtual tables that simplify query operations and improve security by restricting direct access to underlying tables. These enhancements made SQL-99 a powerful and flexible standard for managing relational databases.

The chapter first covered began with SQL Data Definitions and Data Types, Specifying Constraints in SQL, Schema Change Statements on SQL, Basic Queries in SQL, More Complex SQL Queries, INSERT, DELETE and UPDATE statements in SQL, Triggers and Views.

10.2 SQL

SQL (Structured Query Language) is the standard programming language used to manage and manipulate relational databases within a Database Management System (DBMS). It allows users to create, read, update, and delete (CRUD) data within the database. SQL is designed to handle structured data and is integral to tasks such as querying databases to retrieve specific information, defining database schema, and controlling access to the data. The language is composed of various commands, including SELECT, INSERT, UPDATE, DELETE, CREATE, and DROP, each serving different functions in database management. Its widespread adoption and robust capabilities make SQL an essential tool for database administrators and developers.

10.2.1 Importance and uses in database management Importance:

1. **Standardization**: SQL is a standardized language, which means that it can be used across different database systems, ensuring compatibility and ease of learning.
2. **Efficiency**: SQL is optimized for managing large volumes of data, allowing for quick retrieval and manipulation.
3. **Ease of Use**: With its relatively simple syntax, SQL is accessible to both technical and non-technical users, making it a versatile tool for various stakeholders.
4. **Integration**: SQL seamlessly integrates with various programming languages and applications, making it a cornerstone of modern data-driven applications.
5. **Data Integrity**: SQL supports constraints and transactions, ensuring data accuracy and consistency.

Uses:

1. **Data Retrieval**: SQL's SELECT statement allows users to query and retrieve specific data from databases based on defined criteria.
2. **Data Manipulation**: Commands such as INSERT, UPDATE, and DELETE enable users to add, modify, and remove data within the database.
3. **Database Creation and Management**: SQL commands like CREATE, ALTER, and DROP allow users to define and modify database schema, including tables, indexes, and views.
4. **Access Control**: SQL provides mechanisms for setting permissions and roles, ensuring that only authorized users can access or modify the data.
5. **Data Analysis**: SQL's powerful querying capabilities support complex data analysis tasks, including aggregation, sorting, and filtering, facilitating data-driven decision-making.
6. **Automation**: SQL can be used in scripts to automate routine database tasks, improving efficiency and reducing the likelihood of human error.
7. **Reporting**: SQL queries can be used to generate detailed reports, extracting meaningful insights from the raw data stored in databases.

10.2.2 Overview of SQL standards

SQL:2011

- **Year**: 2011
- **Overview**: Introduced temporal data support, enabling better handling of time-based data.
- **Key Features**:
- Temporal tables (system-versioned and application-time period tables)
- Enhanced period data types

SQL:2016

- **Year**: 2016
- **Overview**: Focused on big data support, JSON integration, and other modern data handling capabilities.
- **Key Features**:
- JSON data types and functions
- Enhanced polymorphic table functions
- Row pattern recognition in result sets

SQL:2019

- **Year**: 2019
- **Overview**: The most recent standard, incorporating incremental improvements and refinements.
- **Key Features**:
- Enhanced support for JSON
- Improvements in window functions
- Expanded capabilities for polymorphic table functions

The evolution of SQL standards reflects the changing needs and advancements in database technology. Each iteration builds upon the previous ones, ensuring that SQL remains a powerful and versatile language for managing relational databases. The adherence to these standards by database vendors ensures compatibility and interoperability across different systems, providing a consistent experience for users.

10.3 SQL DATA DEFINITIONS AND DATA TYPES

SQL Data Definition Language (DDL) encompasses commands that define and manage the database schema. DDL commands like CREARE, ALTER, DROP and etc. ensure that the database structure is defined, organized, and maintained efficiently, setting the foundation for data storage and manipulation. SQL data types specify the kind of data that can be stored in a table's columns. Common SQL data types include: numeric, character, binary and etc. These data types ensure that the data is stored in a consistent, efficient, and appropriate format, facilitating accurate data processing and retrieval.

10.3.1 Data Definition Language (DDL)

Data Definition Language (DDL) is a subset of SQL used to define and manage the structure of a database. DDL commands are responsible for creating, modifying, and deleting database objects such as tables, indexes, views, and schemas.

Here are the key DDL commands:

❖ **CREATE**

- **Purpose**: To create new database objects.
- **Examples**:
- **Creating a Table**
- CREATE TABLE employees (

employee_id INT PRIMARY KEY,
first_name VARCHAR(50),
last_name VARCHAR(50), hire_date DATE);

```
CREATE TABLE EMPLOYEE
        ( Fname          VARCHAR(15)          NOT NULL,
          Minit          CHAR,
          Lname          VARCHAR(15)          NOT NULL,
          Ssn            CHAR(9)              NOT NULL,
          Bdate          DATE,
          Address        VARCHAR(30),
          Sex            CHAR,
          Salary         DECIMAL(10,2),
          Super_ssn      CHAR(9),
          Dno            INT                  NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE DEPARTMENT
        ( Dname          VARCHAR(15)          NOT NULL,
          Dnumber        INT                  NOT NULL,
          Mgr_ssn        CHAR(9)              NOT NULL,
          Mgr_start_date DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
```

Fig 10.1 Table creation for COMPANY DATABASE using CREATE Command

❖ **ALTER**

- **Purpose**: To modify existing database objects.
- **Examples**:
- **Adding a Column to a Table**
- ALTER TABLE employees ADD COLUMN email VARCHAR(100);

❖ **DROP**

- **Purpose**: To delete database objects.
- **Examples**:
- **Dropping a Table**:
- DROP TABLE employees;

❖ **TRUNCATE**

- **Purpose**: To remove all rows from a table, quickly and efficiently.
- **Example**
- TRUNCATE TABLE employees;

DDL commands provide the necessary tools to define, manage, and maintain the database schema, ensuring that the database structure aligns with the needs of the application and supports efficient data storage and retrieval.

10.3.2 Data Types

SQL data types specify the kind of data that can be stored in a table's columns. They ensure that data is stored in a consistent and efficient manner, facilitating accurate data processing and retrieval. Here are the primary SQL data types:

❖ **Numeric Data Types**

1. **INTEGER**:
   - Stores whole numbers.
   - Example: INTEGER, INT
   - Usage: employee_id INT

2. **SMALLINT**:
   - Stores smaller range of whole numbers.
   - Usage: age SMALLINT

3. **BIGINT**:
   - Stores larger range of whole numbers.
   - Usage: population BIGINT

4. **DECIMAL(p, s)** or **NUMERIC(p, s)**:
   - Stores fixed-point numbers with precision p and scale s.
   - Usage: salary DECIMAL(10, 2)

5. **FLOAT**:
   - Stores floating-point numbers.
   - Usage: temperature FLOAT

6. **REAL** and **DOUBLE PRECISION**:
   - Stores approximate numeric values.
   - Usage: measurement DOUBLE PRECISION

❖ **Character Data Types**

1. **CHAR(n)**:
   - Stores fixed-length character strings.
   - Usage: gender CHAR(1)

2. **VARCHAR(n)**:
   - Stores variable-length character strings.
   - Usage: name VARCHAR(50)

3. **TEXT**:
   - Stores large variable-length character strings.
   - Usage: description TEXT

❖ **Date and Time Data Types**

1. **DATE**:
   - Stores dates (year, month, day).
   - Usage: birthdate DATE

2. **TIME**:
   - Stores time of day (hours, minutes, seconds).
   - Usage: appointment_time TIME

3. **TIMESTAMP**:
   - Stores date and time.
   - Usage: order_timestamp TIMESTAMP

4. **INTERVAL**:
   - Stores a time interval.
   - Usage: duration INTERVAL

❖ **Binary Data Types**

1. **BINARY**:
   - Stores fixed-length binary data.
   - Usage: binary_data BINARY(16)

2. **VARBINARY**:
   - Stores variable-length binary data.
   - Usage: image VARBINARY(255)

3. **BLOB**:
   - Stores large binary objects.
   - Usage: document BLOB

❖ **Boolean Data Type**

1. **BOOLEAN**:
   - Stores true or false values.
   - Usage: is_active BOOLEAN

❖ **Other Data Types**

1. **ENUM**:
   - Stores one value from a predefined list of values (MySQL specific).
   - Usage: status ENUM('active', 'inactive', 'pending')

2. **SET**:
   - Stores a set of values (MySQL specific).
   - Usage: roles SET('admin', 'user', 'guest')

3. **JSON**:
   - Stores JSON-formatted data.
   - Usage: preferences JSON

Examples:

```
CREATE TABLE employees (
   employee_id INT PRIMARY KEY,
   first_name VARCHAR(50),
   last_name VARCHAR(50),
   hire_date DATE,
   salary DECIMAL(10, 2),
   is_active BOOLEAN);

CREATE TABLE files (
   file_id INT PRIMARY KEY,
   file_name VARCHAR(255),
   file_data BLOB);
```

These data types help define the kind of data each column can hold, ensuring data integrity and optimizing storage.

10.4 SPECIFYING CONSTRAINTS IN SQL

Specifying constraints in SQL is essential for enforcing rules and maintaining data integrity within a database. Constraints ensure that the data adheres to defined standards and prevents invalid data entry.

10.4.1 Types of Constraints

Key types of constraints include:

- **PRIMARY KEY**: Ensures that each value in a column (or a combination of columns) is unique and not null, uniquely identifying each row in a table.
- **FOREIGN KEY**: Establishes a relationship between columns in different tables, ensuring referential integrity by linking a column (or columns) to the primary key of another table.
- **UNIQUE**: Ensures that all values in a column (or a combination of columns) are unique across the entire table.
- **NOT NULL**: Ensures that a column cannot have a null value, requiring that every row must have a value for this column.
- **CHECK**: Enforces a condition that each row must satisfy, restricting the values that can be stored in a column.

10.4.2 Examples of Constraints in Table Definitions

```
CREATE TABLE employees (
   employee_id INT PRIMARY KEY,
   first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,
   email VARCHAR(100) UNIQUE,
   hire_date DATE CHECK (hire_date >= '2000-01-01'));

CREATE TABLE orders (
   order_id INT PRIMARY KEY,
   employee_id INT,
   order_date DATE NOT NULL,
   FOREIGN KEY (employee_id) REFERENCES employees(employee_id));
```

Constraints help maintain the accuracy, reliability, and integrity of the data within the database, ensuring that the database adheres to the specified business rules and logic.

10.5 SCHEMA CHANGE STATEMENTS IN SQL

Schema change statements in SQL are used to modify the structure of an existing database schema, allowing for the addition, alteration, or deletion of database objects such as tables, columns, indexes, and constraints. These changes are essential for adapting the database to evolving requirements.

10.5.1 Altering Schemas

Altering schemas in SQL involves modifying the structure of an existing database schema to accommodate changing requirements or to optimize performance. The primary SQL command used for altering schemas is the ALTER statement. This command allows users to add, modify, or delete database objects such as tables, columns, indexes, and constraints.
Here are the key operations that can be performed with the ALTER statement:

```
-- Add a new column
ALTER TABLE employees ADD COLUMN birth_date DATE;

-- Modify an existing column's data type
ALTER TABLE employees ALTER COLUMN salary DECIMAL(10, 2);

-- Drop a column
ALTER TABLE employees DROP COLUMN temp_data;

-- Add a new primary key constraint
ALTER TABLE employees ADD CONSTRAINT pk_employee_id PRIMARY KEY
(employee_id);

-- Drop a unique constraint
ALTER TABLE employees DROP CONSTRAINT email_unique;
```

```
-- Rename a column
ALTER TABLE employees RENAME COLUMN old_column_name TO new_column_name;

-- Rename a table
ALTER TABLE employees RENAME TO staff_members;
```

Altering schemas is a fundamental aspect of database management, allowing administrators and developers to keep the database structure aligned with application requirements and data integrity rules.

10.5.2 Managing Indexes

Managing indexes in SQL involves creating, modifying, and deleting indexes to optimize query performance and maintain database efficiency. Indexes are special data structures that improve the speed of data retrieval operations on a database table.
Here are the key operations for managing indexes:

```
-- Create a single-column index

CREATE INDEX idx_lastname ON employees(last_name);

-- Create a composite index on first_name and birth_date
CREATE INDEX idx_name_dob ON employees(first_name, birth_date);

-- Create a unique index on email
CREATE UNIQUE INDEX idx_unique_email ON employees(email);

-- Drop an index
DROP INDEX idx_lastname;

-- MySQL-specific syntax to drop an index
ALTER TABLE employees DROP INDEX idx_lastname;
```

By effectively managing indexes, database administrators can significantly enhance query performance, ensuring efficient and fast data retrieval operations.

10.6 BASIC QUERIES IN SQL

Basic queries in SQL involve selecting, filtering, and retrieving data from one or more tables in a database.

10.6.1 SELECT Statement

The SELECT statement in SQL is used to retrieve data from a database. It allows you to specify the columns you want to retrieve and the table from which to retrieve them. The SELECT statement can include various clauses to filter, sort, and group the data.

Basic Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
ORDER BY column1, column2, ...;
```

Example

```
SELECT first_name, last_name, department
FROM employees
WHERE hire_date > '2020-01-01'
ORDER BY last_name ASC;
```

This query will return a list of employees' first names, last names, and departments, filtered and sorted according to the specified criteria.

10.6.2 INSERT statement

The INSERT statement in SQL is used to add new rows of data into a table. You can insert values into all columns of a table or specify which columns to insert data into.

Basic Syntax

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

Example:

```
INSERT INTO employees
VALUES (101, 'John', 'Doe', 'Sales', '2024-07-22');
```

10.7 MORE COMPLEX SQL QUERIES

More complex SQL queries often involve multiple tables, advanced filtering, subqueries, aggregation, and conditional logic to retrieve, manipulate, and analyze data in sophisticated ways. These queries can use various SQL clauses and functions, including JOIN operations, GROUP BY, HAVING, subqueries, CASE statements, and window functions. Complex queries are essential for in-depth data analysis, reporting, and ensuring that intricate business logic is accurately reflected in the data retrieved.

10.7.1 JOIN Operations

JOIN operations in SQL are used to combine rows from two or more tables based on a related column between them. The most common types of JOINs are INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN.

INNER JOIN: Returns rows that have matching values in both tables.

```
SELECT a.column1, b.column2
FROM table1 a
INNER JOIN table2 b ON a.common_column = b.common_column;
```

LEFT JOIN (LEFT OUTER JOIN): Returns all rows from the left table and matched rows from the right table. Unmatched rows in the right table will have NULL values.

```
SELECT a.column1, b.column2
FROM table1 a
LEFT JOIN table2 b ON a.common_column = b.common_column;
```

RIGHT JOIN (RIGHT OUTER JOIN): Returns all rows from the right table and matched rows from the left table. Unmatched rows in the left table will have NULL values.

```
SELECT a.column1, b.column2
FROM table1 a
RIGHT JOIN table2 b ON a.common_column = b.common_column;
```

FULL OUTER JOIN: Returns rows when there is a match in one of the tables. Rows with no match in either table will have NULL values.

```
SELECT a.column1, b.column2
FROM table1 a
FULL OUTER JOIN table2 b ON a.common_column = b.common_column;
```

10.7.2 Subqueries

Subqueries in SQL are queries nested within another SQL query. They allow you to perform operations that would otherwise be impossible or cumbersome with a single query. Subqueries can be used in various clauses, such as SELECT, FROM, WHERE, and HAVING, to provide intermediate results for the main query. They enhance the flexibility and power of SQL by enabling more complex queries and data manipulations.

Types of Subqueries

1. **Scalar Subquery**: Returns a single value.
2. **Row Subquery**: Returns a single row with multiple columns.
3. **Table Subquery**: Returns a set of rows and columns.

Example 1: Subquery in a SELECT Clause

To get the names of employees and their respective department names from employees and departments tables:

```
SELECT e.first_name, e.last_name,
    (SELECT d.department_name
     FROM departments d
     WHERE d.department_id = e.department_id) AS department_name
FROM employees e;
```

Example 2: Subquery in a FROM Clause

To get the department-wise average salary:

```
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id
HAVING AVG(salary) > (SELECT AVG(salary) FROM employees);
```

Subqueries can greatly enhance the capability of SQL queries by allowing more detailed and specific data retrieval, making them essential for complex data analysis and reporting tasks.

10.7.3 Set Operations

Set operations in SQL are used to combine the results of two or more SELECT queries. These operations include UNION, UNION ALL, INTERSECT, and EXCEPT (or MINUS in some databases). Set operations enable you to perform mathematical set operations on query results, allowing for powerful and flexible data manipulation.

10.8 INSERT, DELETE, AND UPDATE STATEMENTS IN SQL

The INSERT, DELETE, and UPDATE statements in SQL are essential for managing and manipulating data within a database. These Data Manipulation Language (DML) statements allow users to add new records, remove existing ones, and modify existing data, respectively.

10.8.1 INSERT Statement

The INSERT statement is used to add new rows to a table. You can insert values into all columns or specify which columns to insert data into.

Example

```
INSERT INTO employees (first_name, last_name, department)
VALUES ('Jane', 'Doe', 'Marketing');
```

10.8.2 DELETE Statement

The DELETE statement is used to remove existing rows from a table based on a specified condition. Without a condition, it will delete all rows in the table.

```
DELETE FROM employees
WHERE employee_id = 101;
```

10.8.3 UPDATE Statement

The UPDATE statement is used to modify existing data in a table. It allows you to set new values for one or more columns based on a specified condition.

```
UPDATE employees
SET department = 'Sales'
WHERE employee_id = 101;
```

10.9 TRIGGERS IN SQL

Triggers in SQL are special types of stored procedures that automatically execute or "fire" when specific database events occur, such as INSERT, UPDATE, or DELETE operations on a table. Triggers are used to enforce business rules, maintain data integrity, audit changes, and synchronize tables. They can be set to execute before or after the event, allowing for pre-processing or post-processing of data. For example, a trigger can be created to automatically log changes to an audit table whenever an employee's salary is updated.

10.9.1 Types of Triggers

In SQL, triggers can be categorized based on the timing of their execution and the events that activate them.

The main types of triggers are:

Based on Timing:

- **BEFORE Triggers**: Execute before the triggering event (INSERT, UPDATE, DELETE) occurs. These are typically used for validation or modification of data before it is committed to the database.
- **Example**: BEFORE INSERT, BEFORE UPDATE, BEFORE DELETE
- **AFTER Triggers**: Execute after the triggering event has occurred. These are often used for logging changes, enforcing referential integrity, or synchronizing data across tables.
- **Example**: AFTER INSERT, AFTER UPDATE, AFTER DELETE

Based on Event:

Triggers based on events in SQL are designed to automatically execute a specified action when certain events—such as INSERT, UPDATE, or DELETE operations—occur on a table. These event-driven triggers help maintain data integrity, enforce business rules, and automate system tasks.

Each type of event trigger serves a specific purpose:

- **INSERT Triggers**: Execute when a new record is added to a table. They can be used to set default values, validate data, or log insert actions.

```
CREATE TRIGGER before_insert_employee
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
   SET NEW.hire_date = NOW();
END;
```

UPDATE Triggers: Execute when an existing record is modified. They are useful for tracking changes, maintaining history logs, or enforcing complex validation rules.

```
CREATE TRIGGER after_update_employee
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
   INSERT INTO employees_audit (employee_id, old_salary, new_salary, change_date)
   VALUES (OLD.employee_id, OLD.salary, NEW.salary, NOW());
END;
```

- **DELETE Triggers**: Execute when a record is removed from a table. They can be employed to prevent accidental deletions, cascade deletions to related tables, or archive deleted data.

```
CREATE TRIGGER before_delete_employee
BEFORE DELETE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO employees_deleted (employee_id, first_name, last_name, department, deletion_date)
  VALUES (OLD.employee_id, OLD.first_name, OLD.last_name, OLD.department, NOW());
END;
```

Triggers enhance the robustness and reliability of database applications by providing automated responses to data changes.

10.9.2 Creating and Dropping Triggers

Creating triggers in SQL involves defining the specific event (INSERT, UPDATE, DELETE) that activates the trigger and the action that should be performed when the trigger fires. Triggers can be set to execute either before or after the specified event.
Dropping a trigger involves removing it from the database, which means it will no longer execute when the specified event occurs.

Example : To drop the previously created log_employee_deletion trigger:

```
DROP TRIGGER log_employee_deletion;
```

SUMMARY

- **Creating Triggers**: Use the CREATE TRIGGER statement to define when the trigger should fire (BEFORE or AFTER an event) and what actions it should perform.
- **Dropping Triggers**: Use the DROP TRIGGER statement to remove an existing trigger, preventing it from executing in response to its associated event.

10.9.3 Use Cases for Triggers

1. **Data Validation and Integrity**
   - **Ensure Data Consistency**: Automatically enforce complex constraints and validation rules that standard constraints cannot handle.
   - **Example**: Prevent an employee's hire date from being earlier than their birth date.

2. **Auditing and Logging**
   - **Track Changes**: Automatically log changes to critical data for audit trails, compliance, and monitoring purposes.
   - **Example**: Log every update to an employee's salary in an audit table.

3. **Enforcing Business Rules**
   - **Implement Business Logic**: Ensure consistent application of business policies by automatically executing specific actions when certain conditions are met.
   - **Example**: Prevent the deletion of a customer record if the customer has pending orders.

4. **Synchronizing Tables**
   - **Maintain Data Synchronization**: Automatically update or synchronize related tables to ensure data consistency across the database.
   - **Example**: Update the inventory stock count whenever an order is placed.

5. **Cascading Actions**
   - **Automate Related Operations**: Perform additional related actions automatically when a certain event occurs, such as cascading deletions or updates.
   - **Example**: Automatically delete all orders related to a customer when the customer record is deleted.

These use cases illustrate the powerful capabilities of triggers in automating and enforcing data management tasks, enhancing data integrity, and ensuring adherence to business rules.

10.10 VIEWS IN SQL

Views in SQL are virtual tables that provide a way to present and query data from one or more tables. They do not store data themselves but instead store a predefined SQL query that dynamically retrieves data from the underlying tables when accessed. Views can simplify complex queries, enhance security by restricting access to specific data, and provide a consistent, abstracted interface to the data. Users can perform SELECT operations on views as if they were actual tables, and in some cases, views can also support INSERT, UPDATE, and DELETE operations, depending on the database system and view definition.

10.10.1 Creating and Managing Views

Views in SQL are virtual tables that represent the result of a stored query. They simplify complex queries, enhance security, and provide a level of abstraction from the underlying table structures. Here's how to create and manage views:

❖ **Creating Views**

Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example

To create a view that shows the full names and departments of employees:

```
CREATE VIEW employee_overview AS
SELECT first_name || ' ' || last_name AS full_name, department
FROM employees;
```

❖ **Using Views**

Once a view is created, you can query it just like a regular table:

```
SELECT * FROM employee_overview;
```

❖ **Modifying Views**

To modify an existing view, you use the CREATE OR REPLACE VIEW statement:

```
CREATE OR REPLACE VIEW employee_overview AS
SELECT first_name, last_name, department, hire_date
FROM employees
WHERE hire_date > '2020-01-01';
```

❖ **Dropping Views**

To remove an existing view, you use the DROP VIEW statement:

```
DROP VIEW employee_overview;
```

Benefits of Using Views

- **Simplify Complex Queries**: Encapsulate complex SQL logic within a view for easier reuse.
- **Enhance Security**: Restrict user access to specific data by granting permissions on views rather than on the underlying tables.
- **Data Abstraction**: Provide a consistent interface to data, even if the underlying schema changes.
-

Views are powerful tools for managing and abstracting data in SQL databases. By creating, modifying, and dropping views, you can simplify query operations, enhance security, and maintain consistent data access interfaces.

10.10.2 Materialized Views

Materialized views are a type of database object that store the result of a query physically, unlike regular views that store only the query itself and generate results dynamically each time they are accessed. Materialized views improve query performance, especially for complex and resource-intensive queries, by precomputing and storing the query results. They are periodically refreshed to stay up-to-date with the underlying data.

❖ **Creating Materialized Views**

Example

To create a materialized view that stores the total sales per department:

```
CREATE MATERIALIZED VIEW total_sales_per_department AS
SELECT department_id, SUM(sale_amount) AS total_sales
FROM sales
GROUP BY department_id;
```

❖ **Refreshing Materialized Views**

Materialized views need to be refreshed to reflect changes in the underlying data. This can be done manually or automatically at specified intervals.

Manual Refresh:

```
    REFRESH MATERIALIZED VIEW view_name;
Automatic Refresh (depends on the database system):
CREATE MATERIALIZED VIEW view_name
REFRESH FAST EVERY 1 HOUR
AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Benefits of Materialized Views

- **Improved Performance**: Speeds up query performance by avoiding repeated execution of complex queries.
- **Data Pre-aggregation**: Useful for pre-aggregating data, which can be directly queried for fast results.
- **Reduced Load**: Decreases the load on the underlying tables during heavy read operations.

Materialized views enhance query performance by storing precomputed results of complex queries. They are particularly beneficial for scenarios requiring frequent access to aggregated data, providing a significant performance boost while reducing the computational load on the database. Regular refreshes ensure the materialized view data remains current with the underlying tables.

10.11 SUMMARY

The chapter covers essential aspects of SQL, beginning with **SQL Data Definitions and Data Types**, which define the structure and nature of data in a database. It explains how to create tables and specify various data types like INTEGER, VARCHAR, and DATE. **Specifying Constraints in SQL** ensures data integrity through primary keys, foreign keys, unique constraints, and check constraints. **Schema Change Statements** such as ALTER TABLE, RENAME, and DROP allow modifications to the database schema. **Basic Queries in SQL** use the SELECT statement to retrieve data, while **More Complex SQL Queries** involve advanced filtering, joins, and subqueries for intricate data retrieval. The **INSERT, DELETE, and UPDATE statements** are fundamental for managing data within tables. **Triggers** are automated responses to specific events, enforcing business rules and maintaining data consistency. Finally, **Views** and **Materialized Views** provide virtual tables for simplified data access and improved query performance, respectively. This comprehensive overview equips readers with the foundational tools for effective database management and manipulation.

10.12 TECHNICAL TERMS

SQL, Shema, Data Types, Select, Insert, View, Materialized view, Trigger

10.13 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Illustrate about Views in SQL
2. Describe about Triggers in SQL
3. Explain about basic SQL operations

Short Notes:

1. Write about insert statement
2. Define Materalized View
3. Explain about how to update view.

10.14 SUGGESTED READINGS

1. Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM, 13(6), 377-387.
2. Date, C. J. (2003). "An Introduction to Database Systems." 8th Edition. Addison-Wesley.
3. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). "Database System Concepts." 6th Edition. McGraw-Hill.
4. Ullman, J. D., & Widom, J. (2008). "A First Course in Database Systems." 3rd Edition. Pearson.

Dr. Vasantha Rudramalla

LESSON- 11

FUNCTIONAL DEPENDENCIES

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Functional Dependencies for Relational Databases. The chapter began Informal Design Guidelines for Relation Schemas, Functional dependencies. After completing this chapter, the student will understand Functional Dependencies for Relational Databases.

11.1 INTRODUCTION

The design of a relational database schema plays a crucial role in ensuring data consistency, integrity, and efficient query processing. A poorly designed schema can lead to problems such as data redundancy, update anomalies, insertion and deletion anomalies, and inconsistent data. To avoid these issues, database designers follow a set of informal design guidelines and apply formal methods like normalization and functional dependency analysis. These approaches help to organize data logically and eliminate unnecessary duplication while maintaining all necessary relationships among attributes.

This chapter focuses on the principles and tools that guide relational schema design. It begins with informal design guidelines that address issues such as attribute semantics, redundancy, null values, and spurious tuples. Then, it introduces the concept of functional dependencies (FDs) — a key theoretical foundation in database normalization. FDs describe the relationship between attributes in a relation and are used to detect design flaws and refine schemas through decomposition. The chapter also explains Armstrong's Axioms, closure of functional dependencies, and decomposition techniques, which together form the backbone of logical database design and normalization up to advanced normal forms.

11.2 INFORMAL DESIGN GUIDELINES FOR RELATION SCHEMAS

11.2.1 Semantics of the Relation Attributes Each relation schema should represent a single entity or concept, with attributes clearly defined to describe the entity's properties. This ensures that the data stored within the relation is meaningful and accurately reflects the real-world scenario it models.

To maintain clear semantics, the designer must ensure that:

1. Each attribute directly corresponds to a property of the entity the relation represents.
2. The naming of attributes is consistent and self-descriptive.
3. The relation should avoid mixing entities or concepts that belong to different real-world objects.

Example 1: Correct Semantics

A properly designed relation representing the **STUDENT** entity:

STUDENT(Student_ID, Student_Name, Date_of_Birth, Department, Email)

- Here, each attribute — Student_ID, Student_Name, Date_of_Birth, Department, and Email — directly describes a **property of a student**.
- The meaning is clear, consistent, and corresponds to a single real-world concept — a *student*.

Example 2: Poor Semantics

An improperly designed relation mixing two different entities:

STUDENT_COURSE(Student_ID, Student_Name, Course_Name, Instructor_Name, Instructor_Phone)

- This table combines attributes of **students**, **courses**, and **instructors** into a single relation.
- Attributes like Instructor_Name and Instructor_Phone describe a *teacher*, not a *student* — violating the rule of single-entity representation.

Such a schema leads to **redundancy** (e.g., repeating instructor details for every student enrolled in the course) and **update anomalies** (if an instructor's phone number changes, multiple records must be updated).

Example 3: Clear Conceptual Separation

To correct the above design, we separate the entities:

STUDENT(Student_ID, Student_Name, Department)

COURSE(Course_ID, Course_Name)

INSTRUCTOR(Instructor_ID, Instructor_Name, Instructor_Phone)

ENROLLMENT(Student_ID, Course_ID, Instructor_ID)

- Each table now represents a **distinct concept**.
- Relationships among students, courses, and instructors are maintained through **foreign keys** in the ENROLLMENT table.
- This preserves semantic clarity, reduces redundancy, and ensures a more flexible and consistent database structure.

Maintaining **semantic clarity** in relation schemas ensures that the database accurately models real-world entities and relationships. Each relation must represent a single, well-defined concept, and its attributes should be directly related to that concept — forming the foundation for reliable and meaningful data management.

11.2.2 Redundant Information in Tuples and Update Anomalies Avoiding redundancy is crucial as it can lead to update anomalies such as insertion, deletion, and modification anomalies. Redundant data requires multiple updates for a single logical change, increasing the risk of inconsistencies.

Redundancy in a relation occurs when the same piece of information is stored multiple times within the database. Although redundancy may sometimes seem harmless, it often leads to serious problems such as **increased storage usage**, **data inconsistency**, and various **update anomalies** during insert, update, or delete operations. In a well-designed database, each fact should be stored **only once**, ensuring that a single update reflects everywhere it is needed.

When redundant information exists, a single logical change in the real world requires multiple updates in the database. If any of these updates are missed, the database becomes **inconsistent** — some tuples may show the old value, while others show the new one. This undermines the reliability and integrity of the data.

Example of Redundancy

Consider the relation:

EMPLOYEE_DEPT(Emp_ID, Emp_Name, Dept_Name, Dept_Location)

Emp_ID	Emp_Name	Dept_Name	Dept_Location
101	Ramesh	IT	Hyderabad
102	Suresh	IT	Hyderabad
103	Meena	HR	Chennai

Here, the **department location** is repeated for every employee in the same department. This duplication leads to **redundant information**, making the system prone to anomalies.

Types of Update Anomalies

1. **Insertion Anomaly**

   Occurs when new information cannot be inserted into the database because other required data is missing.

   - Example: If a new department "Finance" is created but no employee is assigned yet, we **cannot insert** the department information into the above relation without leaving employee-related fields blank.

   Dept_Name = Finance → Cannot insert without Emp_ID or Emp_Name

2. **Update Anomaly**

   Occurs when redundant data must be updated in multiple places, and one or more updates are missed, leading to inconsistencies.

   - Example: If the IT department relocates from Hyderabad to Bangalore, all tuples for employees in IT must be updated. If one record is missed, inconsistent department locations appear.

   Some IT employees show Hyderabad, others show Bangalore → inconsistency.

3. **Deletion Anomaly**

   Occurs when deleting a record inadvertently removes useful information that should have been retained.

- Example: If the only employee in the HR department (Meena) leaves and her record is deleted, **the information about the HR department and its location (Chennai)** is lost.

Deleting Meena → also deletes HR department info.

Corrective Measure

To eliminate redundancy and prevent these anomalies, we can **normalize** the relation by splitting it into smaller, related tables:

EMPLOYEE(Emp_ID, Emp_Name, Dept_Name)

DEPARTMENT(Dept_Name, Dept_Location)

Now, department details are stored **once** in the DEPARTMENT table, and employee information references it via **Dept_Name**. This ensures consistency, easy updates, and no accidental data loss.

11.2.3 Null Values in Tuples Minimize the use of null values as they can complicate queries and interpretations. Null values often indicate missing or inapplicable information, which can lead to ambiguous results and complex query conditions.

The presence of null values in database tuples generally indicates that certain information is missing, unknown, or not applicable. While nulls are sometimes unavoidable, excessive or improper use of them can lead to ambiguity, inconsistent interpretations, and complex query processing. Therefore, good database design practices aim to minimize the use of null values by refining the schema and ensuring that attributes are appropriately defined.

A null value does not mean zero or an empty string—it represents the absence of a known value. However, nulls create challenges for query formulation and evaluation because they behave differently in logical operations. For example, any comparison involving a null (such as NULL = 5 or NULL != 5) evaluates to unknown rather than true or false, making query results less predictable.

Example 1: Problem with Null Values
Consider the relation:

EMPLOYEE(Emp_ID, Emp_Name, Phone, Manager_ID)

Emp_ID	Emp_Name	Phone	Manager_ID
101	Ravi	9876543210	201
102	Meena	NULL	201
103	Arjun	8765432109	NULL

- **Employee 102** has a null in Phone, meaning the number is missing or not available.
- **Employee 103** has a null in Manager_ID, meaning this employee may not be assigned a manager yet (e.g., a department head).

When performing queries like:

SELECT * FROM EMPLOYEE WHERE Manager_ID = 201;

The tuples with NULL in Manager_ID are **ignored**, as NULL values do not satisfy equality conditions.

Example 2: Complex Query Conditions

To include tuples with missing information, queries must use conditional logic:

SELECT * FROM EMPLOYEE WHERE Manager_ID = 201 OR Manager_ID IS NULL;

This makes queries longer, harder to maintain, and more error-prone. If nulls are overused, the interpretation of data becomes uncertain—does NULL mean "unknown," "not applicable," or "not yet assigned"?

Ways to Minimize Null Values

1. **Refine the Schema** – Split relations so that optional attributes are moved into separate tables.

   Example:
   Instead of storing all employee data in one table, use:
   EMPLOYEE(Emp_ID, Emp_Name)
   CONTACT(Emp_ID, Phone)
   Now, only employees with phone numbers appear in the CONTACT table.

2. **Use Default Values Where Appropriate** – Replace nulls with meaningful defaults, such as 'N/A' or 0, if it does not distort semantics.
3. **Apply Constraints** – Use NOT NULL constraints in SQL for essential attributes that must always contain data.

Null values are useful for representing **unknown or inapplicable information**, but their excessive use reduces data clarity and complicates operations. By minimizing nulls through schema design, default values, and proper constraints, database systems maintain **consistency**, **simplicity**, and **accurate query results**.

11.2.4 Spurious Tuples Preventing spurious tuples, which are erroneous data combinations resulting from improper joins, is essential. Proper decomposition and careful schema design help avoid spurious tuples, ensuring that join operations yield meaningful and accurate results.

Spurious tuples are erroneous or meaningless data combinations that appear in the result of a join operation when relations are improperly decomposed or incorrectly joined. They occur when two relations are joined on attributes that do not represent a valid or complete key relationship. The presence of spurious tuples leads to incorrect query results, data inconsistency, and loss of data integrity.

To ensure accurate data retrieval, it is essential that every decomposition of a relation maintains the lossless join property—that is, when the decomposed relations are joined back together, they should reconstruct the original data without creating any extra (spurious) tuples.

Example: Improper Decomposition Leading to Spurious Tuples

Consider the relation:

EMP_PROJECT(Emp_ID, Emp_Name, Project_ID, Project_Name)

If this relation is decomposed incorrectly into:

R1(Emp_ID, Emp_Name)

R2(Emp_ID, Project_ID, Project_Name)

and then joined using a **non-key attribute** such as Emp_Name instead of the primary key Emp_ID, spurious tuples may appear.

Data in the original relation:

Emp_ID	Emp_Name	Project_ID	Project_Name
101	Ravi	P1	AI System
102	Meena	P2	Web Portal

Decomposed Relations:

R1(Emp_ID, Emp_Name)

Emp_ID	Emp_Name
101	Ravi
102	Meena

R2(Emp_ID, Project_ID, Project_Name)

Emp_ID	Project_ID	Project_Name
101	P1	AI System
102	P2	Web Portal

If we mistakenly join them using Emp_Name instead of Emp_ID:

```
SELECT *

FROM R1, R2

WHERE R1.Emp_Name = R2.Project_Name;
```

This join could produce **spurious tuples**, creating invalid combinations of employees and projects that never existed in the original data.

How to Prevent Spurious Tuples

1. **Use Lossless Decomposition:**
   Decompose relations based on **functional dependencies** and **primary keys** to ensure that joins reconstruct the original relation correctly.
2. **Join on Proper Keys:**
   Always perform joins on **primary key–foreign key relationships** rather than on arbitrary or non-key attributes.
3. **Verify with the Lossless Join Test:**

   Before finalizing decomposition, check whether joining decomposed relations preserves all original tuples without adding spurious ones.
4. **Maintain Referential Integrity:**

   Define proper constraints in the DBMS to ensure that key relationships are respected.

Example of Lossless Decomposition

If we correctly decompose the same relation as:

EMPLOYEE(Emp_ID, Emp_Name)

PROJECT(Project_ID, Project_Name)

ASSIGNMENT(Emp_ID, Project_ID)

Joining these tables on their key relationships (Emp_ID and Project_ID) reproduces the original data **without spurious tuples**, ensuring **lossless and meaningful joins**.

Preventing **spurious tuples** is fundamental to reliable database design. By ensuring **lossless decomposition**, using **correct join keys**, and maintaining **referential integrity**, designers avoid erroneous data combinations. Together with other design principles—semantic clarity, reduced redundancy, and minimal nulls—this practice ensures that the resulting database schema is **accurate, efficient, and anomaly-free**.

11.3 FUNCTIONAL DEPENDENCIES

11.3.1 Definition and Concept

A functional dependency (FD) is a constraint between two sets of attributes in a relation. For a relation R, an attribute Y is functionally dependent on attribute X (denoted as X → Y) if for every valid instance of X, that value of X uniquely determines the value of Y.

11.3.2 Trivial and Non-trivial Functional Dependencies

A functional dependency (FD) describes a relationship between two sets of attributes in a relation schema. It is expressed as:

$$X \rightarrow Y$$

which means the value of attribute set X uniquely determines the value of attribute set Y.
Functional dependencies are categorized into trivial and non-trivial, depending on whether the dependent attributes are already part of the determinant attributes.

1. Trivial Functional Dependency

A functional dependency X → Y is said to be trivial if Y is a subset of X. In other words, the dependency already exists by definition and adds no new information.

If $Y \subseteq X$, then $X \rightarrow Y$ is trivial.

Examples:

- {Student_ID, Name} → Student_ID (trivial, since Student_ID ⊆ {Student_ID, Name})
- {Emp_ID} → Emp_ID (trivial)
- {Course_ID, Title} → Title (trivial)

The dependency holds for all possible data in a relation — it is true by the definition of attributes, so it does not affect normalization or decomposition.

Non-Trivial Functional Dependency

A functional dependency X → Y is non-trivial if Y is not a subset of X — meaning the dependent attribute(s) are not already included in the determinant.

If $Y \subseteq / X$, then $X \rightarrow Y$ is non-trivial.

Examples:

- Student_ID → Student_Name
- Emp_ID → Department
- {Course_ID} → Instructor

11.3.3 Closure of a Set of Functional Dependencies

The closure of a set of functional dependencies (FDs) is a crucial concept in relational database theory. It refers to the complete set of all functional dependencies that can be logically inferred from a given set of FDs using a set of inference rules, known as Armstrong's Axioms. The closure helps in understanding all the implications of a given set of FDs and is instrumental in the normalization process.

The closure of a set of functional dependencies, denoted as F^+, represents every dependency that can be derived logically from the original set F through repeated application of the inference rules (Armstrong's Axioms: reflexivity, augmentation, and transitivity). Computing this closure is essential for several database design tasks, such as testing the equivalence of FD sets, finding candidate keys, and verifying the correctness of decompositions.

For example, if we are given FDs like A → B and B → C, then by applying the transitivity rule, we can infer A → C, which is included in F^+. Determining F^+ helps database designers identify hidden dependencies that are not explicitly stated but can be inferred, thus providing a complete picture of the relationships between attributes. In practice, closure computation ensures that the database schema is logically sound, free from redundancy, and adheres to the principles of dependency preservation during normalization and decomposition.

11.3.4 Armstrong's Axioms

Armstrong's Axioms are a set of inference rules used to derive all possible FDs from a given set:

- **Reflexivity**: If Y is a subset of X, then X → Y.
- **Augmentation**: If X → Y, then XZ → YZ for any Z.
- **Transitivity**: If X → Y and Y → Z, then X → Z.

Finding the Closure

The closure of a set of FDs, denoted as F+F^+F+, is found by repeatedly applying Armstrong's Axioms to the given set FFF until no new FDs can be derived.

Example

Consider a relation R(A,B,C)R(A, B, C)R(A,B,C) with the following FDs:

1. A→BA \to BA→B
2. B→CB \to CB→C

To find the closure F+F^+F+ of the set F={A→B,B→C}F = \{ A \to B, B \to C \}F={A→B,B→C}:

1. Start with the given FDs:
    - A→BA \to BA→B
    - B→CB \to CB→C
2. Apply Transitivity to derive a new FD:
    - Since A→BA \to BA→B and B→CB \to CB→C, by Transitivity, A→CA \to CA→C.
3. The closure F+F^+F+ includes:
    - A→BA \to BA→B
    - B→CB \to CB→C
    - A→CA \to CA→C

The closure F+F^+F+ represents all the functional dependencies that can be inferred from the initial set FFF. It is a comprehensive set that captures all the relationships implied by the original FDs.

Applications of Closure

- **Normalization**: Helps in decomposing relations into normalized forms by identifying all possible FDs.
- **Attribute Closure**: Useful for determining if a certain set of attributes can functionally determine another set of attributes.
- **Candidate Keys**: Assists in identifying candidate keys by examining the attribute closure.

Understanding the closure of a set of FDs is fundamental in database design and normalization, as it ensures that all potential data dependencies are considered when structuring the database schema.

11.3.5 Decomposition Using Functional Dependencies

Decomposition involves breaking down a relation into two or more relations based on functional dependencies to achieve a higher normal form. The decomposition should be lossless and dependency-preserving.

Steps for Decomposition

1. **Identify Functional Dependencies**: Determine the set of functional dependencies that hold for the relation.
2. **Check for Violation of Normal Forms**: Identify if the relation violates any of the normal forms (1NF, 2NF, 3NF, BCNF).
3. **Decompose the Relation**: Use functional dependencies to split the relation into smaller relations that conform to the desired normal form.

Example

Consider a relation R(A,B,C,D)R(A, B, C, D)R(A,B,C,D) with the following functional dependencies:

1. A→BA \to BA→B
2. C→DC \to DC→D

To decompose this relation, we can follow these steps:

1. **Identify Functional Dependencies**: The given FDs are A→BA \to BA→B and C→DC \to DC→D.
2. **Check Normal Form**: Let's assume the relation RRR is not in BCNF because each FD does not have a superkey on the left-hand side.
3. **Decompose the Relation**:
    - Based on A→BA \to BA→B:
        - Create R1(A,B)R1(A, B)R1(A,B).
    - Based on C→DC \to DC→D:
        - Create R2(C,D)R2(C, D)R2(C,D).
    - Ensure that remaining attributes are appropriately placed to preserve all functional dependencies:
        - Combine the remaining attributes into another relation if needed: R3(A,C)R3(A, C)R3(A,C).

Thus, the original relation R(A,B,C,D)R(A, B, C, D)R(A,B,C,D) is decomposed into:

- R1(A,B)R1(A, B)R1(A,B)
- R2(C,D)R2(C, D)R2(C,D)
- R3(A,C)R3(A, C)R3(A,C)

Ensuring Lossless Join

To ensure the decomposition is lossless, the join of the decomposed relations should yield the original relation without any loss of information. This can be verified using the following test:

- For the decomposition RRR into R1R1R1 and R2R2R2:
  - The decomposition is lossless if R1∩R2R1 \cap R2R1∩R2 forms a superkey for either R1R1R1 or R2R2R2.

Ensuring Dependency Preservation

To ensure that the functional dependencies are preserved, the union of the projections of the decomposed relations should cover all the original functional dependencies.

Example of Lossless Join and Dependency Preservation

For the decomposed relations:

- R1(A,B)R1(A, B)R1(A,B)
- R2(C,D)R2(C, D)R2(C,D)
- R3(A,C)R3(A, C)R3(A,C)
- Lossless Join: R1∩R3={A}R1 \cap R3 = \{A\}R1∩R3={A} and R2∩R3={C}R2 \cap R3 = \{C\}R2∩R3={C}. Since AAA and CCC can act as keys in their respective decomposed relations, the join is lossless.
- Dependency Preservation: The original dependencies A→BA \to BA→B and C→DC \to DC→D are maintained in R1R1R1 and R2R2R2.

11.4 Summary

This chapter discussed the key principles that guide the **design of relation schemas** in a database to ensure logical consistency, minimal redundancy, and reliable data management. It began with **informal design guidelines**, emphasizing that each relation should represent a single, well-defined concept with attributes that have clear semantics. The importance of avoiding **redundant data**, **null values**, and **spurious tuples** was highlighted, as these issues lead to data anomalies and inaccuracies during database operations. Proper decomposition and meaningful attribute relationships were shown to be essential in creating schemas that are easy to maintain and free from anomalies.

The chapter also introduced the theoretical foundation of **Functional Dependencies (FDs)**—a cornerstone of relational database design. Concepts such as **trivial and non-trivial dependencies**, **closure of FDs**, and **Armstrong's Axioms** were explained, demonstrating their role in normalization and schema refinement. By understanding how attributes determine one another, designers can identify redundant data relationships and decompose

relations effectively. Overall, the chapter establishes a strong basis for progressing toward **normal forms**, ensuring that relational databases are both efficient and logically sound.

11.5 Technical Terms

1. Relation Schema
2. Functional Dependency (FD)
3. Trivial Functional Dependency
4. Non-Trivial Functional Dependency
5. Closure of Functional Dependencies (F^+)
6. Armstrong's Axioms
7. Update Anomalies
8. Lossless Decomposition
9. Spurious Tuples
10. Normalization

11.6 Self-Assessment Questions

Essay Questions

1. Explain the importance of semantics in relation attributes with suitable examples.
2. Discuss redundancy and different types of update anomalies in database design.
3. Define Functional Dependencies and explain their significance in database normalization.
4. Describe the concept of closure of functional dependencies and its applications.
5. Explain Armstrong's Axioms and show how they are used to derive new dependencies.

Short Questions

1. What is the difference between trivial and non-trivial functional dependencies?
2. Define spurious tuples and explain how they can be prevented.
3. What are null values, and why should their use be minimized?
4. What is meant by lossless decomposition?
5. List any two uses of closure in relational database design.

11.7 Suggested Readings

1. Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, Pearson Education.
2. C. J. Date, *An Introduction to Database Systems*, Addison-Wesley.
3. Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, *Database System Concepts*, McGraw Hill.
4. Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*, McGraw Hill.
5. Thomas Connolly and Carolyn Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*, Pearson.
6. Bipin C. Desai, *An Introduction to Database Systems*, Galgotia Publications.
7. Peter Rob and Carlos Coronel, *Database Systems: Design, Implementation, and Management*, Cengage Learning.
8. Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, *Database Systems: The Complete Book*, Pearson.
9. Alexis Leon and Mathews Leon, *Database Management Systems*, Vikas Publishing.
10. Ivan Bayross, *SQL, PL/SQL: The Programming Language of Oracle*, BPB Publications.

Dr. Vasantha Rudramalla

LESSON- 12
NORMALIZATION

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Normalization for Relational Databases. The chapter began Normal Forms Based in Primary Keys, General Definitions of Second and Third Normal Forms, Boyce-Codd Normal Form .After completing this chapter, the student will understand Normalization for Relational Databases.

12.1 INTRODUCTION

Database normalization is a critical process in relational database design that organizes data to reduce redundancy and improve data integrity. By structuring data into smaller, related tables, normalization ensures that the database is efficient, scalable, and easier to maintain. This process not only optimizes storage space but also enhances the accuracy and consistency of the data by eliminating anomalies and minimizing the chances of data duplication.

The primary goals of normalization are to eliminate redundant data, minimize update anomalies, and simplify data structures. By dividing large tables into smaller, more manageable ones and defining clear relationships between them, normalization aims to ensure that each piece of data is stored only once. This improves data consistency and integrity, making the database more reliable and easier to query and update. Ultimately, normalization contributes to a more efficient database system that can handle complex queries and data manipulation tasks with ease

.

12.2 NORMAL FORMS BASED ON PRIMARY KEYS

Normal forms based on primary keys are standards used to organize database schemas to reduce redundancy and improve data integrity. These normal forms include the First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).

12.2.1 First Normal Form (1NF)

It is the foundational stage of database normalization that ensures the table structure is simplified and data is stored in a tabular format with no repeating groups. A table is in 1NF if it meets the following criteria:

1. **Atomicity**: Each column contains atomic (indivisible) values, meaning that each cell holds a single value rather than a set of values or a list.
2. **Uniqueness**: Each column should have unique names, and the order in which data is stored does not matter.
3. **No Repeating Groups**: Each record (row) should be unique, and no two rows should have the same combination of values in all columns.

Example
Consider a table Students before normalization:

StudentID	Name	Courses
1	John Smith	Math, Science
2	Jane Doe	Math, History
3	Bob Brown	Literature, Science

In this table, the Courses column contains multiple values, violating 1NF.
To convert this table to 1NF, we split the multi-valued column into separate rows:

StudentID	Name	Course
1	John Smith	Math
1	John Smith	Science
2	Jane Doe	Math
2	Jane Doe	History
3	Bob Brown	Literature
3	Bob Brown	Science

In this normalized table:

- Each cell contains only a single value.
- The table structure is simplified.
- No repeating groups exist.

By ensuring the table is in 1NF, we have eliminated any repeating groups and made each column contain only atomic values, setting a solid foundation for further normalization processes.

Algorithm: First Normal Form (1NF)

1. **Start.**
2. **Identify the relation (table)** — Examine the attributes (columns) and tuples (rows) in the given unnormalized table.

1. **Check for repeating groups or multivalued attributes.**
   a. If an attribute contains multiple values (e.g., a list or set), it violates 1NF.
2. **Eliminate repeating groups.**
   a. For each repeating group, create a separate tuple for every unique value.
   b. Ensure each cell in the relation holds **only a single atomic value**.
3. **Assign proper attribute names** to all data items, if not already done, ensuring clear column definitions.
4. **Identify a primary key** that uniquely identifies each tuple (row) in the table.
5. **Reorganize the table** such that:
   a. Each column contains atomic values only.
   b. Each row is uniquely identified by the primary key.
6. **Check for data consistency** — Ensure that no information has been lost or duplicated during conversion.
7. **Stop.**

Example
Unnormalized Table:

Student_ID	Student_Name	Subjects
101	Ravi Kumar	DBMS, Java, Python
102	Meena Sharma	AI, ML

After Applying 1NF:

Student_ID	Student_Name	Subject
101	Ravi Kumar	DBMS
101	Ravi Kumar	Java
101	Ravi Kumar	Python
102	Meena Sharma	AI
102	Meena Sharma	ML

Result:
The relation is now in **First Normal Form (1NF)** — all values are **atomic**, and there are **no repeating groups**.
First Normal Form (1NF) – Relational Algebra (Brief)
Given Relation:

STUDENT(Student_ID, Student_Name, Subjects)
Here, **Subjects** is a **multivalued attribute**, violating 1NF.

Relational Algebra Expression:

$$STUDENT_{1NF} = UNNEST_{Subjects}(STUDENT)$$

Resulting Relation:
STUDENT_1NF(Student_ID, Student_Name, Subject)

Student_ID	Student_Name	Subject
101	Ravi Kumar	DBMS
101	Ravi Kumar	Java
101	Ravi Kumar	Python

The **UNNEST** operation converts multivalued attributes into **atomic values**, producing one tuple per subject. Thus, the relation now satisfies **First Normal Form (1NF)** — all attributes hold single, indivisible values.

12.2.2 Second Normal Form (2NF)

It builds on the principles of First Normal Form (1NF) by further reducing redundancy and ensuring that every non-key attribute is fully functionally dependent on the entire primary key. A table is in 2NF if it meets the following criteria:

1. **First Normal Form (1NF)**: The table must already be in 1NF.
2. **Full Functional Dependency**: Every non-key attribute must depend on the entire primary key, not just a part of it. This rule is particularly relevant for tables with composite primary keys.

Example
Consider a table Enrollments that is in 1NF but not in 2NF:

StudentID	CourseID	StudentName	CourseName	Instructor
1	101	John Smith	Math	Dr. Johnson
1	102	John Smith	Science	Dr. Smith
2	101	Jane Doe	Math	Dr. Johnson
3	103	Bob Brown	History	Dr. Adams

In this table:

- The primary key is the composite key (StudentID, CourseID).
- StudentName depends only on StudentID.
- CourseName and Instructor depend only on CourseID.

To convert this table to 2NF, we need to remove partial dependencies by creating separate tables:

1. **Students Table**:
   CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   StudentName VARCHAR(50)
   );

StudentID	StudentName
1	John Smith
2	Jane Doe
3	Bob Brown

Courses Table:

```
CREATE TABLE Courses  (
CourseID INT PRIMARY KEY,
    CourseName VARCHAR(50),
    Instructor VARCHAR(50)
    );
```

CourseID	CourseName	Instructor
101	Math	Dr. Johnson
102	Science	Dr. Smith
103	History	Dr. Adams

Enrollments Table:

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

StudentID	CourseID
1	101
1	102
2	101
3	103

In this normalized structure:

- The Students table ensures that Student Name is fully dependent on Student ID.
- The Courses table ensures that Course Name and Instructor are fully dependent on Course ID.
- The Enrollments table links students to courses without any partial dependencies.

By achieving 2NF, we have eliminated partial dependencies, thus further reducing redundancy and potential update anomalies.

Algorithm: Conversion to 2NF

1. Start with a relation R in **1NF**.
2. Identify the **primary key** — if it is **composite**, check for **partial dependencies**.
3. If any **non-key attribute** depends on only part of the key, move it to a **new relation** along with that part of the key.
4. Keep attributes **fully functionally dependent** on the whole primary key in the original relation.
5. The resulting set of relations are in **2NF**.

Relational Algebra Expression

$$R_{2NF} = \pi_{PartialKey,Dependent}(R) \cup \pi_{FullKey,RemainingAttributes}(R)$$

Example
1NF Relation:
COURSE(Course_ID, Student_ID, Student_Name, Marks)

Functional Dependencies:

- (Course_ID, Student_ID) → Marks
- Student_ID → Student_Name

Here, Student_Name depends **only** on part of the composite key (Student_ID) → **Partial Dependency**.

Decompose into:

1. STUDENT(Student_ID, Student_Name)
2. COURSE_ENROLL(Course_ID, Student_ID, Marks)

Now, all non-key attributes depend fully on their respective primary keys → **2NF**.

12.2.3 Third Normal Form (3NF)

It builds on the principles of Second Normal Form (2NF) by further eliminating redundancy and ensuring that every non-key attribute is not only fully functionally dependent on the primary key but also non-transitively dependent on it. A table is in 3NF if it meets the following criteria:

1. **Second Normal Form (2NF)**: The table must already be in 2NF.
2. **No Transitive Dependency**: No non-key attribute should depend on another non-key attribute. In other words, all non-key attributes must depend only on the primary key.

Example
Consider a table StudentEnrollments that is in 2NF but not in 3NF:

StudentID	CourseID	InstructorID	InstructorName
1	101	201	Dr. Johnson
2	102	202	Dr. Smith
3	101	201	Dr. Johnson

In this table:

- The primary key is the composite key (StudentID, CourseID).
- InstructorName depends on InstructorID, which is a non-key attribute, creating a transitive dependency.

To convert this table to 3NF, we need to remove the transitive dependency by creating separate tables:

1. Instructors Table:

```
CREATE TABLE Instructors (
InstructorID INT PRIMARY KEY,
InstructorName VARCHAR(50)
);
```

InstructorID	InstructorName
201	Dr. Johnson
202	Dr. Smith

Enrolments Table:

```
CREATE TABLE Enrollments (
StudentID INT,
CourseID INT,
InstructorID INT,
PRIMARY KEY (StudentID, CourseID),
FOREIGN KEY (InstructorID) REFERENCES Instructors(InstructorID)
);
```

StudentID	CourseID	InstructorID
1	101	201
2	102	202
3	101	201

In this normalized structure:

- The Instructors table stores the instructor information, ensuring Instructor Name is fully dependent on Instructor ID.
- The Enrollments table links students to courses and instructors without any transitive dependencies.

By achieving 3NF, we have eliminated transitive dependencies, further reducing redundancy and potential anomalies in the database. This normalization ensures that all non-key attributes depend directly on the primary key and not on other non-key attributes

Algorithm: Conversion to 3NF

1. Start with relations in 2NF.
2. Identify transitive dependencies — where a non-key attribute depends on another non-key attribute.
3. Remove transitive dependencies by creating new relations:
   - Move the dependent attributes and the determinant to a separate table.
4. Retain only attributes directly dependent on the primary key in the original table.
5. The resulting relations are in 3NF.

Relational Algebra Expression

$R_{3NF} = \pi_{Key,DirectDependents}(R) \cup \pi_{NonKeyDeterminant,Dependent}(R)$

Example
2NF Relation:
EMPLOYEE(Emp_ID, Emp_Name, Dept_ID, Dept_Name)

Functional Dependencies:

- Emp_ID → Emp_Name, Dept_ID
- Dept_ID → Dept_Name

Dept_Name depends on Dept_ID (a non-key attribute) → Transitive Dependency**.**

Decompose into:

1. EMPLOYEE(Emp_ID, Emp_Name, Dept_ID)
2. DEPARTMENT(Dept_ID, Dept_Name)

No partial or transitive dependencies remain → 3NF achieved.

Normal Form	Removes	Key Condition	Result
1NF	Multivalued attributes	Attributes must be atomic	Flat, atomic relation
2NF	Partial dependency	Non-key attributes fully depend on full key	No partial dependency
3NF	Transitive dependency	Non-key depends only on key	No transitive dependency

12.3 BOYCE-CODD NORMAL FORM (BCNF)

12.3.1 Boyce-Codd Normal Form (BCNF) is an advanced version of the Third Normal Form (3NF) in database normalization. BCNF aims to eliminate redundancy and potential anomalies by ensuring that all functional dependencies in a relation are appropriately managed.

A relation is in Boyce-Codd Normal Form (BCNF) if it satisfies the following conditions:

1. **It is in Third Normal Form (3NF)**: The relation must already meet all the requirements of 3NF.
2. **Every determinant is a candidate key**: For every functional dependency X→YX \to YX→Y, XXX must be a superkey (a set of attributes that uniquely identify a tuple in a relation).

12.3.2 Difference Between 3NF and BCNF

While 3NF ensures that non-key attributes are non-transitively dependent on the primary key, BCNF takes this a step further by addressing situations where 3NF might still allow certain

types of redundancy. Specifically, BCNF ensures that even when a functional dependency involves a part of a candidate key, it does not violate normalization principles.

Example

Consider a table Courses with the following attributes:

CourseID	Instructor	Room
101	Dr. Smith	Room 1
102	Dr. Brown	Room 2
103	Dr. Smith	Room 2

Functional dependencies in this table:

1. CourseID→InstructorCourseID \to InstructorCourseID→Instructor
2. Instructor→RoomInstructor \to RoomInstructor→Room

The composite key here can be either CourseID CourseID CourseID or InstructorInstructorInstructor, as each uniquely identifies a course offering.

Identifying BCNF Violation

In this case, Instructor→RoomInstructor \to RoomInstructor→Room is problematic because Instructor is not a superkey. This means the table is in 3NF (as there are no transitive dependencies), but not in BCNF.

Difference Between 3NF and BCNF

Feature	**Third Normal Form (3NF)**	**Boyce–Codd Normal Form (BCNF)**
Definition	A relation is in 3NF if for every functional dependency **X** → **Y**, at least one of the following holds:1. **X** is a superkey, or2. **Y** is a prime attribute (part of some candidate key).	A relation is in BCNF if for every functional dependency **X** → **Y**, **X** is a superkey.
Condition Relaxation	Allows non-superkey determinants if **Y** is a prime attribute.	Stricter — every determinant **must** be a superkey.
Anomaly Removal	Removes partial and transitive dependencies but may still allow some anomalies.	Removes all anomalies related to functional dependencies.
Dependency Preservation	Always possible — 3NF decomposition preserves all dependencies.	May not preserve all dependencies after decomposition.
Lossless Join Property	Always ensures a lossless join.	Also ensures a lossless join.
Focus	Balances **dependency preservation** and **normalization**.	Focuses on **eliminating redundancy** completely.
Level of	Less restrictive than BCNF.	More restrictive — special

Restriction		case of 3NF.
Example (Violating 3NF/BCNF)	In 3NF: Relation may allow FD where determinant is not a key but dependent is a prime attribute.	In BCNF: Such an FD is **not allowed** — determinant must be a key.
Preferred When	Dependency preservation is more important (e.g., in practical systems).	Data integrity and anomaly elimination are more critical.

12.3.3 Examples and Decomposition into BCNF

To achieve BCNF, we decompose the table into two relations:

Instructors Table:

```
CREATE TABLE Instructors (
   Instructor VARCHAR(50) PRIMARY KEY,
   Room VARCHAR(50)
   );
```

Instructor	Room
Dr. Smith	Room 2
Dr. Brown	Room 2

Courses Table:

```
CREATE TABLE Courses (
   CourseID INT PRIMARY KEY,
   Instructor VARCHAR(50),
   FOREIGN KEY (Instructor) REFERENCES Instructors(Instructor)
   );
```

CourseID	Instructor
101	Dr. Smith
102	Dr. Brown
103	Dr. Smith

Importance

Achieving BCNF ensures that all functional dependencies are properly managed, eliminating redundancy and potential update anomalies. BCNF is particularly useful in complex database designs where multiple candidate keys and intricate dependencies exist, providing a higher level of normalization than 3NF.

By decomposing relations into BCNF, database designers can create more robust, reliable, and efficient database schemas that uphold data integrity and consistency.

Algorithm to decompose a relation into BCNF (high level)

1. Input: relation R with a set of functional dependencies F.
2. If every FD $X \to Y$ in F^+ has X as a superkey of R, then **R is in BCNF** — stop.
3. Otherwise pick a violating FD $X \to Y$ where X is **not** a superkey.
4. Decompose R into two relations:
   a. $R_1 = \pi_{X \cup Y}(R)$ (attributes $X \cup Y$)
   b. $R_2 = \pi_{R \setminus (Y)}(R)$ (attributes of R with Y removed)
5. Replace R by R_1 and R_2; compute the projection of F on these relations (i.e., relevant FDs).
6. Recursively apply the algorithm to R_1 and R_2 until every relation is in BCNF.
7. Output: set of BCNF relations (decomposition).

- Use BCNF to eliminate anomalies caused by FDs whose determinants are not keys.
- Expect possible loss of dependency preservation — if preserving all FDs is essential (e.g., for efficient enforcement), 3NF may be preferred because it always allows dependency preservation while removing many anomalies.

12.4 SUMMARY

This chapter, we explored the concept of normalization, a systematic approach to organizing data in a database to reduce redundancy and improve data integrity. The process of normalization involves dividing larger, complex tables into smaller, related ones using well-defined rules known as normal forms. We began with First Normal Form (1NF), which eliminates repeating groups by ensuring that each attribute contains only atomic (indivisible) values. Second Normal Form (2NF) extends this by removing partial dependencies—ensuring that all non-key attributes are fully functionally dependent on the entire primary key. Third Normal Form (3NF) further refines this by removing transitive dependencies, where non-key attributes depend on other non-key attributes.

We also studied the Boyce–Codd Normal Form (BCNF), which is a stricter version of 3NF, ensuring that every determinant is a candidate key. BCNF provides a higher degree of data consistency by eliminating all possible anomalies related to functional dependencies. The chapter compared 3NF and BCNF, emphasizing that while 3NF allows some redundancy under certain conditions, BCNF enforces stricter dependency rules. Practical examples and decomposition steps illustrated how complex relations can be broken down into BCNF-compliant structures without losing data. Overall, normalization ensures efficient, reliable,

and logically structured databases that support easier maintenance and accurate query processing.

12.5 TECHNICAL TERMS

- Functional Dependency (FD)
- Normalization
- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce–Codd Normal Form (BCNF)
- Anomaly
- Decomposition
- Lossless Decomposition
- Reliability

12.6 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Illustrate about BCNF
2. Describe about 2NF and 3NF
3. Explain about Functional Dependency

Short Notes:

1. Write Transaction Dependency
2. Define Full functional dependency
3. Explain about Armstrong's Axioms

12.7 SUGGESTED READINGS

1. Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM, 13(6), 377-387.
2. Date, C. J. (2003). "An Introduction to Database Systems." 8th Edition. Addison-Wesley.
3. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). "Database System Concepts." 6th Edition. McGraw-Hill.
4. Ullman, J. D., & Widom, J. (2008). "A First Course in Database Systems." 3rd Edition. Pearson.

Dr. Vasantha Rudramalla

LESSON- 13

RELATIONAL DATABASE DESIGN ALGORITHMS

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Relational Database Design Algorithms. The chapter began Informal Design Guidelines for Relation Schemas and Algorithms for Relational Database Schema Design with example. After completing this chapter, the student will understand Relational Database Design Algorithms.

13.1 INTRODUCTION

Designing a good relational database schema is a crucial step in ensuring that data is stored efficiently, consistently, and without redundancy. Poorly designed schemas often lead to update anomalies, data inconsistency, and redundant information. To avoid these problems, designers apply systematic approaches such as normalization and decomposition algorithms that transform large, complex relations into smaller, well-structured relations.

In relational database theory, decomposition refers to the process of breaking down a relation schema into multiple smaller schemas that satisfy certain desirable properties. The key goal is to simplify the database structure while preserving important characteristics such as data integrity, dependency preservation, and lossless join. These ensure that no data is lost or misrepresented during the decomposition process.

For example, consider a relation EMP_DEPT(Emp_ID, Emp_Name, Dept_Name, Dept_Location). If multiple employees belong to the same department, repeating Dept_Name and Dept_Location causes redundancy. Decomposing this into:

EMPLOYEE(Emp_ID, Emp_Name, Dept_Name)
DEPARTMENT(Dept_Name, Dept_Location)

removes redundancy while maintaining all relationships through a foreign key.
Thus, relational design algorithms focus on identifying dependencies and decomposing relations so that the resulting schema is both efficient and logically sound.

13.2 PROPERTIES OF RELATIONAL DECOMPOSITIONS

Decomposition is an essential part of database design. However, not every decomposition is beneficial — an improper decomposition can lead to data loss or dependency violations. To evaluate the quality of a decomposition, we use specific properties that every good decomposition should satisfy.

13.2.1 Need for Decomposition

- A relation schema may contain data redundancy and anomalies (insertion, deletion, update).
- Decomposition helps simplify complex relations into smaller ones, making them easier to manage.
- Each resulting relation should maintain **data integrity** and **consistency** with the original schema.

13.2.2 Desirable Properties of Decomposition

Desirability of Decomposition refers to the benefits and considerations involved in breaking down a relational database schema into smaller, more manageable relations. Decomposition is often guided by the goals of eliminating redundancy, preventing anomalies, and ensuring data integrity. The key criteria for desirable decomposition include maintaining the lossless join property and preserving dependencies.

Importance

- **Reduces Redundancy**: Eliminates duplicate data, thereby saving storage space and ensuring that data updates are more efficient.
- **Prevents Anomalies**: Helps avoid update, insertion, and deletion anomalies that can lead to inconsistent and unreliable data.
- **Enhances Data Integrity**: Ensures that the integrity constraints of the original schema are maintained, thus preserving the accuracy and consistency of the data.
- **Improves Query Performance**: By creating more focused and smaller tables, decomposition can enhance the performance of queries and updates.

Example:
Given FDs:
Emp_ID → Emp_Name, Dept_ID
Dept_ID → Dept_Name

If decomposition allows each FD to be enforced locally in one of the decomposed relations, dependency preservation is satisfied.

A good decomposition must satisfy **three key properties**:
(a) Attribute Preservation
All attributes from the original relation **R** must appear in at least one of the decomposed relations.
Formally:

$$R = R_1 \cup R_2 \cup \ldots \cup R_n$$

where every attribute in R appears in at least one R_i.
✅ *Example:*
Original Relation: STUDENT(Student_ID, Name, Course, Instructor)
Decomposed into:
STUDENT_INFO(Student_ID, Name)
COURSE_INFO(Course, Instructor)
All attributes {Student_ID, Name, Course, Instructor} are preserved.

(b) Lossless (Non-Additive) Join Property

The **Lossless Join Property** is a critical feature in relational database design that ensures data integrity during the decomposition of a relation into smaller relations. A decomposition of a relation RRR into two or more relations R1,R2,...,RnR1, R2, ..., RnR1,R2,...,Rn is said to have the lossless join property if, by joining these decomposed relations, we can exactly recreate the original relation RRR without any loss of information or introduction of spurious tuples.

Importance

- **Data Integrity**: Ensures that the decomposed relations, when joined, yield the exact original dataset, preserving all data accurately.
- **Consistency**: Prevents anomalies and inconsistencies that can arise from improper decomposition.

- **Database Efficiency**: Enables effective normalization by decomposing tables to reduce redundancy while maintaining the ability to reconstruct the original data accurately.

This ensures that no information is lost when decomposed relations are joined back together. For a decomposition of relation **R** into $\mathbf{R_1}$ and $\mathbf{R_2}$, the decomposition is *lossless* if:

$$R_1 \bowtie R_2 = R$$

That is, joining R_1 and R_2 on their common attributes must reproduce the original relation **exactly**.

✅ *Example:*
If EMP(Emp_ID, Emp_Name, Dept_ID, Dept_Name) is decomposed into:
EMPLOYEE(Emp_ID, Emp_Name, Dept_ID)
DEPARTMENT(Dept_ID, Dept_Name)

Then joining on Dept_ID gives back the original EMP relation without generating spurious tuples.

(c) Dependency Preservation

Dependency Preservation is a crucial property in relational database design that ensures all functional dependencies from the original relation are still enforceable after decomposition into smaller relations. A decomposition is said to preserve dependencies if every functional dependency in the original schema can be derived from the set of dependencies in the decomposed schema without requiring access to the original relation.

Importance

- **Maintains Data Integrity**: Ensures that all original constraints are preserved and can be enforced in the decomposed relations, preventing data anomalies.
- **Simplifies Constraint Management**: Allows constraints to be checked and enforced locally within the decomposed relations without needing to join them back together.
- **Efficient Updates and Queries**: Improves performance by enabling efficient updates and queries while maintaining the integrity constraints.

Functional dependencies (FDs) describe constraints that must hold among attributes. A decomposition preserves dependencies if all FDs from the original schema can be checked using only the decomposed relations — without performing costly joins.

13.2.3 Trade-offs in Decomposition

- Sometimes, achieving **lossless join** and **dependency preservation** simultaneously is not possible.
- **BCNF decomposition** ensures lossless join but may lose dependency preservation.
- **3NF decomposition** ensures both dependency preservation and lossless join but may retain minimal redundancy.

Hence, database designers often **balance** between strict normalization and practical enforceability of constraints.

13.2.4 Example – Decomposition of Employee Relation

Given relation:
EMP(Emp_ID, Emp_Name, Dept_ID, Dept_Name)
FDs: Emp_ID → Emp_Name, Dept_ID
Dept_ID → Dept_Name

Decompose into:
EMPLOYEE(Emp_ID, Emp_Name, Dept_ID)
DEPARTMENT(Dept_ID, Dept_Name)

This decomposition:

- Preserves attributes
- Maintains lossless join (common key: Dept_ID)
- Preserves dependencies

✅ Hence, it is a **good decomposition**.

13.3 ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

Algorithms for Relational Database Schema Design are systematic methods used to transform a database schema into a normalized form. These algorithms ensure that the

13.3.1 Algorithm for Testing Lossless Join Decomposition

The algorithm for testing lossless join decomposition ensures that the join of decomposed relations results in the original relation. This involves checking if the common attributes in the decomposed relations form a superkey.

Steps in the Algorithm

1. **Identify the Decomposition**:
   - Let the original relation RRR be decomposed into two or more relations R1,R2,...,RnR1, R2, ..., RnR1,R2,...,Rn.
2. **Construct the Join Dependency Matrix**:
   - Create a matrix where each row represents an attribute in the original relation RRR, and each column represents a decomposed relation RiRiRi.
   - Initialize the matrix with zeros.
3. **Mark the Attributes**:
   - For each decomposed relation RiRiRi, mark the columns corresponding to the attributes present in RiRiRi.
4. **Propagation of Marks**:
   - Propagate the marks across the matrix based on the common attributes between decomposed relations.
5. **Test for Lossless Join**:
   - Check if each row in the matrix has at least one column that is fully marked. This indicates that the original relation can be perfectly reconstructed from the decomposed relations.

Example

1. Consider a relation R(A,B,C) with the following functional dependencies:
   A→BA to BA→B
   B→CB to CB→C
2. Decompose R into R1(A,B) and R2(B,C).

Identify the Decomposition:

- R is decomposed into R1(A,B) and R2(B,C).

Construct the Join Dependency Matrix:

	R1(A, B)	R2(B, C)
A	0	0
B	0	0
C	0	0

1. **Mark the Attributes**:
   - For R1(A,B):
   - Mark columns for attributes A and B.

	R1(A, B)	R2(B, C)
A	1	0
B	1	0
C	0	0

For R2(B,C):

- Mark columns for attributes B and C.

	R1(A, B)	R2(B, C)
A	1	0
B	1	1
C	0	1

2. **Propagation of Marks**:
   - Propagate the marks based on the common attribute B.

5. Test for Lossless Join:

- Check each row:
  - Row for A has at least one mark in the column corresponding to R1.
  - Row for B has marks in both columns.
  - Row for C has at least one mark in the column corresponding to R2.

Since each row has at least one column that is fully marked, the decomposition has the lossless join property.

The algorithm for testing lossless join decomposition ensures that decomposing a relation into smaller relations does not result in the loss of any data. This property is essential for maintaining data integrity and consistency in a relational database schema.

13.3.2Algorithm for Dependency Preservation

This algorithm verifies that all functional dependencies are preserved in the decomposed schema. It involves checking if the closure of the functional dependencies in the decomposed relations includes all original dependencies.

The **Algorithm for Dependency Preservation** is used to verify that all functional dependencies of the original relation are preserved in the decomposed schema. This ensures that the integrity constraints enforced by the functional dependencies can still be checked without needing to access the original relation.

Steps in the Algorithm

1. **Identify the Functional Dependencies**:
   - Let R be the original relation with a set of functional dependencies F.
2. **Decompose the Relation**:
   - Decompose RRR into a set of relations R1,R2,...,Rn.
3. **Project Functional Dependencies**:
   - For each decomposed relation Ri, compute the projection of F on Ri, denoted as Fi. The projection of F on Ri includes all functional dependencies in F that involve only attributes of Ri.
4. **Compute the Closure**:
   - Compute the closure of the union of the projected dependencies F1∪F2∪...∪Fn , denoted as (F1∪F2∪...∪Fn)+.
5. **Check for Dependency Preservation**:
   - Verify that every functional dependency in F is included in the closure (F1∪F2∪...∪Fn)+. If all dependencies in Fare present in the closure, then the decomposition preserves dependencies.

Example

Consider a relation R(A,B,C) with functional dependencies:

1. A→B
2. B→C

Decompose RRR into R1(A,B)R1(A, B)R1(A,B) and R2(B,C)R2(B, C)R2(B,C).

1. **Identify the Functional Dependencies**:
   - F={A→B,B→C}
2. **Decompose the Relation**:
   - Decomposed relations are R1(A,B) and R2(B,C)
3. **Project Functional Dependencies**:
   - For R1(A,B):
   - Projection F1={A→B}
   - For R2(B,C):
   - Projection F2={B→C}
4. **Compute the Closure**:
   - F1∪F2={A→B,B→C}
   - Compute the closure (F1∪F2)+:

- Start with {A→B,B→C}
- From A→B and B→C, by transitivity, derive A→C
- (F1∪F2)+={A→B,B→C,A→C}

5. **Check for Dependency Preservation**:
   - Verify that every functional dependency in F is in (F1∪F2)+ :
   - A→B is in (F1∪F2)+
   - B→C is in (F1∪F2)+
   - All dependencies from F are preserved in the closure.

Since all functional dependencies from the original set F are preserved in the closure of the projected dependencies, the decomposition is dependency-preserving.

The algorithm for dependency preservation ensures that all functional dependencies of the original relation are maintained in the decomposed schema. This is essential for ensuring that the integrity constraints can still be enforced without requiring access to the original relation, thereby maintaining the consistency and reliability of the database schema.

13.3.3 Algorithm for Finding Canonical Cover

Purpose:
To eliminate redundant attributes and FDs from a given set of dependencies.

Steps:
1. Split RHS of each FD so that each has a single attribute.
2. Remove extraneous attributes from LHS and RHS using closure test.
3. Remove redundant dependencies.

Example:
Given FDs: {A → BC, B → C, A → B}
Canonical Cover: {A → B, B → C}.

13.3.4 Algorithm for 3NF Decomposition
Steps:
1. Compute canonical cover Fc.
2. For each FD X → Y in Fc, create a relation R(X ∪ Y).
3. If none of these relations contain a key for R, add one relation with a key.
4. Ensure lossless join.

Produces a dependency-preserving, lossless decomposition.

13.3.5 Algorithm for BCNF Decomposition
Steps:
1. Start with R and FDs.
2. If R is not in BCNF, find dependency X → Y that violates BCNF.
3. Decompose R into:

$R1 = X \cup Y$
$R2 = R – (Y – X)$
1. Repeat for R_1 and R_2 until all relations are in BCNF.

13.3.6 Algorithm for Computing Attribute Closure

Used to determine candidate keys or infer dependencies.

Steps:

1. Start with $X^+ = X$.
2. For each FD $Y \rightarrow Z$ in F, if $Y \subseteq X^+$, add Z to X^+.
3. Repeat until X^+ stops changing.
4. X^+ now represents all attributes functionally determined by X.

Example:

FDs: $\{A \rightarrow B, B \rightarrow C\}$
Closure of A = {A, B, C}.

13.3.7 Example – University Database Design

Given Relation:
COURSE(Course_ID, Title, Dept_ID, Instructor, Credits)

FDs:

- Course_ID → Title, Dept_ID, Credits
- Dept_ID → Instructor

Step 1: Identify redundancy — Instructor depends on Dept_ID.

Step 2: Decompose into:
COURSE(Course_ID, Title, Dept_ID, Credits)
DEPARTMENT(Dept_ID, Instructor)

Step 3: Verify:

- Attribute preservation
- Lossless join (via Dept_ID)
- Dependency preservation

13.4 Normalization and Schema Refinement

Algorithms provide a systematic foundation for schema normalization, ensuring all relations meet 1NF–BCNF standards through dependency testing and decomposition. Normalization improves clarity, reduces anomalies, and makes schema maintenance easier. However, over-normalization can fragment data and degrade query performance, so designers must strike a balance between theoretical perfection and practical efficiency.

13.5 Design Example – Library Database

Given Relation:

LIBRARY(Book_ID, Title, Author, Category, Borrower_ID, Borrower_Name)
FDs:

- Book_ID → Title, Author, Category
- Borrower_ID → Borrower_Name
- Book_ID → Borrower_ID

After decomposition:
BOOK(Book_ID, Title, Author, Category)
BORROWER(Borrower_ID, Borrower_Name)
TRANSACTION(Book_ID, Borrower_ID)

- All anomalies removed
- Lossless join ensured
- Dependencies preserved

13.6 Summary

This lesson presented the **formal foundation and algorithms** of relational database design. You learned about key decomposition properties — **attribute preservation**, **lossless join**, and **dependency preservation** — and how to test them using **algorithmic techniques**. Design algorithms such as **canonical cover**, **attribute closure**, **3NF and BCNF decomposition** enable structured schema refinement.
The algorithms ensure that relational designs are **robust**, **redundancy-free**, and **efficient**, forming the backbone of modern database normalization and schema optimization.

13.7 Technical Terms

1. Decomposition
2. Functional Dependency
3. Canonical Cover
4. Lossless Join
5. Dependency Preservation
6. Attribute Closure
7. 3NF Decomposition
8. BCNF Decomposition
9. Redundancy
10. Normalization

13.8 Self-Assessment Questions

Essay Questions

1. Explain the properties of a good relational decomposition with suitable examples.
2. Discuss the algorithm for testing lossless join decomposition.
3. Describe how dependency preservation can be ensured during schema design.
4. Explain the role and computation of the canonical cover.
5. Differentiate between 3NF and BCNF decomposition algorithms.

Short Questions

1. Define lossless join property.
2. What is attribute preservation?
3. State the purpose of attribute closure.
4. Write one difference between 3NF and BCNF.
5. List the steps of the canonical cover algorithm.

13.9 Suggested Readings

1. Ramez Elmasri & Shamkant B. Navathe — *Fundamentals of Database Systems*, Pearson.
2. C.J. Date — *An Introduction to Database Systems*, Addison-Wesley.
3. Abraham Silberschatz, H.F. Korth, & S. Sudarshan — *Database System Concepts*, McGraw Hill.
4. Raghu Ramakrishnan & Johannes Gehrke — *Database Management Systems*, McGraw Hill.
5. Thomas Connolly & Carolyn Begg — *Database Systems: A Practical Approach*, Pearson.

Dr. U. Surya Kameswari

LESSON- 14
FURTHER DEPENDENCIES

AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the concept of Further Dependencies. The chapter began Functional dependencies, After completing this chapter, the student will understand Further Dependencies which includes 4NF,5NF,6NF and etc.

14.1 INTRODUCTION

As database systems grow in complexity, relations that already satisfy **Boyce–Codd Normal Form (BCNF)** may still exhibit subtle forms of redundancy.
This residual redundancy often arises from **multivalued** or **join dependencies** that are not captured by functional dependencies alone. To address these, higher normal forms — the **Fourth Normal Form (4NF)** and the **Fifth Normal Form (5NF)** — are introduced.
These advanced normal forms extend the principles of normalization by eliminating redundancies that occur due to the presence of **independent multi-valued attributes** or **complex join relationships** among multiple tables.

In addition, other dependency types such as **inclusion dependencies**, **temporal**, and **domain-key constraints** further refine schema design by imposing relationships across tables.
The goal of this lesson is to understand:

- How **multivalued dependencies (MVDs)** can cause redundancy even in BCNF relations,
- How **join dependencies (JDs)** arise and how to achieve **Fifth Normal Form (5NF)**,
- The role of **inclusion dependencies** in maintaining referential integrity, and
- The importance of other advanced dependencies for maintaining data accuracy and consistency.

14.2 MULTIVALUED DEPENDENCIES AND FOURTH NORMAL FORM

14.2.1 Concept of Multivalued Dependencies

A multivalued dependency (MVD) occurs when one attribute in a relation determines a set of values for another attribute independently of other attributes.
It is denoted as:

$X \twoheadrightarrow Y$

This means that for each value of X, there exists a **set of independent values** of Y, unrelated to other attributes in the relation.
An MVD is **trivial** if:

- $Y \subseteq X$, or
- $X \cup Y = R$.

Otherwise, it is **non-trivial**.

14.2.2 Example of MVD

Consider the relation:

COURSE(Course_ID, Textbook, Instructor)

- A course may be taught by several instructors.
- The same course may also have multiple textbooks.

Hence, for a given Course_ID, the attributes Textbook and Instructor are independent multivalued attributes.

Thus:

$Course\_ID \twoheadrightarrow Instructor$

and

$Course\_ID \twoheadrightarrow Textbook$

If we store all combinations in a single relation, redundancy appears:

Course_ID	Instructor	Textbook
C101	Dr. Meena	DBMS
C101	Dr. Meena	SQL
C101	Dr. Ramesh	DBMS
C101	Dr. Ramesh	SQL

Each instructor–textbook pair repeats unnecessarily.

14.2.3 Decomposition into Fourth Normal Form

To remove this redundancy, we decompose the relation based on MVDs:
COURSE_INSTRUCTOR(Course_ID, Instructor)
COURSE_TEXTBOOK(Course_ID, Textbook)
Each now represents an independent multivalued relationship.

When these relations are joined on Course_ID, we can reconstruct the original data without spurious tuples.

14.2.4 Definition of Fourth Normal Form (4NF)

A relation is in Fourth Normal Form (4NF) if it is in Boyce-Codd Normal Form (BCNF) and contains no non-trivial multivalued dependencies. Decomposing into 4NF involves removing MVDs by creating separate relations.

Achieving 4NF

To achieve 4NF, decompose the relation to eliminate MVDs while ensuring that the decomposition maintains the lossless join property.
A relation R is in 4NF if and only if:

- It is in Boyce–Codd Normal Form (BCNF), and
- For every non-trivial multivalued dependency $X \twoheadrightarrow Y$ in R, X is a superkey of R.

Thus, 4NF eliminates redundancies arising from independent multivalued dependencies, ensuring that data is not duplicated across unrelated attributes.

14.2.5 Example Summary

Before 4NF:
COURSE(Course_ID, Instructor, Textbook)
After 4NF decomposition:
COURSE_INSTRUCTOR(Course_ID, Instructor)
COURSE_TEXTBOOK(Course_ID, Textbook)

- Redundancy removed
- Lossless join maintained
- Multivalued dependencies represented separately

Example:
Given the relation R(Student, Course, Hobby) with MVDs Student→→Course and Student→→Hobby :

1. **Original Relation:**

Student	Course	Hobby
John	Math	Reading
John	Math	Swimming
John	Science	Reading
John	Science	Swimming

2. Decompose into Two Relations:

- $R1(Student, Course)$:

Student	Course
John	Math
John	Science

- $R2(Student, Hobby)$:

Student	Hobby
John	Reading
John	Swimming

Fig 14.1 Result of 4NF

These decomposed relations are now in 4NF, eliminating the multivalued dependencies and ensuring that each attribute is independently associated with the key attribute.
Multivalued dependencies highlight situations where one attribute determines a set of values independently of others. Fourth Normal Form (4NF) addresses these dependencies, further refining the database schema to eliminate redundancy and improve data integrity. By decomposing relations to remove MVDs, 4NF ensures a more robust and efficient database design.

14.3 JOIN DEPENDENCIES AND FIFTH NORMAL FORM

14.3.1 Concept of Join Dependency

A join dependency (JD) specifies that a relation can be reconstructed by joining several projections of the relation. It is a generalization of functional and multivalued dependencies.
A **join dependency (JD)** generalizes both **functional** and **multivalued dependencies**. It specifies a constraint that a relation **R** can be **reconstructed by joining multiple projections** of itself.

Formally, a **join dependency** $JD(R_1, R_2, \ldots, R_n)$ holds on relation **R** if:

$R = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

That is, the join of these projections yields exactly **R**, with no extra or missing tuples.

Every **multivalued dependency** is a special case of join dependency where $n = 2$.

14.3.2 Fifth Normal Form (5NF)

A relation is in Fifth Normal Form (5NF) if it is in 4NF and contains no non-trivial join dependencies. Decomposing into 5NF involves breaking down the relation into smaller relations that can be joined without loss of information.
A relation R is in Fifth Normal Form (5NF), also called Project-Join Normal Form (PJNF), if:

- It is in 4NF, and
- Every non-trivial join dependency in R is implied by the candidate keys of R.5NF ensures that a relation cannot be decomposed further without losing information or introducing spurious tuples.

14.3.3 Example of 5NF

Consider the relation:
SUPPLY(Supplier, Part, Project)
Interpretation:

- A supplier supplies certain parts.
- Each part is used in certain projects.

If these three facts are independent, redundancy appears because each combination of (Supplier, Part, Project) repeats values unnecessarily.

Supplier	Part	Project
S1	P1	J1
S1	P2	J1
S1	P1	J2

This redundancy can be eliminated by decomposing into:
SUPPLIER_PART(Supplier, Part)
SUPPLIER_PROJECT(Supplier, Project)
PART_PROJECT(Part, Project)
Joining these three relations reconstructs the original data — a lossless join satisfying 5NF.

Example
Given the relation R(A,B,C) with a join dependency:

1. Original Relation:

A	B	C
1	X	10
2	Y	20
1	X	20
2	Y	10

These decomposed relations are now in 5NF, eliminating the join dependencies and ensuring that the original relation can be reconstructed without loss of information. Join dependencies are a powerful concept in relational database theory, allowing for the reconstruction of a relation from its projections. Fifth Normal Form (5NF) addresses these dependencies, ensuring that the database schema is fully normalized, with no redundant data and all join dependencies preserved. Achieving 5NF guarantees the most refined and efficient database design, capable of handling complex data relationships with minimal redundancy and maximum data integrity.

2. **Decompose into Three Relations:**

- $R1(A, B)$:

A	B
1	X
2	Y

- $R2(B, C)$:

B	C
X	10
Y	20
X	20
Y	10

- $R3(A, C)$:

A	C
1	10
2	20
1	20
2	10

Fig 14.2 Result of 5NF

14.3.4 Significance of 5NF

5NF removes redundancies caused by complex **join dependencies** that cannot be expressed as simple functional or multivalued dependencies.
It ensures the database schema represents **independent facts only once**, which is vital for large-scale distributed and analytical systems.

14.4 Inclusion Dependencies

Inclusion Dependencies (INDs) are constraints in a relational database that ensure values in certain columns (or sets of columns) of one relation must also appear in certain columns of another relation. This concept is fundamental in enforcing referential integrity, typically implemented through foreign key constraints.

14.4.1 Definition

An **inclusion dependency (IND)** specifies that a set of values in one relation must appear as values in another relation.
It is denoted as:

$$R_1[X] \subseteq R_2[Y]$$

Meaning: The set of values of attribute(s) **X** in relation $\mathbf{R_1}$ must exist as values of attribute(s) **Y** in relation $\mathbf{R_2}$.
Inclusion Dependencies can be formally defined as follows: Given two relations R1and R2, an inclusion dependency specifies that a set of attributes A in R1 must match a set of attributes B in R2. This is denoted as R1[A]⊆R2[B].

Types of Inclusion Dependencies:

1. **Simple Inclusion Dependencies**: The dependency involves a single attribute or a simple set of attributes.
   - **Example**: The foreign key constraint where the DepartmentID in an Employees table must match the DepartmentID in a Departments table.
2. **Compound Inclusion Dependencies**: The dependency involves a compound set of attributes.
   - **Example**: A dependency involving a combination of attributes such as (EmployeeID, ProjectID) in an Assignments table matching (EmployeeID, ProjectID) in a Projects table.

14.4.2 Example

In a university database:
STUDENT(Student_ID, Name, Dept_ID)
DEPARTMENT(Dept_ID, Dept_Name)

We define an inclusion dependency:

$$STUDENT[Dept\_ID] \subseteq DEPARTMENT[Dept\_ID]$$

This ensures that every student's department exists in the DEPARTMENT table — a form of referential integrity constraint.

Example: In a Students and Enrollments schema, ensuring that every StudentID in the Enrollments table appears in the Students table.

```
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   StudentName VARCHAR(100)
);
```

```
CREATE TABLE Enrollments (
   EnrollmentID INT PRIMARY KEY,
   StudentID INT,
   CourseID INT,
   FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

Data Consistency Across Relations:
Example: Ensuring that all product IDs in an OrderDetails table exist in a Products table.

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

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

14.4.3 Role of Inclusion Dependencies

- Enforce **foreign key constraints**.
- Maintain **consistency across relations**.
- Prevent insertion of invalid or orphan references.
- Essential for **distributed databases**, where relations may span multiple sites.

Enforcing inclusion dependencies involves defining foreign keys and other constraints to maintain consistency between related tables.

Example of Enforcing Inclusion Dependencies:

Departments:

DepartmentID	DepartmentName
1	HR
2	IT

Employees:

EmployeeID	EmployeeName	DepartmentID
101	Alice	1
102	Bob	2

The foreign key constraint ensures that every DepartmentID in the Employees table must match a DepartmentID in the Departments table:

```
CREATE TABLE Departments (
   DepartmentID INT PRIMARY KEY,
   DepartmentName VARCHAR(100)
);
```

```
CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   EmployeeName VARCHAR(100),
   DepartmentID INT,
   FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

Inclusion Dependencies are crucial for maintaining referential integrity and consistency in relational databases. By ensuring that values in certain columns of one relation must appear in columns of another relation, INDs help prevent data anomalies and enforce relationships between different entities in the database schema.

14.5 OTHER DEPENDENCIES AND NORMAL FORMS

Beyond functional, multivalued, and join dependencies, there are other forms of constraints that play roles in advanced schema design.

14.5.1 Domain-Key Normal Form (DKNF)

A relation is in Domain-Key Normal Form if it meets all domain constraints and key constraints, ensuring that all possible constraints are captured by domain and key dependencies.
A relation is in **Domain-Key Normal Form (DKNF)** if **all constraints** on the relation are **a logical consequence** of:

- **Domain constraints** (allowable values for each attribute), and
- **Key constraints** (uniqueness of primary keys).

In DKNF, all anomalies are theoretically removed. However, it is difficult to achieve in practice because defining every constraint explicitly is complex.

Achieving DKNF
To achieve DKNF, a relation must be carefully designed to ensure that all constraints are captured through domain and key constraints. This often involves:

1. **Eliminating All Non-Domain, Non-Key Constraints**: Ensure that there are no constraints other than those imposed by domains and keys.
2. **Redesigning Schema**: If necessary, redesign the schema to incorporate all constraints into the domains and keys.

Benefits

- **Eliminates All Anomalies**: By only having domain and key constraints, the relation is free from update, insertion, and deletion anomalies.
- **Simplifies Constraint Management**: Constraints are easier to understand, enforce, and manage since they are limited to domains and keys.

Domain-Key Normal Form (DKNF) represents the highest level of normalization, ensuring that a database schema is free from all possible anomalies by relying solely on domain and key constraints. Achieving DKNF involves designing the schema in such a way that all necessary restrictions on data are captured by the permissible values of attributes and the uniqueness of tuples, resulting in a highly robust and reliable database structure.

14.5.2 Sixth Normal Form (6NF)

6NF is a rarely used, specialized normal form designed for temporal and data warehousing applications.
A relation is in 6NF if it is in 5NF and cannot be decomposed further without losing information.
Used mainly in time-dependent data systems where attributes vary independently over time.

Example:
EMP_SALARY(Emp_ID, Salary, Effective_Date)
Each change in salary forms a new tuple — representing data evolution over time.

14.5.3 Summary of Normal Forms

Normal Form	Based On	Removes	Key Concept
1NF	Atomic values	Repeating groups	Single-valued attributes
2NF	Functional dependency	Partial dependency	Full functional dependency
3NF	Functional dependency	Transitive dependency	Non-key → Non-key dependency
BCNF	Functional dependency	Overlapping candidate keys	Every determinant is key
4NF	Multivalued dependency	Redundant multivalued data	MVDs → Superkey
5NF	Join dependency	Redundant joins	Lossless join from projections
DKNF	All constraints	All anomalies	Domain + Key rules
6NF	Temporal / advanced	Non-decomposable	Time-varying data

14.6 Summary

This lesson expanded normalization beyond functional dependencies into multivalued and join dependencies, leading to the Fourth and Fifth Normal Forms.
A relation in 4NF eliminates redundancy caused by independent multivalued attributes, while 5NF ensures no redundant data exists even under complex join conditions.
Additionally, inclusion dependencies and higher forms such as DKNF and 6NF were introduced to handle inter-table constraints and temporal data.
These advanced dependencies provide a theoretical foundation for creating highly reliable, semantically accurate, and redundancy-free databases suitable for enterprise and analytical systems.

14.7 Technical Terms

1. Multivalued Dependency (MVD)
2. Join Dependency (JD)
3. Fourth Normal Form (4NF)
4. Fifth Normal Form (5NF)
5. Inclusion Dependency (IND)
6. Domain-Key Normal Form (DKNF)

7. Sixth Normal Form (6NF)
8. Lossless Join
9. Referential Integrity
10. Projection-Join Normal Form (PJNF)

14.8 Self-Assessment Questions

Essay Questions

1. Define Multivalued Dependency (MVD). Explain its role in Fourth Normal Form (4NF) with an example.
2. Discuss Join Dependencies and describe how Fifth Normal Form (5NF) is achieved.
3. Explain the importance of Inclusion Dependencies in relational database design.
4. Differentiate between BCNF, 4NF, and 5NF with examples.
5. What is Domain-Key Normal Form (DKNF)? Explain its advantages and limitations.

Short Questions

1. Write the notation used for multivalued dependency.
2. Define trivial and non-trivial MVDs.
3. What is a join dependency?
4. Define Fifth Normal Form (5NF).
5. What does inclusion dependency ensure?
6. What is referential integrity?
7. Mention one example of a 6NF relation.
8. What is lossless decomposition?
9. Define Domain Constraint.
10. What is the practical use of 4NF in database design?

14.9 Suggested Readings

1. Ramez Elmasri & Shamkant B. Navathe, *Fundamentals of Database Systems*, Pearson Education.
2. C.J. Date, *An Introduction to Database Systems*, Addison-Wesley.
3. Abraham Silberschatz, Henry F. Korth & S. Sudarshan, *Database System Concepts*, McGraw Hill.
4. Raghu Ramakrishnan & Johannes Gehrke, *Database Management Systems*, McGraw Hill.
5. Thomas Connolly & Carolyn Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*, Pearson.
6. Hector Garcia-Molina, Jeffrey Ullman & Jennifer Widom, *Database Systems: The Complete Book*, Pearson.
7. Peter Rob & Carlos Coronel, *Database Systems: Design, Implementation, and Management*, Cengage Learning.
8. Alexis Leon & Mathews Leon, *Database Management Systems*, Vikas Publishing.

Dr. U. Surya Kameswari

LESSON- 15

DOCUMENT ORIENTED DATA

AIMS AND OBJECTIVES

The primary aim of this lesson is to provide a comprehensive understanding of document-oriented data models and the principles underlying their schema design, focusing on how these differ from traditional relational database structures. The lesson aims to develop the learner's ability to design efficient, scalable, and flexible document-based schemas for real-world applications such as e-commerce platforms.

After completing this lesson, learners will be able to:

- Understand the fundamental concepts of document-oriented databases and their advantages over relational systems.
- Explain the principles of schema design specific to document-based data models, including embedding, referencing, and denormalization.
- Design and model data for an e-commerce application using document-oriented approaches.
- Differentiate between databases, collections, and documents, and explain their roles in data organization.
- Apply indexing, validation, and optimization techniques to improve query performance and maintain data integrity.
- Evaluate trade-offs between flexibility, redundancy, and consistency in document-oriented systems.
- Implement best practices for scalability, schema evolution, and data security in document databases.
- Compare and contrast document-oriented schema design with relational schema design principles.

STRUCTURE:

15.1 INTRODUCTION

The rapid growth of unstructured and semi-structured data has led to the emergence of NoSQL (Not Only SQL) databases, which prioritize flexibility, scalability, and performance. Among the four major types of NoSQL systems—key-value stores, column stores, graph databases, and document-oriented databases—the document-oriented model has gained significant popularity for web-scale applications.

15.1.1 What Are Document-Oriented Databases?

A document-oriented database stores data as documents, usually in JSON (JavaScript Object Notation) or BSON (Binary JSON) format. Each document is a self-describing unit containing field-value pairs, arrays, and nested objects.

Example:

```
{
  "name": "Lavanya",
  "email": "lavs@example.com",
  "skills": ["Python", "MongoDB", "Data Modeling"],
  "address": {"city": "Guntur", "state": "Andhra Pradesh"}
}
```

Unlike relational rows and columns, this representation supports hierarchical and flexible structures, allowing different documents to have varying fields.

15.1.2 Characteristics of Document Databases

- Schema flexibility: Fields can differ between documents.
- Hierarchical structure: Supports nesting and arrays.
- Indexing support: Enables fast query execution.
- Horizontal scalability: Uses sharding for distributed storage.
- Ease of evolution: Fields can be added or removed without schema migration.

15.1.3 Popular Document Databases

Database	Description
MongoDB	Open-source, BSON-based, highly scalable document store.
CouchDB	JSON-based database using HTTP/REST API for access.
Firebase Firestore	Cloud-hosted document database for mobile/web apps.
RavenDB	ACID-compliant document store optimized for .NET applications.

1. MongoDB

Description:

MongoDB is an open-source, NoSQL document-oriented database developed by MongoDB Inc. It stores data in BSON (Binary JSON) format, which supports richer data types (e.g., dates, binary data, and nested arrays). MongoDB is widely recognized as the most popular document database due to its simplicity, scalability, and community support.

Key Features:

- Dynamic schema: No fixed table structure—documents in the same collection can vary in fields.
- Rich query language: Supports filters, projections, aggregations, and text search.
- Indexing and Aggregation Framework: Allows creation of multiple index types and complex data analysis pipelines.
- Sharding and Replication: Enables horizontal scaling and high availability.
- Driver Support: Available for all major programming languages including Python, Java, and C#.

Architecture Overview:

MongoDB uses a client–server architecture, where data is stored in collections within databases. It employs replica sets for fault tolerance and sharding for distributed data storage.

Advantages:

- High scalability and performance.
- Easy integration with application frameworks (e.g., Node.js, Django).
- Powerful aggregation pipeline for analytics.
- Widely supported and documented.

Typical Use Cases:

- E-commerce platforms (product catalogs, orders).
- Content management systems (CMS).
- IoT applications collecting varied sensor data.
- Real-time analytics dashboards.

2. CouchDB

CouchDB, developed by the Apache Software Foundation, is a JSON-based, schema-free document database that uses the HTTP/REST protocol for data access and manipulation. Each document is uniquely identified by an ID and revision number, making it ideal for synchronization and offline-first applications.

Key Features:

- RESTful API: Data is accessed and modified through HTTP requests (GET, PUT, POST, DELETE).
- MVCC (Multi-Version Concurrency Control): Prevents conflicts without locking documents.
- Replication and Synchronization: Data can be easily replicated across nodes or devices.
- MapReduce Views: Query and aggregation mechanism using JavaScript functions.
- Fault-tolerant and crash-only design: Ensures durability of data.

Architecture Overview:

CouchDB uses a single-node or clustered architecture, depending on deployment **needs.** Data is stored in documents that include metadata and revision information, allowing easy conflict resolution during replication.

Advantages:
- Perfect for offline-first mobile applications.
- Simple, human-readable data access via REST API.
- Strong data consistency with revision control.
- Excellent for distributed and replicated environments.

Typical Use Cases:
- Offline mobile apps with synchronization (e.g., field data collection).
- Web apps requiring replication across servers or regions.
- Document versioning systems.
- Configuration storage for distributed systems.

3. Firebase Firestore

Firebase Cloud Firestore, developed by Google, is a serverless, cloud-hosted document database that is part of the Firebase platform. It is designed for mobile and web applications, providing real-time synchronization and automatic scaling.

Key Features:
- Real-time data synchronization: Updates propagate instantly across connected clients.
- Hierarchical data structure: Organizes documents in collections and subcollections.
- Serverless environment: No infrastructure management required.
- Offline data persistence: Supports caching for mobile devices.
- Strong integration with Firebase Authentication and Cloud Functions.

Architecture Overview:

Firestore is a fully managed cloud service where data is replicated across multiple Google Cloud regions for reliability and low latency. Developers interact with it using SDKs or REST APIs.

Advantages:
- Real-time synchronization between web and mobile clients.
- Automatic scaling and global availability.
- Built-in security through Firebase rules and Google IAM.
- Tight integration with analytics, authentication, and hosting services.

Typical Use Cases:
- Real-time chat or messaging applications.
- Collaborative tools (e.g., shared whiteboards, note-taking apps).
- Mobile gaming backends.
- Event tracking and analytics storage.

4. RavenDB

RavenDB is a fully ACID-compliant, open-source document-oriented database written in C#, optimized for the .NET ecosystem. It combines the benefits of a document store with strong transactional guarantees, offering an ideal balance between performance and consistency.

Key Features:

- ACID Transactions: Ensures atomicity and consistency across multiple documents.
- Multi-Document Queries: Supports complex joins, projections, and full-text search.
- Integrated ETL and Indexing Engine: Automatic indexing and data export to SQL or other stores.
- Built-in GUI (Studio): Provides a visual management interface.
- High Performance: Uses an internal storage engine optimized for SSDs.

Architecture Overview:

RavenDB supports both on-premises and cloud-based deployments. It uses safe-by-default design, meaning all operations are transactional and secure. It provides cluster replication, ensuring high availability.

Advantages:

- Full ACID compliance ensures strong consistency.
- Simple setup and robust management tools.
- Optimized for .NET developers using C# or ASP.NET Core.
- Supports distributed clusters with automatic failover.

Typical Use Cases:

- Enterprise-grade business applications requiring data consistency.
- Financial and transactional systems.
- Document management platforms.
- API backends for .NET-based systems.

15.1.5 Comparative Summary

Feature	MongoDB	CouchDB	Firebase Firestore	RavenDB
Data Format	BSON (Binary JSON)	JSON	JSON	JSON
Query Interface	Rich query language & aggregation framework	REST API (HTTP-based)	SDKs & REST API	LINQ & REST
Transactions	Multi-document (since v4.0)	MVCC revision control	ACID at document level	Full ACID support
Scalability	High (sharding, replication)	Horizontal replication	Auto-scaling (serverless)	Cluster replication
Use Case Focus	Web apps, analytics, e-commerce	Offline sync, distributed apps	Real-time mobile/web apps	Enterprise & financial apps
Hosting Model	Self-hosted or managed Atlas	Self-hosted	Cloud-managed	Self-hosted or cloud
Developer Ecosystem	Multi-language drivers	REST-based integration	Tight Firebase SDK integration	Optimized for .NET stack

15.2 Principles of Schema Design in Document Databases

Schema design determines how data is structured, stored, and retrieved. In document databases, schema design is use-case driven rather than normalized as in relational systems.

15.2.1 Schema Flexibility

- Document databases allow documents within the same collection to have different structures.
- This flexibility supports iterative development and agile data modeling.

Example:

```
// Document 1
{"name": "Alice", "email": "alice@example.com"}

// Document 2
{"name": "Bob", "email": "bob@example.com", "phone": "9876543210"}
```

Both documents coexist in the same collection, even though the second has an additional field.

15.2.2 Modeling Philosophy

Relational design focuses on normalization; document design focuses on application query patterns.

Questions to consider before designing:

- What data is accessed together most frequently?
- Which queries are most common?
- How frequently does data change?

The goal is to reduce joins and retrieve complete data in a single query.

15.2.3 Embedding vs. Referencing

Pattern	Description	When to Use
Embedding	Store related data within a single document.	When related data is small and accessed together.
Referencing	Use identifiers to link separate documents.	When data is reused or large in size.

Example (Embedding):

```
{
  "student_id": "S001",
  "name": "Anita",
  "subjects": [
    {"code": "CS101", "name": "Data Structures"},
    {"code": "CS102", "name": "DBMS"}
  ]
}
```

Example (Referencing):

```
{
  "student_id": "S001",
  "name": "Anita",
  "subject_ids": ["CS101", "CS102"]
}
```

15.2.4 Denormalization

Denormalization is intentional data duplication to improve query performance. In document databases, it is preferred for read-heavy applications.

Example:

Instead of joining Product and Category collections, store category information directly inside each product document.

```
{
  "product_id": "P101",
  "name": "Bluetooth Speaker",
  "category": {"id": "C10", "name": "Electronics"}
}
```

15.2.5 Schema Validation

Modern databases like MongoDB support JSON schema validation:

```
db.createCollection("users", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "email"],
      properties: {
        email: { bsonType: "string", pattern: "^.+@.+$" }
      }
    }
  }
})
```

This enforces data consistency while preserving flexibility.

15.2.6 Indexing Strategies

Indexes improve query performance. Common types:

- Single-field index (e.g., name)
- Compound index (e.g., category + price)
- Text index (for searching descriptions)
- Geospatial index (for location-based data)

Example:

```
db.products.createIndex({ category: 1, price: -1 })
```

15.2.7 Trade-offs in Schema Design

Concern	Impact
Flexibility	Increases development agility
Duplication	Speeds reads, increases update complexity
Referencing	Saves space, adds query overhead
Embedding	Fast reads, slower updates if nested data changes

15.3 Designing an E-Commerce Data Model

E-commerce systems handle large-scale, interconnected data—making them ideal for document modeling.

15.3.1 Key Entities

- Users – customer details, addresses, preferences
- Products – item catalog, categories, specifications
- Orders – transaction records
- Carts – temporary order details
- Reviews – feedback and ratings
- Payments – transaction metadata

15.3.2 Sample Collections and Documents

Users Collection

```
{
 "_id": "U001",
 "name": "Lavanya",
 "email": "lavs@example.com",
 "addresses": [
   {"type": "Home", "city": "Guntur"},
   {"type": "Work", "city": "Vijayawada"}
 ]
}
```

Products Collection

```
{
 "_id": "P101",
 "name": "Wireless Mouse",
 "category": "Electronics",
 "price": 499,
 "specs": {"brand": "Logitech", "color": "Black"},
 "stock": 100
}
```

Orders Collection

```
{
 "_id": "O9001",
 "user_id": "U001",
 "order_date": "2025-11-01",
```

```
  "items": [
    {"product_id": "P101", "quantity": 2, "price": 499}
  ],
  "total_amount": 998,
  "status": "Paid"
}
```

Reviews Collection

```
{
  "_id": "R2001",
  "user_id": "U001",
  "product_id": "P101",
  "rating": 5,
  "comment": "Excellent quality!"
}
```

15.3.3 Query Examples

Find all paid orders of a user:

```
db.orders.find({ user_id: "U001", status: "Paid" })
```

List products below ₹1000 in Electronics:

```
db.products.find({ category: "Electronics", price: { $lt: 1000 } })
```

Aggregate total sales by category:

```
db.orders.aggregate([
  { $unwind: "$items" },
  { $lookup: { from: "products", localField: "items.product_id", foreignField: "_id", as: "product_info" } },
  { $group: { _id: "$product_info.category", totalSales: { $sum: "$items.price" } } }
])
```

15.3.4 Performance and Scalability

- Sharding: Distribute data across multiple servers.
- Index frequently queried fields: e.g., user_id, category.
- Use caching layers: Redis or in-memory caching for hot data.

15.4 Databases, Collections, and Documents

15.4.1 Databases

Logical containers for collections.

Example:

```
use EcommerceDB
```

15.4.2 Collections

Equivalent to tables but schema-less.

Example:

```
db.createCollection("products")
```

15.4.3 Documents

Atomic units of storage, stored as JSON/BSON objects.

Example:

```
{
  "name": "Lavanya",
  "cart": [
    {"product": "Mouse", "qty": 2},
    {"product": "Keyboard", "qty": 1}
  ]
}
```

15.4.4 CRUD Operations

Operation	Command	Description
Create	insertOne()	Add new document
Read	find()	Retrieve documents
Update	updateOne()	Modify document
Delete	deleteOne()	Remove document

15.4.5 Aggregation Framework

Used for analytics:

```
db.orders.aggregate([
  { $match: { status: "Paid" } },
  { $group: { _id: "$user_id", totalSpent: { $sum: "$total_amount" } } }
])
```

15.5 Other Design Considerations and Best Practices

1. Design for Read Efficiency – Embed data for frequent reads.
2. Maintain Data Integrity – Use JSON validation and atomic updates.
3. Index Appropriately – Avoid over-indexing.
4. Plan for Schema Evolution – Handle version changes in code.
5. Implement Access Control – Use RBAC and field-level encryption.
6. Monitor and Optimize – Regularly review performance metrics.

15.6 Summary

Document-oriented databases provide a flexible, scalable alternative to relational models. This lesson discussed:

- Schema design principles (embedding, referencing, denormalization)
- Modeling an e-commerce application
- Database, collection, and document structures
- Performance and design best practices

15.7 Technical Terms

1. Document Database
2. Collection
3. Embedding
4. Referencing
5. Denormalization
6. Sharding
7. Indexing
8. BSON

15.8 Self-Assessment Questions

Short Answer:

1. Define document-oriented databases.
2. What are the advantages of schema flexibility?
3. Differentiate between embedding and referencing.
4. Explain denormalization with an example.
5. What is the purpose of sharding?

Long Answer:

1. Explain the principles of schema design in document databases.
2. Design and describe an e-commerce schema using document-oriented techniques.
3. Discuss the roles of databases, collections, and documents.
4. Compare document-oriented and relational models.
5. Discuss schema validation and indexing strategies in MongoDB.

15.9 Suggested Readings

1. Kristina Chodorow – *MongoDB: The Definitive Guide*, O'Reilly Media.
2. Rick Copeland – *MongoDB Applied Design Patterns*, O'Reilly Media.
3. Ramez Elmasri & Shamkant Navathe – *Fundamentals of Database Systems*, Pearson Education.
4. Couchbase Documentation – *Schema Design Principles*.
5. MongoDB University – *Data Modeling Fundamentals*.

Dr. U. Surya Kameswari

LESSON- 16
QUERIES AND AGGREGATIONE-COMMERCE'S

AIMS AND OBJECTIVES:

Aim

The aim of this lesson is to provide a comprehensive understanding of how queries and aggregation operations are performed in document-oriented databases, particularly MongoDB, within the context of e-commerce applications.

This lesson focuses on teaching learners how to efficiently retrieve, filter, and summarize data from large datasets using MongoDB's Query Language (MQL) and the Aggregation Framework, enabling effective analytics and reporting.

After completing this lesson, learners will be able to:

1. Understand the importance of querying and aggregation in document-oriented databases.
2. Describe the MongoDB Query Language (MQL) and its syntax for data retrieval.
3. Execute various query operations to filter, sort, and project data in e-commerce contexts.
4. Apply conditional operators such as $and, $or, $in, $gt, $lt, and regular expressions in queries.
5. Explain the concept and structure of the Aggregation Framework in MongoDB.
6. Design and implement aggregation pipelines for computing totals, averages, counts, and grouped data.
7. Perform real-world e-commerce analytics, such as aggregating total sales, identifying top-selling products, and summarizing customer activity.
8. Differentiate between queries and aggregations, and understand when to use each.
9. Optimize query and aggregation performance using indexing and pipeline optimization techniques.

STRUCTURE:

16.1.1 OVERVIEW

16.1 INTRODUCTION

16.1.1 Overview

In modern data-driven applications such as **e-commerce systems**, the ability to **query and aggregate information** efficiently is vital for operational performance and business intelligence.

While document-oriented databases like **MongoDB** provide flexible data storage, the real power lies in their ability to perform complex **queries** and **aggregations** on massive datasets without the need for predefined schemas or complex joins.

Queries allow developers to **retrieve specific pieces of information**, such as all orders placed by a customer or all products under a certain price range, while aggregation enables the **summarization and analysis of data**, such as calculating total sales, average ratings, or daily revenue.

This lesson introduces the principles of querying and aggregating data in document-oriented databases, focusing particularly on **MongoDB's Query Language (MQL)** and **Aggregation Framework** within an **e-commerce context**.

The rapid growth of unstructured and semi-structured data has led to the emergence of NoSQL (Not Only SQL) databases, which prioritize flexibility, scalability, and performance. Among the four major types of NoSQL systems—key-value stores, column stores, graph databases, and document-oriented databases—the document-oriented model has gained significant popularity for web-scale applications.

16.1.2 Importance of Queries and Aggregation in E-Commerce

E-commerce platforms deal with large volumes of data generated from multiple entities — **users, products, orders, reviews, and payments**.

To manage and analyze this data effectively, two key operations are essential:

- **Queries** – for fetching, filtering, and sorting specific data based on user-defined conditions.
- **Aggregations** – for summarizing and computing statistics across large datasets.
- For example:
- Querying all **pending orders** placed by a specific user.
- Aggregating **total sales revenue** for a particular product category.
- Computing the **average product rating** from customer reviews.
- Together, these operations enable **data-driven insights**, **performance monitoring**, and **strategic decision-making** for e-commerce businesses.

16.1.3 Querying in Document Databases

In document-oriented systems such as MongoDB, data is stored as **JSON-like documents**, which can be queried using **field-value pairs** and **operators**.
Unlike SQL queries, which rely on predefined tables and joins, MongoDB queries operate directly on nested and hierarchical document structures.

A basic MongoDB query uses the find() method:

db.products.find({ category: "Electronics", price: { $lt: 1000 } })
This retrieves all documents from the products collection where the category is "Electronics" and the price is less than ₹1000.

MongoDB also supports advanced query features such as:

- Comparison operators ($gt, $lt, $eq)
- Logical operators ($and, $or)
- Array operators ($in, $all)
- Regular expressions and text search

This makes querying in MongoDB flexible, intuitive, and powerful for large datasets.

16.1.4 Aggregation in Document Databases

While queries focus on retrieving individual documents, **aggregation** is used for **grouping and summarizing** data.

MongoDB's **Aggregation Framework** allows performing complex analytical operations such as counting, averaging, summing, and grouping data — similar to SQL's GROUP BY and aggregate functions but more flexible and scalable.

Example:
To compute total sales for each product category:

```
db.orders.aggregate([
  { $unwind: "$items" },
  { $group: { _id: "$items.category", totalSales: { $sum: "$items.price" } } }
])
```

This aggregation pipeline performs the following:

- **$unwind** – breaks the array of items into individual documents.
- **$group** – groups data by category and calculates total sales per category.

Aggregation thus enables **data transformation**, **reporting**, and **real-time analytics** directly within the database layer.

16.1.5 Query Optimization and Indexing

Efficient query performance is essential in large-scale e-commerce systems. MongoDB uses **indexes** to accelerate query execution by avoiding full collection scans. Common index types include:

- **Single-field indexes** – for frequent lookups on one field.
- **Compound indexes** – for filtering on multiple fields (e.g., category + price).
- **Text indexes** – for keyword-based product searches.
- **Geospatial indexes** – for location-based queries (e.g., nearby stores or delivery zones).

Well-designed indexes can drastically improve performance and scalability, especially when combined with properly structured queries and aggregation pipelines.

16.1.6 Real-World Example: E-Commerce Query and Aggregation Use Cases

Use Case 1: Retrieve all orders placed by a specific customer in the last 30 days.
Use Case 2: Calculate total revenue per product category.
Use Case 3: Find the top 10 best-selling products by sales volume.
Use Case 4: Determine the average rating for each product.
Use Case 5: Generate a monthly report of new customers and total transactions.
These examples demonstrate how queries and aggregations power various **operational** and **analytical** functions of modern e-commerce systems — from recommendation engines to performance dashboards.

16.2 QUERYING IN E-COMMERCE APPLICATIONS

16.2.1 Overview of Querying in E-Commerce

In an **e-commerce system**, querying enables the retrieval of data from various collections such as users, products, orders, and reviews.

Effective queries ensure that customers can **search products**, **track orders**, and **view personalized recommendations** quickly.

In document-oriented databases like **MongoDB**, querying is performed using the **MongoDB Query Language (MQL)**, which uses a **JSON-like syntax** to filter and manipulate data. Queries are executed through methods like find(), findOne(), and aggregate().

Example basic query:

```
db.products.find({ category: "Electronics" })
```

This retrieves all documents from the products collection where the category is **Electronics**.

16.2.2 Common Query Scenarios in E-Commerce

E-commerce data can be vast and dynamic. Some common query operations include:

- Fetching **user information** for authentication or profile display.
- Searching **products** by category, price range, or brand.
- Listing **orders** placed by a customer.
- Filtering **reviews** for a product.
- Finding **top-selling** or **newly added** products.

Each of these operations relies on structured query design using appropriate MongoDB operators.

16.2.3 Querying Customer Information

Customer data may be stored as:

```
{
  "_id": "U101",
  "name": "Lavanya",
  "email": "lavs@example.com",
  "city": "Guntur",
  "loyalty_points": 1200
}
```

To find a customer by email:
db.users.find({ email: "lavs@example.com" })
To find all customers with more than 1000 loyalty points:
db.users.find({ loyalty_points: { $gt: 1000 } })
These queries are straightforward, readable, and efficient when indexes are applied to frequently queried fields like email or city.

16.2.4 Querying Products

Product data is the heart of any e-commerce platform. A typical product document might look like:

```
{
  "_id": "P101",
  "name": "Wireless Mouse",
  "category": "Electronics",
  "brand": "Logitech",
  "price": 499,
  "rating": 4.5
}
```

Examples of product queries:

- Fetch all products under ₹1000:
- db.products.find({ price: { $lt: 1000 } })
- Find all products in the "Electronics" category with a rating above 4:
- db.products.find({ category: "Electronics", rating: { $gt: 4 } })
- Retrieve specific fields (projection):
- db.products.find({ category: "Electronics" }, { name: 1, price: 1, _id: 0 })

16.2.5 Querying Orders

Order data often includes nested documents and arrays:

```
{
  "_id": "O9001",
  "user_id": "U101",
  "order_date": "2025-10-28",
  "status": "Shipped",
  "items": [
    { "product_id": "P101", "quantity": 2, "price": 499 },
    { "product_id": "P102", "quantity": 1, "price": 899 }
  ]
}
```

Examples:

- Retrieve all orders for a specific user:
- db.orders.find({ user_id: "U101" })
- Find all "Shipped" orders:
- db.orders.find({ status: "Shipped" })
- Retrieve all orders placed after a certain date:
- db.orders.find({ order_date: { $gte: "2025-10-01" } })

MongoDB can efficiently query even deeply nested fields using **dot notation**, e.g., items.product_id.

16.2.6 Querying Reviews

Customer feedback and reviews are stored separately in a reviews collection:

```
{
  "_id": "R301",
  "product_id": "P101",
  "user_id": "U101",
  "rating": 5,
  "comment": "Excellent product!"
}
```

Examples:

- Fetch all reviews for a specific product:
- db.reviews.find({ product_id: "P101" })
- Find reviews with rating less than 3 (to analyze negative feedback):
- db.reviews.find({ rating: { $lt: 3 } })

16.2.7 Using Comparison and Logical Operators

MongoDB provides rich query operators for filtering data precisely.

Operator	Description	Example
$gt	Greater than	{ price: { $gt: 1000 } }
$lt	Less than	{ rating: { $lt: 4 } }
$and	Combine multiple conditions	{ $and: [{ category: "Electronics" }, { price: { $lt: 2000 } }] }
$or	Match any condition	{ $or: [{ category: "Books" }, { category: "Stationery" }] }
$in	Match values in an array	{ category: { $in: ["Electronics", "Appliances"] } }

16.2.8 Sorting, Limiting, and Pagination

When displaying product listings, queries often include sorting and pagination.

- Sort products by price (ascending):
- db.products.find().sort({ price: 1 })
- Get the top 5 most expensive products:
- db.products.find().sort({ price: -1 }).limit(5)
- Implement pagination (skip first 10 results and show next 5):
- db.products.find().skip(10).limit(5)

These are commonly used in e-commerce user interfaces where results are shown page by page.

16.2.9 Querying Nested Arrays and Embedded Documents

E-commerce documents often contain arrays of embedded documents (e.g., order items or multiple addresses).

Example:

db.orders.find({ "items.product_id": "P101" })

This query retrieves all orders containing a particular product ID within the items array.

MongoDB also allows querying for array size or matching multiple elements:

db.orders.find({ "items": { $size: 2 } })

16.2.10 Text Search Queries

For product searches, MongoDB's **text index** supports keyword-based search:
db.products.createIndex({ name: "text", description: "text" })
db.products.find({ $text: { $search: "wireless keyboard" } })
This enables full-text search functionality similar to e-commerce site search bars.

16.2.11 Query Performance and Indexing

Efficient querying requires **indexing** on frequently used fields such as:

- user_id in orders
- category and price in products
- product_id in reviews

Index creation example:
db.orders.createIndex({ user_id: 1 })
Indexes improve performance significantly by reducing scan time across large datasets.

16.3 MONGODB QUERY LANGUAGE (MQL)

16.3.1 Overview

The **MongoDB Query Language (MQL)** is the primary mechanism used to retrieve, filter, and manipulate data stored in a MongoDB database.
Unlike traditional SQL, MQL is **document-oriented** and operates on **JSON-like structures**, allowing developers to query nested documents and arrays with great flexibility.
In **e-commerce systems**, MQL enables developers to:

- Search for products by price, category, or availability.
- Retrieve user orders and purchase histories.
- Filter reviews and ratings.
- Generate targeted marketing insights.

Each query in MQL is expressed as a **JSON document** specifying the selection criteria and projection fields.

Example:
db.products.find({ category: "Electronics", price: { $lt: 2000 } })
This retrieves all products under the "Electronics" category that cost less than ₹2000.

16.3.2 Basic Query Syntax

The fundamental query operation in MongoDB is the **find()** method, which retrieves documents from a collection that match specified conditions.

Syntax:
db.collection.find(<query>, <projection>)

- **query** – specifies filtering criteria (like a WHERE clause in SQL).
- **projection** – specifies which fields to return (like SELECT columns).

Example:
db.users.find({ city: "Guntur" }, { name: 1, email: 1, _id: 0 })
This returns only the name and email fields for users located in Guntur.

16.3.3 Comparison Operators

MongoDB provides several comparison operators for filtering numeric and string values.

Operator	Meaning	Example
$eq	Equal to	{ price: { $eq: 1000 } }
$ne	Not equal to	{ category: { $ne: "Books" } }
$gt	Greater than	{ price: { $gt: 500 } }
$lt	Less than	{ price: { $lt: 1000 } }
$gte	Greater than or equal	{ rating: { $gte: 4 } }
$lte	Less than or equal	{ quantity: { $lte: 10 } }

Example **Query:**
Fetch all products with a price between ₹500 and ₹1500:
db.products.find({ price: { $gte: 500, $lte: 1500 } })

16.3.4 Logical Operators

Logical operators allow combining multiple conditions in a single query.

Operator	Description	Example
$and	Matches all conditions	{ $and: [{ category: "Electronics" }, { price: { $lt: 2000 } }] }
$or	Matches any condition	{ $or: [{ category: "Books" }, { category: "Stationery" }] }
$not	Negates a condition	{ price: { $not: { $gt: 5000 } } }
$nor	None of the conditions match	{ $nor: [{ category: "Toys" }, { category: "Gadgets" }] }

16.3.5 Querying Arrays

E-commerce data often includes **arrays** (e.g., product tags, order items).
Example:

```
{
 "product_id": "P120",
 "name": "Bluetooth Headphones",
 "tags": ["wireless", "electronics", "audio"]
}
```

- Find products tagged as “audio”:
  - db.products.find({ tags: "audio" })
- Match any tag in a list:
  - db.products.find({ tags: { $in: ["wireless", "gaming"] } })
- Match all specified tags:
  - db.products.find({ tags: { $all: ["wireless", "electronics"] } })

16.3.6 Querying Embedded Documents

Documents in MongoDB may contain nested structures. You can query embedded fields using **dot notation**.

Example Order Document:

```
{
  "_id": "O101",
  "user_id": "U001",
  "shipping": {
    "address": "Main Street",
    "city": "Vijayawada",
    "pincode": "520001"
  }
}
```

Find orders shipped to Vijayawada:

```
db.orders.find({ "shipping.city": "Vijayawada" })
```

16.3.7 Projection – Selecting Specific Fields

To limit the number of fields returned, MQL uses **projection**.

Syntax:

```
db.collection.find(<query>, <projection>)
```

Example:

```
db.products.find(
  { category: "Electronics" },
  { name: 1, price: 1, _id: 0 }
)
```

This query returns only the product name and price fields.
Projection reduces data transfer and improves query performance.

16.3.8 Sorting and Limiting Results

Sorting is achieved using the **sort()** method, and result count can be controlled with **limit()**.

Examples:

- Sort by ascending price:
- db.products.find().sort({ price: 1 })
- Sort by descending rating:
- db.products.find().sort({ rating: -1 })
- Get the top 10 most expensive products:
- db.products.find().sort({ price: -1 }).limit(10)

16.3.9 Using Regular Expressions and Text Search

MongoDB supports pattern matching for partial searches using regular expressions.

Example:

```
db.products.find({ name: { $regex: "wireless", $options: "i" } })
```

This retrieves all products whose names contain “wireless” (case-insensitive).
For advanced search, MongoDB provides **text indexes**:

```
db.products.createIndex({ name: "text", description: "text" })
db.products.find({ $text: { $search: "Bluetooth Speaker" } })
```

16.3.10 Querying with $exists and $type

You can check whether a field exists or verify its data type.

Examples:

- Find documents missing the “rating” field:
- db.products.find({ rating: { $exists: false } })
- Find documents where “price” is of numeric type:
- db.products.find({ price: { $type: "number" } })

16.3.11 Querying Orders by Date

E-commerce systems frequently need date-based queries for reporting.

Example Order Document:

```
{
  "order_id": "O501",
  "order_date": ISODate("2025-10-10T10:30:00Z"),
  "status": "Delivered"
}
```

Query all orders in October 2025:

```
db.orders.find({
  order_date: { $gte: ISODate("2025-10-01"), $lt: ISODate("2025-11-01") }
})
```

16.3.12 Compound Queries

Compound queries combine multiple filters to match complex criteria.

Example:

Find all “Electronics” products priced below ₹2000 and rated above 4:

```
db.products.find({
  $and: [
    { category: "Electronics" },
    { price: { $lt: 2000 } },
    { rating: { $gt: 4 } }
  ]
})
```

16.3.13 Query Optimization

MongoDB optimizes query performance through:

- **Index utilization**
- **Covered queries** (where all requested fields are in the index)
- **Explain plans** to analyze query execution.

Example:

```
db.products.find({ category: "Electronics" }).explain("executionStats")
```

This command provides insight into query efficiency and index usage.

16.3.14 Example Queries in E-Commerce

Purpose	Example Query
List all active customers in a city	db.users.find({ city: "Guntur" })
Retrieve all paid orders	db.orders.find({ status: "Paid" })
Display products under ₹1000	db.products.find({ price: { $lt: 1000 } })
Find top-rated products	db.products.find({ rating: { $gte: 4.5 } })
Search reviews with “excellent”	db.reviews.find({ comment: /excellent/i })

16.4 AGGREGATION FRAMEWORK

16.4.1 Introduction

While queries in MongoDB are used to retrieve individual documents, the **Aggregation Framework** is designed to perform **data analysis and computation** directly within the database.

It allows you to **process large datasets**, transform documents, and **generate summarized results** — similar to SQL’s GROUP BY, SUM(), COUNT(), and AVG() operations, but with much greater flexibility and scalability.

In **e-commerce applications**, aggregation is essential for tasks such as:

- Calculating total revenue per product or category.
- Finding average customer ratings.
- Counting total orders or customers.
- Generating sales reports by region or time period.

MongoDB’s aggregation is **pipeline-based**, where data flows through multiple stages, each performing a specific transformation.

16.4.2 What Is an Aggregation Pipeline?

An **aggregation pipeline** is a sequence of stages that process data step-by-step. Each stage transforms documents and passes the results to the next stage.

Pipeline Concept:

Input Collection → Stage 1 ($match) → Stage 2 ($group) → Stage 3 ($sort) → Output

Each stage performs an operation such as filtering, grouping, or sorting. The power of the aggregation framework lies in chaining multiple stages together for complex analytics.

Basic Syntax:

```
db.collection.aggregate([
  { <stage1> },
  { <stage2> },
  ...
])
```

16.4.3 Key Aggregation Stages

MongoDB provides several built-in stages. The most frequently used are listed below with examples.

(a) $match – Filtering Documents
This stage filters documents according to specified criteria, similar to the find() query.
Example: Retrieve only "Paid" orders.
{ $match: { status: "Paid" } }

(b) $project – Selecting Specific Fields
The $project stage reshapes documents by including, excluding, or computing new fields.

Example: Show only product_id, quantity, and computed total_price:
```
{
 $project: {
  product_id: 1,
  quantity: 1,
  total_price: { $multiply: ["$quantity", "$price"] }
 }
}
```

(c) $group – Grouping and Summarizing Data

This stage groups documents by a specified field and computes aggregate values such as totals or averages.

Example: Calculate total sales for each product.
```
{
 $group: {
  _id: "$product_id",
  totalSales: { $sum: "$price" }
 }
}
```

(d) $sort – Sorting Results
Used to arrange output documents in ascending or descending order.

Example: Sort categories by total revenue (descending).
{ $sort: { totalRevenue: -1 } }

(e) $limit – Restricting Output Count
Limits the number of documents returned.
Example: Show top 5 best-selling products.
{ $limit: 5 }

(f) $skip – Skipping Documents
Used for pagination in analytics reports.
Example: Skip first 10 records, return next 5.
{ $skip: 10 }

(g) $unwind – Deconstructing Arrays
When a field contains an array (like multiple order items), $unwind splits the array elements into individual documents.

Example:
{ $unwind: "$items" }
If each order contains multiple items, this stage processes each item separately.

(h) $lookup – Performing Joins

The $lookup stage performs a left outer join between two collections, similar to SQL joins.
Example: Join orders with products to display product names in each order.

```
{
 $lookup: {
   from: "products",
   localField: "items.product_id",
   foreignField: "_id",
   as: "product_details"
 }
}
```

(i) $addFields / $set – Creating New Fields
These stages add or modify fields dynamically.
Example: Add a computed field total to each order.
{ $addFields: { total: { $sum: "$items.price" } } }

(j) $count – Counting Documents
This stage outputs a count of all documents that passed previous stages.
Example:
{ $count: "totalOrders" }

16.4.4 Example: Complete Aggregation Pipeline

Let's calculate the **total sales revenue per product category** from an e-commerce database.

```
db.orders.aggregate([
  { $unwind: "$items" },
  {
    $lookup: {
      from: "products",
      localField: "items.product_id",
      foreignField: "_id",
      as: "product_info"
    }
  },
  { $unwind: "$product_info" },
  {
    $group: {
      _id: "$product_info.category",
```

```
    totalRevenue: { $sum: "$items.price" },
    totalQuantity: { $sum: "$items.quantity" }
   }
  },
  { $sort: { totalRevenue: -1 } }
])
```

Explanation:

1. $unwind — Expands each order's items array.
2. $lookup — Joins orders with products to get category info.
3. $group — Calculates total revenue and quantity for each category.
4. $sort — Sorts categories by descending revenue.

Result Example:

```
[
  { "_id": "Electronics", "totalRevenue": 350000, "totalQuantity": 900 },
  { "_id": "Home Appliances", "totalRevenue": 210000, "totalQuantity": 450 },
  { "_id": "Books", "totalRevenue": 90000, "totalQuantity": 1200 }
]
```

16.4.5 Aggregation Operators

Aggregation operators are used within stages to perform calculations or transformations.

Operator	Purpose	Example
$sum	Adds numeric values	{ $sum: "$amount" }
$avg	Computes average	{ $avg: "$rating" }
$min	Minimum value	{ $min: "$price" }
$max	Maximum value	{ $max: "$price" }
$push	Creates an array of values	{ $push: "$product_id" }
$first / $last	Returns first or last element in a group	{ $first: "$order_date" }

16.4.6 Aggregation vs. Simple Queries

Aspect	Query (find())	Aggregation (aggregate())
Purpose	Retrieve documents	Transform and summarize data
Output	Individual records	Computed summaries
Complexity	Simple filters	Multi-stage pipelines
Use Case	Search for users or products	Generate sales reports or averages

16.4.7 Real-World E-Commerce Aggregation Use Cases

1. **Total Revenue per Product Category**
   - { $group: { _id: "$category", total: { $sum: "$price" } } }
2. **Average Rating per Product**
   - { $group: { _id: "$product_id", avgRating: { $avg: "$rating" } } }
3. **Number of Orders per Customer**
   - { $group: { _id: "$user_id", orders: { $sum: 1 } } }
4. **Top 5 Selling Products**
   - { $sort: { totalSales: -1 } }, { $limit: 5 }

5. **Daily Revenue Trends**
   - { $group: { _id: "$order_date", dailyRevenue: { $sum: "$amount" } } }

16.4.8 Performance Optimization in Aggregations

- **Use Indexes:** Especially on fields used in $match and $sort.
- **Place $match Early:** Filter data before grouping to reduce processing.
- **Use Projection ($project):** Include only necessary fields.
- **Monitor Pipeline Performance:** Use .explain() to analyze query execution.
- **Avoid Excessive $lookup:** Large joins can slow down performance; denormalize if necessary.

Example:

```
db.orders.aggregate([
  { $match: { status: "Paid" } },
  { $group: { _id: "$user_id", totalSpent: { $sum: "$total_amount" } } }
]).explain("executionStats")
```

16.4.9 Advantages of Aggregation Framework

- Performs **real-time analytics** directly in the database.
- Reduces the need for external reporting tools.
- Handles **large volumes of data** efficiently.
- Offers flexibility through pipeline design.
- Integrates easily with BI dashboards and APIs.

16.5 AGGREGATING ORDERS IN E-COMMERCE

16.5.1 Overview

In an e-commerce database, the **orders collection** contains valuable information about products purchased, quantities, and amounts paid.
Using the **aggregation framework**, we can summarize this data to obtain key business insights such as **total revenue**, **number of orders**, and **top customers**.

16.5.2 Total Sales Per User

To calculate how much each customer has spent:

```
db.orders.aggregate([
  { $match: { status: "Paid" } },
  { $group: { _id: "$user_id", totalSpent: { $sum: "$total_amount" } } },
  { $sort: { totalSpent: -1 } }
])
```

✅ ***Shows top-spending customers.***

16.5.3 Top-Selling Products

To identify the most frequently purchased items:

```
db.orders.aggregate([
  { $unwind: "$items" },
  { $group: { _id: "$items.product_id", totalSold: { $sum: "$items.quantity" } } },
```

```
  { $sort: { totalSold: -1 } },
  { $limit: 5 }
])
```

✅ ***Lists the top 5 best-selling products.***

16.5.4 Total Revenue by Category

Join the orders and products collections to summarize sales by category:

```
db.orders.aggregate([
  { $unwind: "$items" },
  {
    $lookup: {
      from: "products",
      localField: "items.product_id",
      foreignField: "_id",
      as: "product_info"
    }
  },
  { $unwind: "$product_info" },
  {
    $group: {
      _id: "$product_info.category",
      totalRevenue: { $sum: "$items.price" }
    }
  },
  { $sort: { totalRevenue: -1 } }
])
```

✅ *Helps identify the highest-earning categories.*

16.5.5 Average Order Value (AOV)

To measure the average purchase amount per order:

```
db.orders.aggregate([
  { $group: { _id: null, avgOrderValue: { $avg: "$total_amount" } } }
])
```

✅ ***Useful for evaluating customer spending behavior.***

16.5.6 Monthly Sales Trend

To view revenue over time:

```
db.orders.aggregate([
  {
    $group: {
      _id: { month: { $month: "$order_date" }, year: { $year: "$order_date" } },
      monthlyRevenue: { $sum: "$total_amount" }
    }
  },
  { $sort: { "_id.year": 1, "_id.month": 1 } }
])
```

✅ ***Generates a monthly sales chart for performance tracking.***

16.6 AGGREGATION IN DETAIL

16.6.1 Overview

The **aggregation framework** in MongoDB provides a powerful way to analyze and transform data. It processes documents through multiple **stages**, where each stage performs a specific operation — such as filtering, grouping, or calculating totals. Aggregations are widely used in **e-commerce analytics** to compute **sales summaries**, **customer metrics**, and **product trends**.

16.6.2 The $group Stage

The $group stage groups documents by a key and performs aggregate operations like sum, average, or count.

Example – Total Sales per Product:

```
{
 $group: {
   _id: "$product_id",
   totalSales: { $sum: "$total_amount" }
 }
}
```

✅ Groups by product_id and sums total sales.

16.6.3 Common Accumulators

Operator	Description	Example Use
$sum	Adds values	{ $sum: "$amount" }
$avg	Averages values	{ $avg: "$rating" }
$min	Smallest value	{ $min: "$price" }
$max	Largest value	{ $max: "$price" }
$count	Counts documents	{ $sum: 1 }

16.6.4 Using $lookup for Joins

$lookup connects data from multiple collections, like orders and products.

Example:

```
{
 $lookup: {
   from: "products",
   localField: "items.product_id",
   foreignField: "_id",
   as: "product_info"
 }
}
```

✅ Brings product details into order data.

16.6.5 THE $PROJECT STAGE

Used to select specific fields or create new computed fields.

Example:

```
{
  $project: {
    product_name: 1,
    total_value: { $multiply: ["$price", "$quantity"] }
  }
}
```

✅ Computes a new field total_value.

16.6.6 THE $UNWIND STAGE

Splits arrays into separate documents — useful for analyzing order items.

Example:

```
{ $unwind: "$items" }
```

✅ Processes each order item individually.

16.6.7 COMBINING MULTIPLE STAGES

Complex analytics can be performed by combining multiple stages in a pipeline.

Example – Total Revenue per Category:

```
db.orders.aggregate([
  { $unwind: "$items" },
  { $lookup: {
      from: "products",
      localField: "items.product_id",
      foreignField: "_id",
      as: "product_info"
    } },
  { $unwind: "$product_info" },
  { $group: {
      _id: "$product_info.category",
      revenue: { $sum: "$items.price" }
    } },
  { $sort: { revenue: -1 } }
])
```

✅ Shows category-wise revenue ranking.

16.7 PRACTICAL EXAMPLES AND CASE STUDIES

16.7.1 Overview

In real-world e-commerce systems, **queries and aggregations** are essential for generating **reports, dashboards, and insights**.

MongoDB's flexibility allows developers to extract meaningful information from large datasets to improve **sales strategies**, **inventory planning**, and **customer engagement**.

16.7.2 Example 1 – Sales Dashboard

A dashboard displays real-time sales metrics using aggregation pipelines.

Pipeline:

```
db.orders.aggregate([
  { $match: { status: "Paid" } },
  { $group: { _id: null, totalSales: { $sum: "$total_amount" }, totalOrders: { $sum: 1 } } }
])
```

✅ *Shows overall sales and total orders.*

16.7.3 Example 2 – Top 5 Selling Products

Identify products with the highest sales volume.

```
db.orders.aggregate([
  { $unwind: "$items" },
  { $group: { _id: "$items.product_id", totalQty: { $sum: "$items.quantity" } } },
  { $sort: { totalQty: -1 } },
  { $limit: 5 }
])
```

✅ *Useful for inventory and marketing planning.*

16.7.4 Example 3 – Customer Insights

Determine top-spending customers.

```
db.orders.aggregate([
  { $match: { status: "Paid" } },
  { $group: { _id: "$user_id", totalSpent: { $sum: "$total_amount" } } },
  { $sort: { totalSpent: -1 } }
])
```

✅ *Helps identify loyal or high-value customers.*

16.8 BEST PRACTICES AND OPTIMIZATION TECHNIQUES

Efficient query and aggregation performance is crucial in large-scale **e-commerce systems**. As data grows, poorly designed queries or pipelines can slow down operations. Following best practices ensures faster results, optimized resource use, and a better user experience.

Key optimization strategies include:

- Indexing critical fields.
- Filtering early in pipelines.
- Using projections and limits.
- Reducing $lookup usage.
- Monitoring queries and caching results.

16.9 SUMMARY

This lesson explored how **queries and aggregations** are used in **document-oriented databases**, particularly MongoDB, to manage and analyze **e-commerce data**.

Key points covered include:

- The role of **queries** for retrieving and filtering data using MongoDB Query Language (MQL).
- The concept and structure of the **Aggregation Framework** for data summarization.

- Practical **e-commerce examples**, such as retrieving top-selling products, computing total revenue, and generating customer insights.
- Optimization techniques like **indexing, projection, and pipeline efficiency**.
- Efficient use of queries and aggregations allows e-commerce businesses to make **data-driven decisions**, improve **performance**, and gain **actionable insights** from large datasets.

16.10 TECHNICAL TERMS

1. MQL (MongoDB Query Language)
2. Aggregation Framework
3. Pipeline
4. Stage
5. Accumulator
6. $match
7. $group
8. $lookup
9. $unwind
10. Projection
11. Index
12. Denormalization
13. Sharding

16.11 SELF-ASSESSMENT QUESTIONS

Short Answer Questions

1. What is MongoDB Query Language (MQL)?
2. Differentiate between a query and an aggregation.
3. What is the purpose of the $group stage in MongoDB?
4. Define the term *accumulator* in the context of aggregations.
5. List any three operators used in aggregation pipelines.

Long Answer Questions

1. Explain in detail the working of the MongoDB Query Language (MQL) with suitable e-commerce examples.
2. Describe the stages of the Aggregation Framework and explain their roles with examples.
3. Write a MongoDB aggregation pipeline to calculate total sales per product category.

16.12 SUGGESTED READINGS

1. Kristina Chodorow – *MongoDB: The Definitive Guide*, O'Reilly Media.
2. Rick Copeland – *MongoDB Applied Design Patterns*, O'Reilly Media.
3. Ramez Elmasri & Shamkant B. Navathe – *Fundamentals of Database Systems*, Pearson Education.
4. MongoDB Documentation – *Aggregation Pipeline Stages and Operators*.
5. MongoDB University – *Data Modeling and Aggregation Framework*.

6. Google Cloud Documentation – *Building E-Commerce Analytics with MongoDB Atlas*.
7. Alex Giamas – *Practical MongoDB Aggregations*, Leanpub.
8. Amazon Web Services – *NoSQL Design Patterns for Scalable Applications*.
9. Couchbase Documentation – *Query and Indexing Best Practices*.
10. Michael Harrison – *Mastering MongoDB 6.x: Expert Techniques for Data Aggregation and Performance Optimization*.

Dr. U. Surya Kameswari

LESSON- 17

UPDATES ATOMIC OPERATIONS AND DELETES

AIMS AND OBJECTIVES:

The aim of this lesson is to explain how updates, atomic operations, and deletions are performed in document-oriented databases, particularly in MongoDB, and how these operations ensure data consistency, accuracy, and reliability in dynamic applications such as e-commerce systems.

This lesson emphasizes the principles behind modifying and maintaining data, introduces the concept of atomicity in document processing, and explores the practical usage of MongoDB's update and delete operations in real-world business scenarios.

After completing this lesson, learners will be able to:

- Understand the principles of document update and delete operations in MongoDB.
- Differentiate between update types — single, multiple, and replacement updates.
- Apply update operators such as $set, $inc, $unset, $push, and $pull to modify specific fields.
- Explain the concept of atomicity and its role in ensuring consistent data transactions.
- Implement atomic operations for concurrent updates and multi-user environments.
- Perform practical update operations in e-commerce applications, such as modifying prices, stock, and customer details.
- Use transactions to maintain data integrity across multiple collections.
- Execute and manage document deletions, both individually and in bulk.
- Implement soft delete strategies to preserve historical or inactive data.
- Optimize update and delete operations for high-performance e-commerce databases using indexing and bulk operations.

STRUCTURE:

17.1 INTRODUCTION

17.1.1 Overview

In a document-oriented database like MongoDB, updates and deletes are fundamental operations used to modify or remove data from collections.
Unlike relational databases where updates affect rows in tables, MongoDB updates operate on JSON-like documents, providing greater flexibility in modifying nested structures and arrays.

Updates enable applications to reflect real-time changes—for example, updating product prices, stock quantities, customer details, or order statuses—while deletes help maintain a clean and efficient database by removing outdated or redundant records.
In addition, atomic operations ensure that updates to a document are applied completely or not at all, preserving data consistency even in concurrent environments where multiple users or processes may attempt to modify the same data.

17.1.2 The Need for Updates in MongoDB

E-commerce platforms and real-time applications require frequent updates to maintain accurate information.

For example:

- When a customer places an order, the stock quantity must be reduced.
- If an order is canceled, the inventory must be restored.
- When a user edits their profile, the corresponding document must be updated instantly.

MongoDB provides efficient methods like updateOne(), updateMany(), and replaceOne() to modify documents selectively or entirely, without affecting other unrelated data.

17.1.3 Understanding Deletes in MongoDB

Delete operations are equally important to maintain data hygiene and relevance.
For example, an e-commerce company may need to:

- Delete products that are no longer sold.
- Remove customer accounts upon request (to comply with data privacy regulations).
- Purge old transaction records after a certain period for archival or compliance purposes.

MongoDB provides the deleteOne() and deleteMany() methods to remove specific or multiple documents efficiently, ensuring that only targeted data is deleted.

17.1.4 Importance of Atomic Operations

Atomic operations ensure that all changes to a document occur as a single, indivisible action. If multiple users attempt to update the same document simultaneously, MongoDB guarantees that each update is applied in isolation, preventing conflicts and data corruption.
For instance, when two customers attempt to purchase the last available unit of a product simultaneously, MongoDB's atomic update ensures that only one order is confirmed, and the stock value is adjusted correctly.

Atomicity is particularly critical in e-commerce for:

- Order management (preventing duplicate confirmations).
- Inventory synchronization (avoiding negative stock).
- Financial transactions (ensuring accurate billing and wallet updates).

17.1.5 Updates and Deletes in E-Commerce Applications

In an e-commerce context, updates and deletes occur constantly.
Here are some real-world examples:

Scenario	**Operation Type**	**MongoDB Method**
Updating product prices or descriptions	Update	updateOne()
Increasing loyalty points after purchase	Atomic Update	$inc
Changing order status from "Pending" to "Delivered"	Update	updateMany()
Removing expired offers or discount coupons	Delete	deleteMany()
Deleting inactive user accounts	Delete	deleteOne()
Restoring inventory after order cancellation	Atomic Update	$inc (reverse stock)

Such operations ensure that the database remains accurate, responsive, and consistent with real-world changes.

Example:

An online store wants to update all "Winter Collection" products to offer a 10% discount.

```
db.products.updateMany(
  { category: "Winter Collection" },
  { $mul: { price: 0.9 } }
)
```

This single command reduces the price of all products in the specified category by 10%, illustrating the simplicity and power of MongoDB's update capabilities.
Similarly, to remove all expired discount codes, the following delete operation can be performed:

db.coupons.deleteMany({ expiry_date: { $lt: new Date() } })
Both operations are efficient, atomic (per document), and ideal for real-time e-commerce

17.2 UNDERSTANDING DOCUMENT UPDATES

17.2.1 Overview

In MongoDB, **update operations** modify existing documents in a collection. They can change single fields, multiple fields, or replace an entire document. Common methods include updateOne(), updateMany(), and replaceOne().

17.2.2 The updateOne() and updateMany() Methods

- **updateOne()** modifies the first matching document.
- **updateMany()** modifies all documents that match the filter.

Example:
db.products.updateOne({ _id: "P101" }, { $set: { price: 499 } })
db.products.updateMany({ category: "Books" }, { $inc: { stock: 10 } })

17.2.3 Replace Operation

replaceOne() completely replaces an existing document with a new one.
db.users.replaceOne({ _id: "U001" }, { name: "Lavanya", city: "Guntur" })

17.2.4 Common Update Operators

- **$set** – Assigns a new value to a field.
  - { $set: { status: "Delivered" } }
- **$inc** – Increases or decreases a numeric field.
  - { $inc: { loyalty_points: 100 } }
- **$unset** – Removes a field.
  - { $unset: { discount: "" } }
- **$rename** – Renames a field.
  - { $rename: { "old_field": "new_field" } }
- **$push / $pull** – Adds or removes array elements.
  - { $push: { tags: "new-arrival" } }
  - { $pull: { tags: "outdated" } }
- **$addToSet** – Adds a value to an array only if it doesn't exist.
  - { $addToSet: { tags: "featured" } }

17.2.5 Upsert Option

An **upsert** updates a document if it exists; otherwise, it inserts a new one.

```
db.products.updateOne(
  { name: "Wireless Mouse" },
  { $set: { stock: 50 } },
  { upsert: true }
)
```

17.2.6 Updating Embedded Documents

Fields inside nested structures can be updated using **dot notation**.

```
db.orders.updateOne(
  { _id: "O1001" },
  { $set: { "shipping.city": "Vijayawada" } }
)
```

17.2.7 E-Commerce Examples

- **Update order status:**
- db.orders.updateOne({ _id: "O2001" }, { $set: { status: "Shipped" } })
- **Increase stock after restock:**
- db.products.updateMany({ category: "Electronics" }, { $inc: { stock: 25 } })
- **Add a new review to a product:**

db.products.updateOne({ _id: "P101" }, { $push: { reviews: "Excellent!" } })

17.3 ATOMIC OPERATIONS

17.3.1 Overview

Atomic operations in MongoDB ensure that updates to a document are **completed entirely or not at all**.

Each document update is atomic by default, even when multiple clients are modifying data simultaneously.

17.3.2 Single-Document Atomicity

All updates to a **single document** (including embedded fields and arrays) are atomic. This prevents partial updates and maintains consistency.

Example:

When two users order the same item, MongoDB ensures only one successfully decreases the stock count.

17.3.3 Common Atomic Operators

- **$inc** – Adjusts numeric fields atomically.
  - db.products.updateOne({ _id: "P101" }, { $inc: { stock: -1 } })
- **$set** – Updates field values.
  - db.orders.updateOne({ _id: "O201" }, { $set: { status: "Delivered" } })
- **$push / $pull** – Modifies arrays atomically.
  - db.users.updateOne({ _id: "U001" }, { $push: { wishlist: "P205" } })

17.3.4 Atomicity in Embedded Documents

Updates made to nested fields are applied atomically at the **document level**.

```
db.orders.updateOne(
  { _id: "O301" },
  { $set: { "payment.status": "Confirmed" } }
)
```

17.3.5 Multi-Document Transactions

For operations spanning multiple documents or collections, MongoDB uses **transactions** to maintain atomicity.

```
const session = db.getMongo().startSession()
session.startTransaction()
db.orders.updateOne({ _id: "O1001" }, { $set: { status: "Paid" } }, { session })
db.users.updateOne({ _id: "U1001" }, { $inc: { wallet: -500 } }, { session })
session.commitTransaction()
```

17.3.6 Importance in E-Commerce

- Prevents overselling of products.
- Ensures consistent order and payment updates.
- Maintains synchronization between user and inventory data.
- Supports safe concurrent updates during high traffic.

17.4 E-COMMERCE UPDATE EXAMPLES

17.4.1 Overview

E-commerce systems frequently perform **update operations** to manage inventory, customer information, and order statuses.
MongoDB provides flexible update methods that allow changes to be made efficiently in real-time.

17.4.2 Updating Product Details

Example 1 – Changing Product Price:

```
db.products.updateOne({ _id: "P101" }, { $set: { price: 999 } })
```

Example 2 – Increasing Stock:

```
db.products.updateOne({ _id: "P101" }, { $inc: { stock: 20 } })
```

Example 3 – Adding Tags:

```
db.products.updateOne({ _id: "P101" }, { $addToSet: { tags: "new-arrival" } })
```

17.4.3 Updating Customer Information

Example – Change Address and Phone Number:

```
db.users.updateOne(
  { _id: "U201" },
  { $set: { "address.city": "Vijayawada", phone: "9876543210" } }
)
```

17.4.4 Updating Order Status

Example – Change Order Status from Pending to Shipped:

```
db.orders.updateOne(
  { _id: "O501", status: "Pending" },
  { $set: { status: "Shipped", shipped_date: new Date() } }
)
```

17.4.5 Updating Product Reviews

Example – Add a New Review:

```
db.products.updateOne(
  { _id: "P301" },
  { $push: { reviews: { user: "U101", rating: 5, comment: "Excellent!" } } }
)
```

17.4.6 Bulk Updates for Promotions

Example – Apply 10% Discount to All Electronics:

```
db.products.updateMany(
  { category: "Electronics" },
  { $mul: { price: 0.9 } }
)
```

17.4.7 Loyalty Points Update

Example – Reward Customers After Purchase:

```
db.users.updateMany(
  { total_orders: { $gte: 5 } },
  { $inc: { loyalty_points: 50 } }
)
```

17.4.8 Inventory Adjustment after Return

Example – Restock Returned Product:

```
db.products.updateOne({ _id: "P205" }, { $inc: { stock: 1 } })
```

17.5 Nuts and Bolts of MongoDB Updates

17.5.1 Overview

MongoDB provides multiple update methods and operators that allow precise modifications of documents. These updates can target single or multiple documents, and can also insert data when no match is found (upsert).

17.5.2 The updateOne() Method

Updates the **first matching document**.

Syntax:

```
db.collection.updateOne(filter, update, options)
```

Example:

```
db.products.updateOne({ _id: "P101" }, { $set: { price: 750 } })
```

17.5.3 The updateMany() Method

Updates **all matching documents**.

Example:
db.products.updateMany({ category: "Books" }, { $inc: { stock: 5 } })

17.5.4 The replaceOne() Method

Replaces an entire document with a new one.
Example:
db.users.replaceOne({ _id: "U101" }, { name: "Lavanya", city: "Guntur" })

17.5.5 The findOneAndUpdate() Method

Finds and updates a document, returning the **modified document**.

Example:
```
db.orders.findOneAndUpdate(
  { _id: "O101" },
  { $set: { status: "Delivered" } },
  { returnDocument: "after" }
)
```

17.5.6 Update Options

- **upsert: true** → Inserts if no matching document exists.
- **arrayFilters** → Filters array elements during updates.
- **multi** → Allows updating multiple documents (for legacy use).

Example (upsert):
```
db.products.updateOne(
  { name: "Smart Lamp" },
  { $set: { stock: 15 } },
  { upsert: true }
)
```

17.5.7 Using Date and Conditional Fields

MongoDB provides operators for timestamps and conditional inserts.
```
db.orders.updateOne(
  { _id: "O201" },
  {
   $currentDate: { updated_at: true },
   $setOnInsert: { created_at: new Date() }
  },
  { upsert: true }
)
```

17.5.8 Performance Considerations

- Use **indexes** on frequently updated fields.
- Avoid frequent updates to large arrays.
- Prefer $set for partial updates instead of replacing entire documents.
- Use **bulk operations** for batch updates.

Example – Bulk Write:

```
db.products.bulkWrite([
  { updateOne: { filter: { _id: "P101" }, update: { $inc: { stock: 5 } } } },
  { updateOne: { filter: { _id: "P102" }, update: { $set: { price: 899 } } } }
])
```

17.6 Deleting Documents

Aspect	Description	Example / Use Case
Definition	Removes documents from a MongoDB collection based on filter criteria.	Deleting expired coupons or cancelled orders.
Primary Methods	deleteOne() – Deletes first matching document.deleteMany() – Deletes all matching documents.	db.orders.deleteMany({ status: "Cancelled" })
Filter Use	Ensures only targeted data is deleted to prevent data loss.	db.users.deleteOne({ _id: "U501" })
Soft Delete	Marks documents as inactive instead of removing them.	db.products.updateOne({ _id: "P101" }, { $set: { is_deleted: true } })
Bulk Delete	Removes multiple outdated or irrelevant records efficiently.	db.logs.deleteMany({ created_at: { $lt: ISODate("2025-01-01") } })
Precaution	Avoid using empty filters (e.g., {}) to prevent accidental full deletion.	❌ db.orders.deleteMany({})
Performance Tips	Use indexes on filter fields and schedule deletions during low traffic.	Improves efficiency in large datasets.
Use in E-Commerce	Clean up old orders, remove expired offers, or deactivate discontinued products.	Keeps database optimized and up to date.

17.6 Deleting Documents

Aspect	Description	Example / Use Case
Definition	Removes documents from a MongoDB collection based on filter criteria.	Deleting expired coupons or cancelled orders.
Primary Methods	deleteOne() – Deletes first matching document. deleteMany() – Deletes all matching documents.	db.orders.deleteMany({ status: "Cancelled" })
Filter Use	Ensures only targeted data is deleted to prevent data loss.	db.users.deleteOne({ _id: "U501" })
Soft Delete	Marks documents as inactive instead of removing them.	db.products.updateOne({ _id: "P101" }, { $set: { is_deleted: true } })
Bulk Delete	Removes multiple outdated or irrelevant records efficiently.	db.logs.deleteMany({ created_at: { $lt: ISODate("2025-01-01") } })
Precaution	Avoid using empty filters (e.g., {}) to prevent accidental full deletion.	❌ db.orders.deleteMany({})

Performance Tips	Use indexes on filter fields and schedule deletions during low traffic.	Improves efficiency in large datasets.
Use in E-Commerce	Clean up old orders, remove expired offers, or deactivate discontinued products.	Keep

17.8 E-Commerce Case Study: Inventory and Order Synchronization

Aspect	**Description**	**Example / Implementation**
Scenario	Synchronizing inventory and order status during customer purchases.	When a product is ordered, stock decreases and order status updates.
Challenge	Prevent inconsistent updates (e.g., overselling or duplicate orders).	Two customers buying the last item simultaneously.
Solution	Use **atomic updates** or **transactions** to ensure consistency.	Atomic decrement of stock and order creation in one step.
Atomic Update Example	js db.products.updateOne({ _id: "P101", stock: { $gt: 0 } }, { $inc: { stock: -1 } })	Reduces stock by one only if available.
Transactional Example	js session.startTransaction() db.orders.updateOne({ _id: "O101" }, { $set: { status: "Confirmed" } }, { session }) db.products.updateOne({ _id: "P101" }, { $inc: { stock: -1 } }, { session }) session.commitTransaction()	Ensures both order confirmation and stock deduction occur together.
Result	Prevents partial or conflicting updates between orders and products.	Maintains accurate stock and order data in real time.
Benefits	- Reliable order management. - Accurate inventory levels. - Seamless user experience.	

17.9 Performance and Optimization

Aspect	Description	Example / Recommendation
Goal	Improve the efficiency and reliability of update and delete operations in MongoDB.	Ensures faster processing in large e-commerce databases.
Use Indexes	Create indexes on frequently queried or updated fields.	db.orders.createIndex({ user_id: 1 })
Use Projection	Fetch or update only required fields.	db.products.find({}, { name: 1, price: 1 })
Batch Operations	Perform multiple updates or deletes together for better efficiency.	db.products.bulkWrite([...])
Avoid Large Array Modifications	Minimize updates on documents with large arrays.	Store reviews or logs in separate collections.
Place $match Early	Filter documents early in aggregation or bulk updates.	Reduces memory usage and improves speed.
Use Upserts Wisely	Enable upsert only when necessary to prevent unintended inserts.	updateOne({ name: "item" }, { $set: {...} }, { upsert: true })
Soft Deletes for History	Mark documents as inactive instead of deleting.	db.users.updateOne({ _id: "U101" }, { $set: { is_deleted: true } })
Monitor with .explain()	Analyze query plans and optimize indexes.	db.orders.find({ status: "Paid" }).explain("executionStats")
Hardware and Sharding	Use replica sets and sharding for scalability and fault tolerance.	Distribute data across multiple nodes.
Automation	Schedule cleanup and optimization tasks during off-peak hours.	Cron jobs for archiving and deleting old data.

17.10 SUMMARY

In this lesson, we explored how MongoDB handles updates, atomic operations, and deletions within document-oriented databases. Update operations allow partial or complete modification of documents using operators such as $set, $inc, $push, and $unset. Atomic operations ensure that each update to a document is executed completely or not at all, preventing conflicts in concurrent environments. We also discussed deletion operations using deleteOne() and deleteMany(), along with soft deletes for preserving data history. The lesson further examined how these operations apply to e-commerce systems—including order management, inventory synchronization, and data consistency across collections—supported by transactions and bulk operations for high performance.

17.11 Technical Terms

- Atomic Operation
- Upsert
- Soft Delete
- Transaction
- Bulk Write

17.12 Self-Assessment Questions

Short Answer Questions

1. What is the difference between updateOne() and updateMany() in MongoDB?
2. Define atomic operation and explain its importance in databases.
3. What is an upsert operation?
4. How does MongoDB handle deletion operations safely?
5. What is the advantage of using soft deletes in e-commerce databases?

Long Answer Questions

1. Explain the concept of atomic operations in MongoDB with suitable e-commerce examples.
2. Discuss different update operators in MongoDB and their use in real-world applications.
3. Describe the process and advantages of using transactions in MongoDB for maintaining data consistency.
4. Compare hard delete and soft delete strategies with examples from e-commerce applications.
5. How can performance be optimized for frequent update and delete operations in large-scale MongoDB systems?

17.13 Suggested Readings

1. Kristina Chodorow – *MongoDB: The Definitive Guide*, O'Reilly Media.
2. Rick Copeland – *MongoDB Applied Design Patterns*, O'Reilly Media.
3. MongoDB Documentation – *Update, Delete, and Atomic Operations*.
4. Alex Giamas – *Practical MongoDB Aggregations*, Leanpub.
5. Ramez Elmasri & Shamkant B. Navathe – *Fundamentals of Database Systems*, Pearson Education.
6. MongoDB University – *Transactions and Multi-Document Atomicity*.
7. AWS Whitepaper – *Building Scalable NoSQL Applications Using MongoDB*.

Dr. U. Surya Kameswari