

**DATABASE MANAGEMENT SYSTEMS**  
**MASTER OF COMPUTER APPLICATIONS (MCA)**  
**SEMESTER-I, PAPER-II**

**LESSON WRITERS**

**Dr. Kampa Lavanya**

Assistant Professor  
Department of CS&E  
University College of Sciences  
Acharya Nagarjuna University

**Dr. U. Surya Kameswari**

Assistant Professor  
Department of CS&E  
University College of Sciences  
Acharya Nagarjuna University

**Dr. Neelima Guntupalli**

Assistant Professor  
Department of CS&E  
University College of Sciences  
Acharya Nagarjuna University

**Mrs. Appikatla Puspha Latha**

Faculty  
Department of CS&E  
University College of Sciences  
Acharya Nagarjuna University

**EDITOR**

**Dr. Kampa Lavanya**

Assistant Professor  
Department of CS&E  
University College of Sciences  
Acharya Nagarjuna University

**DIRECTOR, I/c.**

**Prof. V. Venkateswarlu**

M.A., M.P.S., M.S.W., M.Phil., Ph.D.

Professor  
Centre for Distance Education  
Acharya Nagarjuna University  
Nagarjuna Nagar 522 510

Ph: 0863-2346222, 2346208  
0863- 2346259 (Study Material)  
Website [www.anucde.info](http://www.anucde.info)  
E-mail: [anucdedirector@gmail.com](mailto:anucdedirector@gmail.com)

# **MCA: DATABASE MANAGEMENT SYSTEMS**

**First Edition : 2025**

**No. of Copies :**

**© Acharya Nagarjuna University**

**This book is exclusively prepared for the use of students of Master of Computer Applications (MCA), 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*

**MASTER OF COMPUTER APPLICATIONS (MCA)**  
**Semester-I, Paper-II**  
**102MC24: DATABASE MANAGEMENT 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.

**Disk Storage, Basic File Structures and Hashing:** Introduction, Secondary Storage Devices, Buffering of Blocks, Placing file Records on Disk, Operations on Files, Files of Unordered Records, Files of Ordered Records, Hashing Techniques, Other Primary File Organizations, Parallelizing Disk Access using RAID Technology.

**Indexing Structures for Files:** Types of Single-Level Ordered Indexes, Multilevel Indexes and Dynamic Multilevel Indexes Using B-Trees and B\* Trees, Indexes on Multiple Keys, Other Types of Indexes.

**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-II**

**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, Schema Change Statements on SQL, Basic Queries in SQL, More Complex SQL Queries, INSERT, DELETE and UPDATE statements in SQL, Triggers and Views.

### **Unit-III**

**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 for Relational Database Schema Design, Multivalued Dependencies and Fourth Normal Form, Join Dependencies and Fifth Normal Form, Inclusion Dependencies, Other Dependencies and Normal Forms.

### **Unit-IV**

**Introduction to Transaction Processing Concepts and Theory:** Introduction to Transaction Processing, Transaction and System Concepts, Desirable Properties of Transactions, Characterizing Schedules Based on Recoverability, Characterizing schedules Based on Serializability.

**Concurrency Control Techniques:** Two Phase Locking Techniques for Concurrency Control, Concurrency Control Based on Timestamp Ordering, Multiversion Concurrency control techniques, Validation concurrency control Techniques, Granularity of Data Items and multiple Granularity Locking.

**Distributed Databases and Client Server Architectures:** Distributed Database Concepts, Data Fragmentation, Replication, and allocation Techniques for Distributed Database Design, Types of Distributed Database Systems, An Overview of 3 Tier Client Server Architecture.

#### **Prescribed Text:**

RamezElmasri, Shamkant B. Navathe, "Fundamentals of Database Systems", Fifth Edition, Pearson Education (2007)

#### **Reference Books:**

- 1) Peter Rob, Carlos Coronel, "Database Systems" - Design, Implementation and Management, Eighth Edition, Thomson (2008).
- 2) C.J. Date, A.Kannan, S. Swamynathan, "An Introduction to Database Systems", VII Edition Pearson Education (2006).
- 3) Raman A Mata - Toledo, Panline K. Cushman, "Database Management Systems", Schaum's Outlines, TMH (2007).
- 4) Steven Feuerstein, "Oracle PL/SQL-Programming", 10<sup>th</sup> Anniversary Edition, OREILLY (2008).

(102MC24)

**M.C.A. DEGREE EXAMINATION, MODEL QUESTION PAPER**  
**MCA-FIRST SEMESTER**  
**DATABASE MANAGEMENT SYSTEMS**

**Time: Three hours**

**Maximum: 70 marks**

**SECTION-A**

**Answer Question No.1 Compulsory:**

**7 x 2 = 14 M**

- 1 a) Mention advantages of using DBMS approach.
- b) Write about SELECT Operation?
- c) What is Relationship set s? Explain with Example.
- d) How is the trigger different from view?
- e) What is join dependency? Explain.
- f) Define Boyce-Codd Normal Form
- g) What Desirable Properties of Transactions?
- h) Explain about multiple Granularity Locking.

**SECTION-B**

**Answer ONE Question from each unit:**

**4 x 14 = 56 M**

**UNIT-I**

- 2 a) Discuss about specialization and generalization.
- b) Discuss about Entity sets and Entity Types

**OR**

- 3 a) Compare and Construct schema and instances
- b) Discuss about Hashing Techniques.

**UNIT-II**

- 4 a) Discuss about SELECT Operation with example?
- b) Illustrate the Transactions and Dealing with Constraint Violations.

**OR**

- 5 Discuss Tuple Calculus with example?

**UNIT-III**

- 6 a) Describe the Third Normal Form with example.
- b) How does inclusion dependency works? Explain.

**OR**

- 7 a) What is Functional Dependency? Explain with an example.
- b) Discuss the following
  - i. Algorithms for Relational Database Schema Design II NF
  - ii. Join Dependency.

**UNIT-IV**

- 8 Illustrate Characterizing Schedules Based on Recoverability.

**OR**

- 9 a) How Two Phase Locking Techniques for Concurrency Control work? Explain.
- b) Discuss 3 Tier Client Server Architecture with neat sketch.

# CONTENTS

<b>S.No.</b>	<b>TITLE</b>	<b>PAGE No.</b>
1	DATABASES AND DATABASE USERS	1.1-1.10
2	DATABASE SYSTEM CONCEPTS AND ARCHITECTURE	2.1-2.18
3	DISK STORAGE, BASIC FILE STRUCTURES AND HASHING	3.1-3.14
4	INDEXING STRUCTURES FOR FILES	4.1-4.12
5	DATA MODELING USING THE ER MODEL	5.1-5.16
6	THE ENHANCED ENTITY-RELATIONSHIP MODEL	6.1-6.15
7	THE RELATIONAL DATA MODEL AND RELATIONAL DATABASE CONSTRAINTS	7.1-7.11
8	THE RELATIONAL ALGEBRA	8.1-8.14
9	THE RELATIONAL CALCULUS	9.1-9.12
10	SQL-99	10.1-10.20
11	FUNCTIONAL DEPENDENCIES AND NORMALIZATION FOR RELATIONAL DATABASES	11.1-11.13
12	RELATIONAL DATABASE DESIGN ALGORITHMS AND FURTHER DEPENDENCIES	12.1-12.16
13	INTRODUCTION TO TRANSACTION PROCESSING CONCEPTS AND THEORY	13.1-13.17
14	CONCURRENCY CONTROL TECHNIQUES	14.1-14.18
15	DISTRIBUTED DATABASES AND CLIENT SERVER ARCHITECTURES	15.1-15.13

# **LESSON-1**

## **DATABASES AND DATABASE USERS**

### **AIMS AND OBJECTIVES:**

The end of the lesson student will be able to:

- Understand the concept of Databases and Database Users.
- Understanding of Characteristics of the Database Approach.
- Explore about Actors on the Scene, Workers behind the scene
- Know about Advantages of the using the DBMS Approach.

### **STRUCTURE:**

- 1.1. Introduction**
- 1.2. Characteristics of the Database Approach**
- 1.3. Actors on the Scene**
- 1.4. Advantages of DBMS Approach**
- 1.5. Applications of DBMS**
- 1.6. Summary**
- 1.7. Technical Terms**
- 1.8. Self-Assessment Questions**
- 1.9. Suggested Readings**

#### **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.

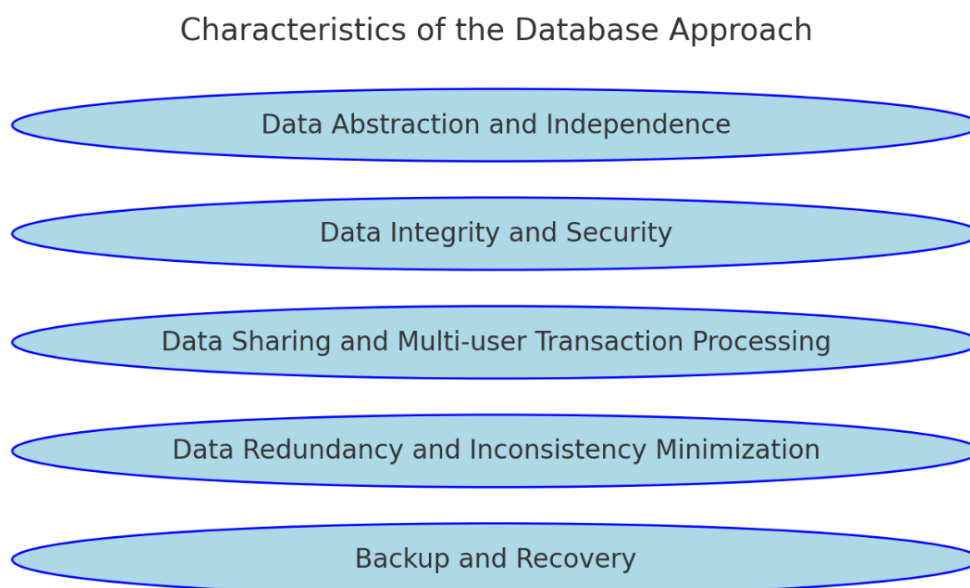
The chapter first covered the Characteristics of the Database Approach, Actors on the Scene, Workers behind the scenes, Advantages of the using the DBMS Approach and etc.

#### **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.



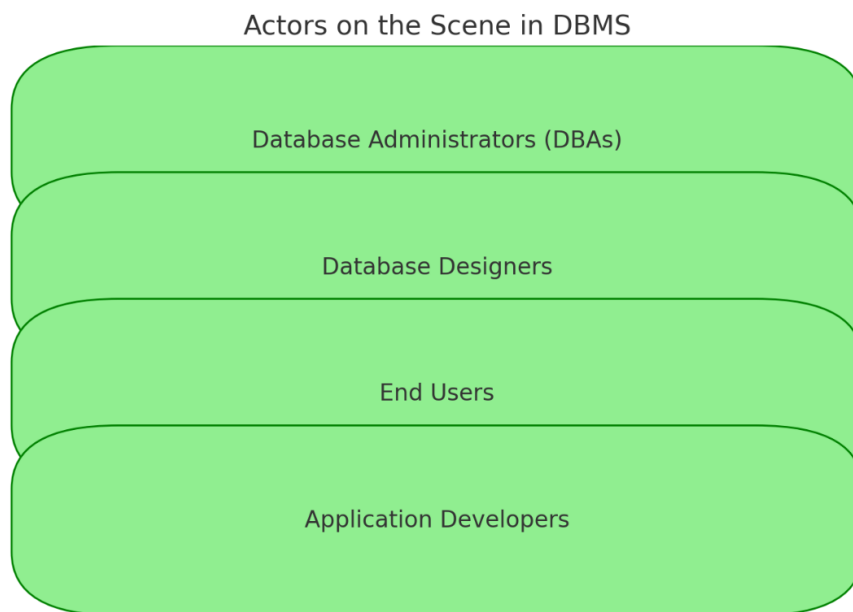
**Fig. 1.1 Characteristics of DBMS**

### 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.



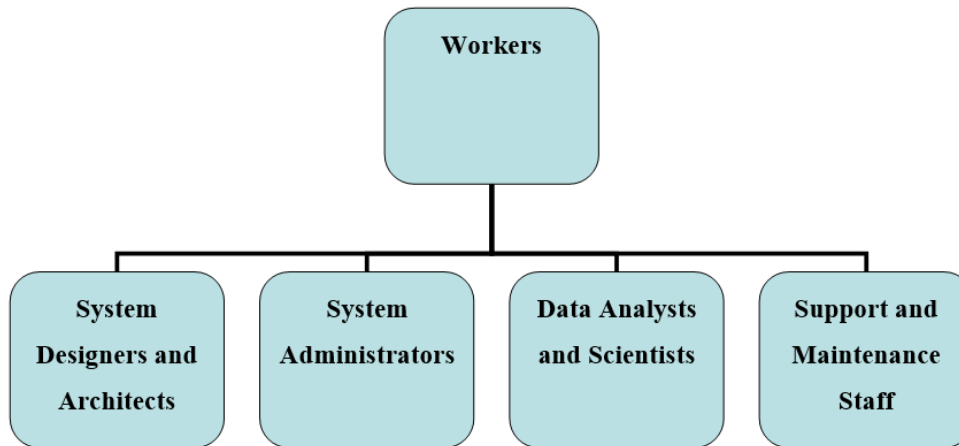
**Fig. 1.2 Actors in DBMS**

### 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.



**Fig. 1.3 Workers in DBMS**

## **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

**Table 1.1. Advantages of DBMS Concept**

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.

#### ❖ **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.

### 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. Kampa Lavanya**

## **LESSON-2**

### **DATABASE SYSTEM CONCEPTS AND ARCHITECTURE**

#### **AIMS AND OBJECTIVES:**

The primary goal of this chapter is to understand the concept of Database system concepts and Architecture. The chapter began with understanding of Data Models, Schemas and Instances, Three Schema architecture and Data Independence, Database Languages and Interfaces. 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 system concepts and architecture.

#### **STRUCTURE:**

- 2.1. Introduction**
- 2.2. Data Models**
- 2.3. Schemas and Instances**
- 2.4. Three-Schema Architecture and Data Independence**
- 2.5. Database Languages and Interfaces**
- 2.6. Centralized and Client/Server Architecture for DBMS**
- 2.7. Classification of Database Management Systems**
- 2.8. Summary**
- 2.9. Technical Terms**
- 2.10. Self-Assessment Questions**
- 2.11. Suggested Readings**

#### **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.

The chapter first covered the Data Models, Schemas and Instances, Three Schema architecture and Data Independence, Database Languages and Interfaces, Centralized and Client/Server Architecture for DBMS, and Classification of Database Management Systems.

#### **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.

```

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. 2.1 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);
```

Customers Table
CustomerID
-----
1

Orders Table
OrderID
-----
101

**Fig. 2.2 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. 2.3 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.

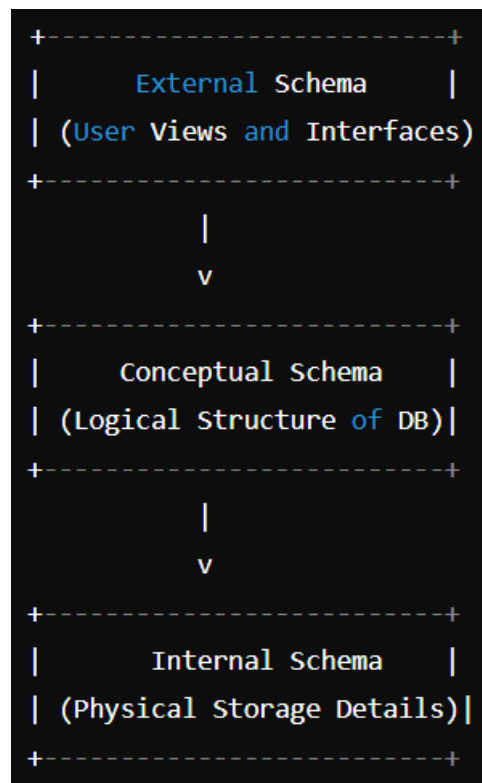


Fig. 2.4 Three-tier Architecture

#### ❖ 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 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.

## 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 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.

## 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 PhoneNumber 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: Programming interfaces for database interaction.

```
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()
```

**❖ 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 CENTRALIZED AND CLIENT/SERVER ARCHITECTURE FOR DBMS:**

### **2.6.1 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.

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.

### **2.6.2 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

### 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.

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.

**2.7 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.

**2.8 SUMMARY:**

Understanding the foundational concepts and architectures of database systems is crucial for efficient data management. Data models, schemas, instances, the three-schema architecture, and data independence provide the theoretical framework for designing and managing databases. Database languages and interfaces facilitate data interaction, while centralized and client/server architectures offer different approaches to database implementation. The classification of DBMS highlights the diversity of database systems available to meet various needs. This comprehensive overview underscores the importance and versatility of DBMS in modern information management.

**2.9 TECHNICAL TERMS:**

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

**2.10 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 Client Server Architecture

**2.11 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.

## **LESSON-3**

### **DISK STORAGE, BASIC FILE STRUCTURES AND HASHING**

#### **AIMS AND OBJECTIVES:**

The primary goal of this chapter is to understand the concept of Disk Storage, Basic File Structures and Hashing. The chapter began with understanding of Introduction, Disk storage, File structure and Hashing. The Secondary Storage Devices, Buffering of Blocks, placing file Records on Disk, Operations on Files, Files of Unordered Records, Files of Ordered Records, Hashing Techniques, Other Primary File Organizations, Parallelizing Disk Access using RAID Technology are the key concepts discussed. After completing this chapter, the student will understand Disk Storage, Basic File Structures and Hashing.

#### **STRUCTURE:**

- 3.1. Introduction**
- 3.2. Disk Storage**
- 3.3. Basic File Structures**
- 3.4. Hashing**
- 3.5. Other Primary File Organizations**
- 3.6. Parallelizing Disk Access using RAID Technology**
- 3.7. Summary**
- 3.8. Technical Terms**
- 3.9. Self-Assessment Questions**
- 3.10. Suggested Readings**

#### **3.1 INTRODUCTION:**

Efficient data storage and retrieval are fundamental aspects of database management systems (DBMS). Disk storage, basic file structures, and hashing techniques are critical for optimizing data access and ensuring database performance. This chapter delves into various aspects of disk storage, file structures, and hashing, providing a comprehensive understanding of these essential components.

The chapter first covered understanding of Introduction, Secondary Storage Devices, Buffering of Blocks, Placing file Records on Disk, Operations on Files, Files of Unordered Records, Files of Ordered Records, Hashing Techniques, Other Primary File Organizations, Parallelizing Disk Access using RAID Technology and etc.

## 3.2 DISK STORAGE:

Disk storage is a crucial component of Database Management Systems (DBMS), providing the primary medium for persistent data storage. This section delves into the components, mechanisms, and strategies involved in disk storage within a DBMS, offering insights into how data is organized, stored, and accessed efficiently.

### 3.2.1 Components of Disk Storage

#### ❖ Hard Disk Drives (HDDs):

- **Structure:** Comprises spinning disks (platters) coated with a magnetic material, read/write heads, and an actuator arm.
- **Operation:** Data is read/written as the platters spin and the read/write heads move to the appropriate location.

#### ❖ Solid-State Drives (SSDs):

- **Structure:** Utilizes flash memory with no moving parts.
- **Operation:** Data is read/written electronically, resulting in faster access times compared to HDDs.

#### ❖ Optical Disks:

- **Structure:** Includes CDs, DVDs, and Blu-ray discs, which use laser technology to read/write data.
- **Operation:** Suitable for storing large amounts of data that are infrequently accessed.

#### ❖ Magnetic Tapes:

- **Structure:** Sequential storage medium often used for archival purposes.
- **Operation:** Ideal for backup and long-term storage due to high capacity and low cost

### 3.2.2 Disk Storage Mechanisms

#### ❖ Tracks and Sectors:

- **Tracks:** Concentric circles on the disk surface where data is recorded.
- **Sectors:** Subdivisions of tracks, typically the smallest unit of data storage on a disk.

#### ❖ Cylinders:

- **Definition:** A set of tracks located at the same position on multiple platters.
- **Purpose:** Allows simultaneous reading/writing across multiple platters.

#### ❖ Blocks:

- **Definition:** The smallest unit of data transfer between the disk and memory, typically consisting of several sectors.
- **Block Size:** Determines the efficiency of disk operations, with larger blocks reducing the overhead of disk I/O operations.

### 3.2.3 Data Organization on Disk

#### ❖ Sequential Organization:

- **Description:** Records are stored in a sequential order, typically by primary key.
- **Advantages:** Efficient for sequential access and range queries.
- **Disadvantages:** Insertion and deletion can be slow due to the need to maintain order.

#### ❖ Heap (Unordered) Organization:

- **Description:** Records are placed in the order they arrive, with no particular sequence.
- **Advantages:** Fast insertion and deletion.
- **Disadvantages:** Slow search operations due to the lack of order.

#### ❖ Clustered Organization:

- **Description:** Related records are stored close to each other based on a clustering key.
- **Advantages:** Improves performance for queries accessing related records.
- **Disadvantages:** Can lead to fragmentation and requires reorganization over time.

#### ❖ Indexed Organization:

- **Description:** Uses indexes to locate records, with the data file organized based on the index key.
- **Advantages:** Fast search and retrieval using the index.
- **Disadvantages:** Overhead of maintaining the index structure.

### 3.2.4 Disk Access Mechanisms

#### ❖ Direct Access:

- **Description:** Allows random access to any part of the disk, enabling efficient data retrieval.
- **Use Cases:** Suitable for databases where records are frequently accessed in a non-sequential manner.

#### ❖ Sequential Access:

- **Description:** Data is accessed in a linear sequence, typically used for read/write operations on tapes.
- **Use Cases:** Ideal for batch processing and archival storage.

#### ❖ Caching:

- **Description:** Temporarily storing frequently accessed data in faster storage (cache) to reduce access times.

- **Benefits:** Significantly improves performance by minimizing direct disk I/O operations.

❖ **Buffering:**

- **Description:** Using memory buffers to hold data blocks during transfer between disk and memory.
- **Benefits:** Enhances performance by reducing the number of disk accesses and allowing efficient data transfer.

### 3.2.5 Disk Scheduling Algorithms

❖ **First-Come, First-Served (FCFS):**

- **Description:** Processes requests in the order they arrive.
- **Advantages:** Simple and fair.
- **Disadvantages:** Can lead to inefficient disk utilization and high seek times.

❖ **Shortest Seek Time First (SSTF):**

- **Description:** Selects the request with the shortest seek time from the current head position.
- **Advantages:** Reduces overall seek time.
- **Disadvantages:** Can cause starvation of longer requests.

❖ **SCAN (Elevator Algorithm):**

- **Description:** Moves the disk head in one direction, servicing requests, then reverses direction at the end.
- **Advantages:** Provides a more uniform wait time compared to SSTF.
- **Disadvantages:** Can lead to longer wait times for requests just missed in the current direction.

❖ **Circular SCAN (C-SCAN):**

- **Description:** Similar to SCAN, but after reaching the end, the head moves back to the beginning and starts again.
- **Advantages:** Provides a more consistent wait time than SCAN.
- **Disadvantages:** Can result in higher overall seek times.

Disk storage is a fundamental component of DBMS, providing the necessary infrastructure for persistent data storage and efficient access. Understanding the mechanisms of disk storage, including data organization, access methods, and scheduling algorithms, is crucial for optimizing database performance. By leveraging appropriate storage strategies and technologies, such as SSDs, caching, and buffering, database systems can achieve high

performance and reliability, ensuring efficient data management and retrieval. This comprehensive overview underscores the importance of disk storage in the design and operation of effective DBMS.

### 3.3 BASIC FILE STRUCTURES IN DBMS:

Basic file structures in a Database Management System (DBMS) are fundamental for organizing and managing data efficiently. The way data is stored and accessed on disk significantly impacts the performance of database operations such as insertion, deletion, searching, and updating records. This section covers the primary file structures used in DBMS: Heap Files, Sorted Files, and Indexed Files.

#### 3.3.1 Heap Files (Unordered Files)

**Description:** Heap files, also known as unordered files, store records in no particular order. They are the simplest form of file organization where records are placed in the order they arrive.

##### Characteristics:

- **Insertion:** Fast, as new records are added at the end of the file.
- **Deletion:** Typically slower, as it can create gaps that need to be managed.
- **Search:** Inefficient, as it may require scanning the entire file to find a specific record.
- **Update:** Similar to deletion, can be inefficient if the file needs to be scanned or reorganized.

##### Advantages:

- Simple to implement and manage.
- Efficient for applications with heavy insert operations and infrequent searches.

##### Disadvantages:

- Poor performance for search and update operations.
- Can lead to wasted space due to gaps left by deleted records.

##### Use Cases:

- Suitable for small datasets or applications where search and update operations are infrequent.

#### 3.2.2 Sorted Files (Ordered Files)

- Sorted files store records in a specific order based on a key attribute. This organization facilitates efficient searching and range queries.



**Characteristics:**

**Insertion:** Slower, as records need to be inserted in the correct position to maintain order. This may involve shifting subsequent records.

- **Deletion:** Easier to manage than heap files, as gaps can be filled by shifting records.
- **Search:** Efficient for binary search and range queries, significantly reducing the number of records to be scanned.
- **Update:** More efficient than heap files, but still may require shifting records to maintain order.

**Advantages:**

- Fast search and retrieval operations, particularly for range queries.
- Maintains a predictable order, facilitating efficient data access.

**Disadvantages:**

- Insertion and deletion operations can be slow due to the need to maintain order.
- Requires additional processing to keep the file sorted.

**Use Cases:**

- Ideal for applications requiring efficient search operations and range queries, such as transaction logs or time-series data.

**3.2.3 Indexed Files**

- Indexed files use an index to speed up the search and retrieval of records. An index is a separate data structure that holds pointers to records in the data file, allowing for quick access based on key values.

**Types of Indexes:**

- **Primary Index:** Built on the primary key, ensuring unique and efficient access to records.
- **Secondary Index:** Built on non-primary key fields, allowing efficient queries on those fields.

**Characteristics:**

- **Insertion:** Efficient, but may require updating the index.
- **Deletion:** Requires updating the index to remove references to deleted records.
- **Search:** Very efficient, as the index provides direct access to records.
- **Update:** Requires updating both the data file and the index.

**Advantages:**

- Significantly improves search and retrieval operations.

- Supports efficient query processing and access paths.

**Disadvantages:**

- Overhead of maintaining the index structure.
- Requires additional storage for the index.

**Use Cases:**

- Suitable for large datasets with frequent search operations, such as relational databases and content management systems.

Understanding basic file structures is crucial for optimizing database performance and ensuring efficient data management. Heap files offer simplicity and efficiency for insert-heavy applications but fall short in search and update operations. Sorted files provide efficient search capabilities but require additional processing to maintain order. Indexed files offer the best performance for search and retrieval operations, albeit with the overhead of maintaining the index. Choosing the appropriate file structure depends on the specific requirements of the application, such as the frequency of insertions, deletions, searches, and updates. This comprehensive overview highlights the importance of selecting the right file structure for achieving optimal database performance.

**3.4. HASHING:**

Hashing is a powerful technique used in Database Management Systems (DBMS) to efficiently locate and retrieve data records based on search keys. By converting a search key into a unique address in the database using a hash function, hashing provides constant-time complexity for insert, delete, and search operations under ideal conditions. This section covers the fundamentals of hashing, including hash functions, collision resolution techniques, and dynamic hashing methods.

**Components:**

- **Hash Table:** The data structure that stores the records. It is an array of fixed size, where each index corresponds to a bucket.
- **Hash Function:** A function that converts a search key into an index in the hash table.
- **Buckets:** Slots in the hash table where records are stored. Each bucket can hold one or more records.

**3.4.1 Hash Function**

A good hash function distributes keys uniformly across the hash table to minimize collisions and ensure efficient data retrieval.

It can be estimated in three ways

**❖ Division-Remainder Method:**

- **Description:** The key is divided by a prime number, and the remainder is used as the hash value.

- **Formula:**  $h(\text{key}) = \text{key} \% \text{prime\_number}$
- **Example**

```
def hash_function(key, prime_number):
    return key % prime_number
```

#### ❖ **Multiplicative Hashing:**

- **Description:** The key is multiplied by a constant fraction, and the fractional part is used as the hash value.
- **Formula:**  $h(\text{key}) = \text{floor}(N * (\text{key} * A \% 1))$  where  $A$  is a constant ( $0 < A < 1$ ) and  $N$  is the size of the hash table.
- **Example**

```
def hash_function(key, A, N):
    return int(N * ((key * A) % 1))
```

#### ❖ **Universal Hashing:**

- Uses a family of hash functions to minimize the probability of collisions.
- **Example**

```
import random

def universal_hash_function(key, p, m, a, b):
    return ((a * key + b) % p) % m

# p is a prime number larger than the maximum key value
# m is the size of the hash table
# a and b are randomly chosen constants
```

### 3.4.2 Collision Resolution

Collisions occur when two keys hash to the same index in the hash table. Various techniques are used to handle collisions effectively.

- ❖ **Linear Probing:** When a collision occurs, the next available slot is checked sequentially.

```
def linear_probing(hash_table, key, hash_value):
    while hash_table[hash_value] is not None:
        hash_value = (hash_value + 1) % len(hash_table)
    return hash_value
```

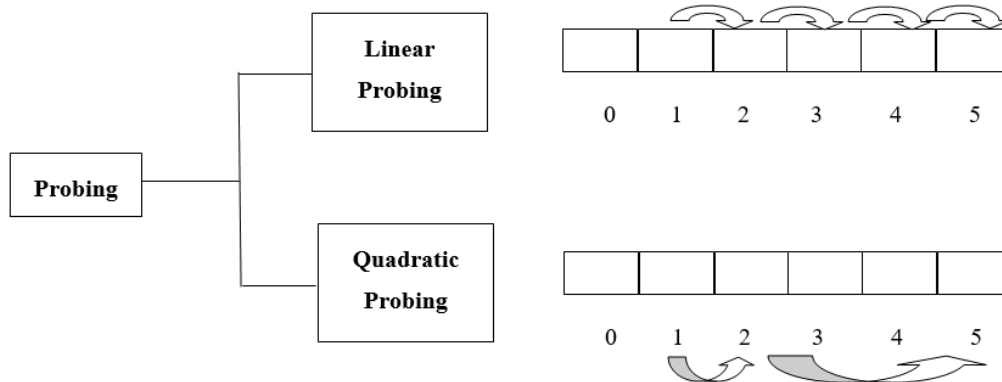
- ❖ **Quadratic Probing:** Similar to linear probing but uses a quadratic function to find the next slot.

```
def quadratic_probing(hash_table, key, hash_value):
```

```

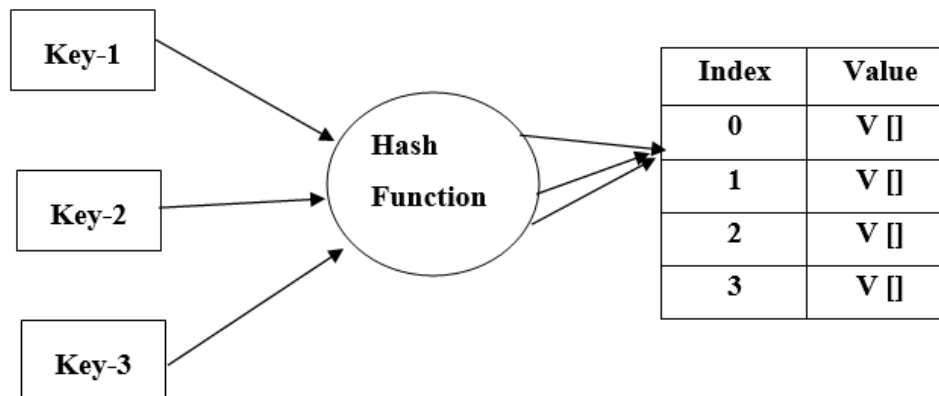
i = 1
while hash_table[hash_value] is not None:
    hash_value = (hash_value + i**2) % len(hash_table)
    i += 1
return hash_value

```



**Fig. 3.1 Conceptual view of Linear and Quadratic Probing**

- ❖ **Double Hashing:** Uses a second hash function to determine the step size for resolving collisions.



**Fig. 3.2 Doble Hashing overview**

```

def double_hashing(hash_table, key, hash_value, hash_function2):
    step_size = hash_function2(key)
    while hash_table[hash_value] is not None:
        hash_value = (hash_value + step_size) % len(hash_table)
    return hash_value
    hash_value = (hash_value + step_size) % len(hash_table)
    return hash_value

```

### ❖ Chaining:

- **Description:** Each bucket in the hash table points to a linked list of records that hash to the same index.
- **Example**

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None

class HashTable:
    def __init__(self, size):
        self.table = [None] * size

    def insert(self, key, value, hash_function):
        hash_value = hash_function(key)
        if self.table[hash_value] is None:
            self.table[hash_value] = Node(key, value)
        else:
            current = self.table[hash_value]
            while current.next:
                current = current.next
            current.next = Node(key, value)
```

### 3.4.3 Dynamic Hashing

Dynamic hashing methods allow the hash table to grow or shrink dynamically as the number of records changes.

#### 1. Extendible Hashing:

- **Description:** Uses a directory of pointers to bucket addresses, allowing the hash table to expand dynamically.
- **Advantages:** Provides efficient space utilization and handles dynamic changes in the dataset.

#### 2. Linear Hashing:

- **Description:** Allows incremental expansion of the hash table without a directory, redistributing entries as needed.
- **Advantages:** Simplifies the rehashing process and maintains efficient access times.

### 3.5 OTHER PRIMARY FILE ORGANIZATIONS IN DBMS:

In addition to the basic file structures such as heap files, sorted files, and indexed files, there are several other primary file organization methods used in Database Management Systems (DBMS) to optimize data storage and retrieval. These methods include clustered files, multilevel indexing, and tree-based structures like B-trees and B+-trees.

#### 3.5.1. Clustered Files

- Clustered file organization stores related records together on disk to minimize the number of disk I/O operations during retrieval.

##### Characteristics:

- **Clustering Key:** A key used to group related records together.
- **Physical Proximity:** Records that are frequently accessed together are stored close to each other on disk.

##### Advantages:

- Improved performance for queries that retrieve related records.
- Reduced disk I/O for range queries and joins.

##### Disadvantages:

- Insertion and deletion can be complex and may require reorganization of records.
- Clustering is less effective if the access patterns change frequently.

##### Use Cases:

- Suitable for databases with predictable access patterns, such as geographic information systems or data warehousing.

#### 3.5.2 Multilevel Indexing

- Multilevel indexing uses multiple levels of indexes to improve search efficiency, particularly for large databases.

##### Types:

- **Single-Level Index:** A simple index on a file.
- **Multilevel Index:** An index on an index, used to reduce the number of disk I/O operations required to locate a record.

##### Characteristics:

- **Hierarchical Structure:** Higher-level indexes point to lower-level indexes, ultimately pointing to the data blocks.

- **Balanced Tree:** Ensures that the index remains balanced, providing consistent search times.

**Advantages:**

- Efficient search operations, reducing the number of disk accesses.
- Can handle very large datasets effectively.

**Disadvantages:**

- Overhead of maintaining multiple levels of indexes.
- Complexity in updating indexes during insertions and deletions.

**Use Cases:**

- Suitable for large databases with high query performance requirements, such as financial systems or e-commerce platforms.

**3.5.3 B-Trees and B+-Trees**

- B-trees and B+-trees are balanced tree structures used for indexing in databases, ensuring logarithmic search times and efficient range queries.

**Characteristics of B-Trees:**

- **Balanced Structure:** Maintains a balanced tree by redistributing nodes during insertions and deletions.
- **Nodes:** Each node contains multiple keys and pointers to child nodes.
- **Order:** The maximum number of children a node can have.

**Characteristics of B+-Trees:**

- **Leaf Nodes:** All records are stored at the leaf nodes, which are linked together for sequential access.
- **Internal Nodes:** Only store keys and pointers, improving search efficiency.
- **Balanced Structure:** Ensures that the tree remains balanced, providing consistent performance.

**Advantages:**

- Efficient search, insertion, and deletion operations.
- B+-trees provide efficient sequential access and range queries.

**Disadvantages:**

- Overhead of maintaining the tree structure during updates.
- Complexity in implementation and management.

**Use Cases:**

- Suitable for databases requiring efficient search and range queries, such as indexing in relational databases and file systems.

Various primary file organization methods in DBMS provide different trade-offs in terms of performance, complexity, and suitability for specific types of queries and workloads. Clustered files enhance performance for related record queries, multilevel indexing improves search efficiency for large databases, and tree-based structures like B-trees and B+-trees offer balanced and efficient indexing. Hash-based indexing provides fast equality searches, while bitmap indexes are ideal for low-cardinality attributes in read-heavy environments. Understanding these file organization methods enables database designers to select the appropriate structure based on the application's specific requirements, ensuring optimal performance and efficient data management.

### **3.6 PARALLELIZING DISK ACCESS USING RAID TECHNOLOGY IN DBMS:**

RAID (Redundant Array of Independent Disks) technology is used to improve the performance, reliability, and fault tolerance of disk storage in Database Management Systems (DBMS). By combining multiple physical disks into a single logical unit, RAID allows for parallel data access and enhances overall system efficiency. This section covers various RAID levels, their characteristics, advantages, disadvantages, and use cases.

#### **Benefits of RAID in DBMS**

##### **1. Improved Performance:**

- By striping data across multiple disks, RAID allows for parallel read and write operations, significantly enhancing data access speeds.

##### **2. Fault Tolerance:**

- Redundancy in RAID configurations (such as RAID 1, RAID 5, and RAID 6) ensures that data can be recovered in the event of disk failures, providing high availability and reliability.

##### **3. Increased Storage Capacity:**

- Combining multiple disks into a single logical unit allows for greater overall storage capacity, which can be expanded as needed.

##### **4. Load Balancing:**

- Distributing data across multiple disks helps balance the load, preventing any single disk from becoming a bottleneck.

### **3.7 SUMMARY:**

Disk storage is a fundamental component of DBMS, providing the necessary infrastructure for persistent data storage and efficient access. Understanding the mechanisms of disk storage, including data organization, access methods, and scheduling algorithms, is crucial for optimizing database performance. By leveraging appropriate storage strategies and



technologies, such as SSDs, caching, and buffering, database systems can achieve high performance and reliability, ensuring efficient data management and retrieval. This comprehensive overview underscores the importance of disk storage in the design and operation of effective DBMS. In addition file structure, hashing, other parallel mechanisms i.e., RAID also discussed in this chapter.

### **3.8 TECHNICAL TERMS:**

Disk storage, File Structure, B Tree, B++ Tree, RAID, Parallelism Schema, Secondary storage, Hard disk, Hashing, Hash Function, Chaining and etc.

### **3.9 SELF ASSESSMENT QUESTIONS:**

#### **Essay Questions:**

- 1) Illustrate about Disk storages
- 2) Describe about Hashing in DBMS
- 3) Explain about File Structure in DBMS.

#### **Short Notes:**

- 1) Write about RAID
- 2) Define Hash Function
- 3) List collision resolution techniques.

### **3.10 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.

**Dr. Kampa Lavanya**

## **LESSON-4**

### **INDEXING STRUCTURES FOR FILES**

#### **AIMS AND OBJECTIVES:**

The primary goal of this chapter is to understand the concept of Indexing Structures for Files. The chapter began with understanding of Types of Single-Level Ordered Indexes, Multilevel Indexes and Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup> Trees, Indexes on Multiple Keys, Other Types of Indexes. After completing this chapter, the student will understand Indexing Structures for Files.

#### **STRUCTURE:**

- 4.1. Introduction**
- 4.2. Indexing Structure**
- 4.3. Indexing Methods**
- 4.4. Other Types of Indexes.**
- 4.5. Summary**
- 4.6. Technical Terms**
- 4.7. Self-Assessment Questions**
- 4.8. Suggested Readings**

#### **4.1 INTRODUCTION:**

Efficient data retrieval is a cornerstone of effective database management. Indexing structures are vital tools in Database Management Systems (DBMS) for accelerating query performance. This chapter provides a comprehensive overview of various indexing techniques, including single-level ordered indexes, multilevel indexes, dynamic multilevel indexes using B-trees and B<sup>+</sup> trees, indexes on multiple keys, and other types of indexes.

The chapter first covered began with understanding Indexing. Later discussed Types of Single-Level Ordered Indexes, Multilevel Indexes and Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup> Trees, Indexes on Multiple Keys, and Other Types of Indexes

#### **4.2 INDEXING STRUCTURE:**

- Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
- The index is a type of data structure. It is used to locate and access the data in a database table quickly.

Indexes can be created using some database columns.



**Fig. 4.1 Structure of Index**

- The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
- The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

### **4.3 INDEXING METHODS:**

Indexing allows for efficient access to the database. Number of indexing methods, including single-level ordered indexes, multilevel indexes, dynamic multilevel indexes using B-trees and B+ trees, indexes on multiple keys, and other types of specialized indexes.

#### **4.3.1 Single-Level Ordered Indexes**

This section indexing methods includes the few of techniques which are discussed below:

##### **❖ Primary Indexes**

- A primary index is an ordered file with two fields: the primary key and a pointer to the corresponding data block.

##### **Characteristics:**

- **Dense Index:** Contains an index entry for every record.
- **Sparse Index:** Contains index entries for only some records, typically one per block.

##### **Advantages:**

- Faster access to data due to direct pointers.
- Suitable for sequentially ordered data.

##### **Disadvantages:**

- Requires additional storage.
- Maintenance overhead if the primary key is updated.

##### **Example:**

- Indexing books in a library by ISBN.

##### **❖ Secondary Indexes**

- A secondary index provides efficient access based on non-primary key attributes.

**Characteristics:**

- It can be dense or sparse.
- May include duplicate key values.

**Advantages:**

- Enhance performance for queries on non-primary key attributes.
- Allows efficient access for complex queries.

**Disadvantages:**

- Additional storage and maintenance required.
- Increased complexity with multiple secondary indexes.

**Example:**

- Indexing books by author name.

**❖ Clustering Indexes**

- A clustering index is created on data sorted on a non-primary key. The data file is organized based on this clustering field.

**Characteristics:**

- Improves performance for queries involving the clustering field.
- Only one clustering index per table.

**Advantages:**

- Efficient for range queries.
- Reduces data retrieval time.

**Disadvantages:**

- Complex insertion and deletion.
- Requires periodic reorganization.

**Example:**

- Indexing students by graduation year.

**4.3.2 Multilevel Ordered Indexes**

This section indexing methods includes the few of techniques which are discussed below:

**❖ Single-Level Indexes**

- Single-level indexes involve a single layer of indexing, which can become inefficient as the database grows.

**Challenges:**

- Inefficient for large datasets.
- Limited scalability.

**Example:**

- Indexing a small employee dataset.

**❖ Multilevel Indexes**

- Multilevel indexes use a hierarchical structure with multiple index levels. The top level points to lower-level indexes.

**Characteristics:**

- Hierarchical structure.
- Reduces disk accesses for searches.

**Advantages:**

- Efficient for large datasets.
- Scalable.

**Disadvantages:**

- Increased maintenance complexity.
- Requires additional storage.

**Example:**

- Indexing a large customer dataset.

**4.3.3 Dynamic Multilevel Indexes**

This section indexing methods includes the few of techniques which are discussed below:

**❖ B-Trees**

- A B-tree is a balanced tree that maintains sorted data and allows logarithmic time searches, insertions, and deletions.

**Key Features of a B-Tree**

1. **Balanced Tree:** All leaf nodes are at the same level.
2. **Dynamic Structure:** Automatically adjusts its height during insertions and deletions.
3. **Node Properties:**
  - Each node contains multiple keys and pointers.
  - Keys in a node are stored in sorted order.
  - Internal nodes contain pointers to child nodes, separating key ranges.
4. **Search Efficiency:** Binary search is performed within each node.
5. **Height Minimization:** B-Trees minimize the height by allowing nodes to hold more keys, reducing disk I/O.

**Structure of a B-Tree**

- **Order (m):** Maximum number of children a node can have.

- Each node (except the root) must have at least  $\lceil m/2 \rceil$  children.
- A node can contain at most  $m-1$  keys.
- **Keys:** Divide the value range into subranges for child pointers.

### ➤ Search in a B-Tree

**Objective:** Find if a specific key exists and retrieve the associated record.

**Algorithm:**

1. Start at the root node.
2. Perform a **binary search** within the current node to locate the key or find the range where the key might exist.
3. If the key is found in the current node:
  - Return the record (or success).
4. If the key is not found in the current node:
  - Traverse the appropriate child node (based on the range of keys).
  - Repeat steps 2-4.
5. If a leaf node is reached without finding the key:
  - Return “key not present”.

**Complexity:**

- Search involves traversing  $O(\log_m N)$  levels, where  $N$  is the number of keys and  $m$  is the order.

### ➤ Insertion in a B-Tree

**Objective:** Insert a new key into the B-Tree while maintaining its balanced structure.

**Algorithm:**

1. **Search for the Correct Position:**
  - Traverse the tree to find the appropriate leaf node where the key should be inserted.
2. **Insert the Key:**
  - Add the key to the node in sorted order.
  - If the node has space ( $< m-1$  keys), the process ends.
3. **Handle Overflow:**
  - If the node overflows ( $\geq m$  keys):
    - Split the node into two nodes, each containing  $\lceil m/2 \rceil$  keys.
    - Propagate the middle key to the parent node.

#### 4. Propagate Changes Upwards:

- If the parent node also overflows, repeat the splitting process recursively up the tree.

#### 5. Adjust the Root:

- If the root node overflows, create a new root with two children, increasing the tree's height.

#### Complexity:

- Search  $O(\log mN)$  + Potential rebalancing.

#### ➤ Deletion in a B-Tree

**Objective:** Remove a key from the B-Tree while ensuring it remains balanced.

#### Algorithm:

##### 1. Search for the Key:

- Locate the key in the tree.
- If the key is not found, return "key not present."

##### 2. Delete the Key:

- If the key is in a leaf node:
  - Remove the key.
  - If the node still has enough keys ( $\geq \lfloor m/2 \rfloor - 1 \geq \lceil m/2 \rceil - 1$ ), the process ends.
- If the key is in an internal node:
  - Replace it with the predecessor (largest key in the left subtree) or successor (smallest key in the right subtree) and delete the replacement key from the corresponding leaf.

##### 3. Handle Underflow:

- If a node has too few keys ( $< \lfloor m/2 \rfloor - 1 < \lceil m/2 \rceil - 1 < \lfloor m/2 \rfloor - 1$ ):
  - **Borrow from Sibling:**
    - Redistribute keys with a sibling (left or right) to balance the nodes.
  - **Merge with Sibling:**
    - If redistribution is not possible, merge the node with a sibling and remove a key from the parent.
- Repeat the underflow handling process recursively up the tree.

##### 4. Adjust the Root:

- If the root becomes empty (only one child remains), make the child the new root, reducing the tree's height.

**Complexity:**

- Search  $O(\log_m N)$  + Potential rebalancing.

**Advantages:**

- Efficient search, insertion, and deletion.
- Consistent performance.

**Disadvantages:**

- Maintenance overhead.
- Complex implementation.

**Example:**

- Indexing product records in an e-commerce system.

**❖ B+ Trees**

- A B+ tree is an extension of the B-tree where all data is stored in leaf nodes, and internal nodes only store keys.

**Operations****➤ Search:**

- Traverse from root, comparing keys.
- At each node, use binary search for faster key location.

**Algorithm:**

1. Start at the root node.
2. Perform a binary or sequential search within the current node to find:
  - The range where the key may exist.
  - A pointer to the child node where the search should continue.
3. Traverse down to the next node based on the pointer.
4. If a leaf node is reached:
  - Search for the key in the leaf.
  - If found, return the corresponding record.
  - If not found, return "key not present."

**Complexity:**

- **Search Time:**  $O(\log_m N)$  where  $N$  is the number of keys and  $m$  is the order.

**➤ Insertion:**

- Traverse to the appropriate leaf node.
- Insert key into the sorted list of keys in the node.
- If the node overflows, split and propagate changes upward.



**Algorithm:****1. Search for the Correct Leaf:**

- Traverse the tree starting from the root to find the appropriate leaf node for the key.

**2. Insert into the Leaf:**

- Add the key in sorted order to the leaf.
- If the leaf has space ( $m-1 < m-1 < m-1$  keys), the process ends.

**3. Handle Overflow:**

- If the leaf overflows ( $\geq m \geq m \geq m$  keys), split the leaf into two nodes:
  - Create a new leaf node and redistribute keys evenly.
  - Propagate the middle key to the parent node.
- If the parent overflows, repeat the splitting process recursively up the tree.

**4. Adjust Pointers:**

- Update sibling pointers for the linked list of leaf nodes.

**Complexity:**

- **Insertion Time:**  $O(\log_m N)$  for search + potential updates.

**➤ Deletion:**

- Locate the key.
- Replace it (if internal) with the predecessor/successor key.
- If underflow occurs (fewer than  $\lceil m/2 \rceil$  keys), rebalance by borrowing from siblings or merging nodes.

**Algorithm:****1. Search for the Key:**

- Locate the key to be deleted in the leaf node.

**2. Delete from the Leaf:**

- Remove the key from the leaf node.
- If the leaf node still has enough keys ( $\geq \lceil m/2 \rceil - 1 \geq \lceil m/2 \rceil - 1$ ), the process ends.

**3. Handle Underflow:**

- If the leaf node has too few keys:
  - **Borrow from Sibling:** Redistribute keys with a sibling node (left or right).

- **Merge with Sibling:** If redistribution is not possible, merge the underflowing node with a sibling and update the parent node.

#### 4. Update Parent:

- If a parent node becomes deficient due to a merge, handle underflow recursively up the tree.

#### 5. Special Case:

- If the root becomes empty, replace it with its only child, reducing the tree's height.

**Complexity: Deletion Time:**  $O(\log_m N)$  for search + potential updates.

#### Advantages:

- Faster search and retrieval.
- Efficient range queries.

#### Disadvantages:

- Similar maintenance complexity to B-trees.
- Additional storage for linked leaf nodes.

#### Example:

- Indexing transaction records in a financial system.

### 4.3.4 Indexes on Multiple Keys

This section indexing methods includes the few of techniques which are discussed below:

#### ❖ Composite Indexes

- Composite indexes are created on multiple columns of a table, allowing efficient access based on column combinations.

#### Characteristics:

- Index entries are sorted by column combinations.
- Supports multi-column queries.

#### Advantages:

- Improves multi-column query performance.
- Reduces need for multiple single-column indexes.

#### Disadvantages:

- Additional storage and maintenance.
- Increased complexity.

#### Example:

- Indexing employees by department and job title.

### ❖ Covering Indexes

- A covering index includes all columns needed for a query, allowing the query to be answered entirely from the index.

#### Characteristics:

- Index entries contain all required query columns.
- Reduces disk accesses.

#### Advantages:

- Improves query performance.
- Avoids table lookups.

#### Disadvantages:

- Additional storage and maintenance.
- Large and complex indexes.

#### Example:

- Indexing sales records by date, product ID, and quantity.

## 4.4 OTHER TYPES OF INDEXES:

Additional indexing method is discussed in this section which is given below:

1. Bitmap Indexes
2. Full-Text Indexes
3. Special Indexes

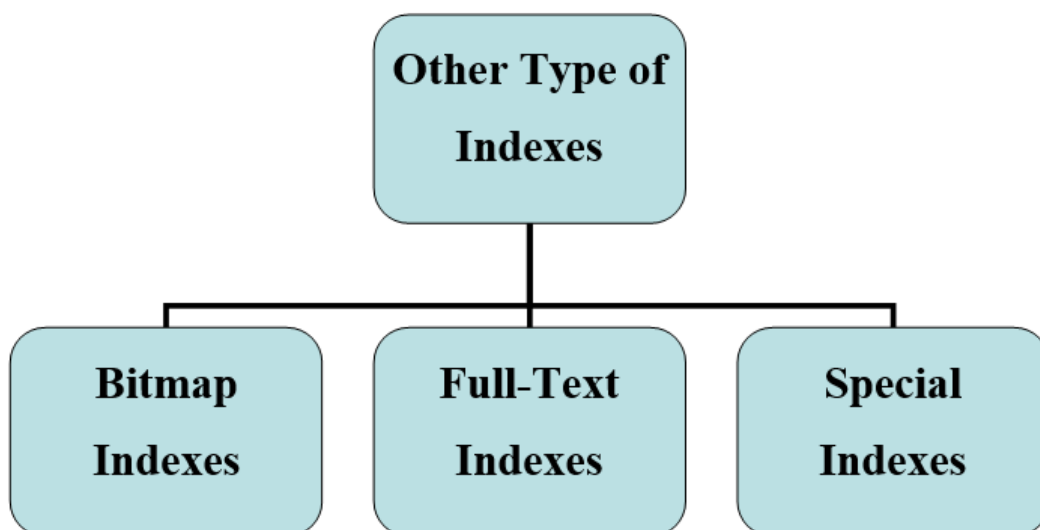


Fig. 4.2 Classification of Other Type of Indexes

#### 4.4.1 Bitmap Indexes

- Bitmap indexes use bit arrays to represent the presence or absence of a value for an attribute, efficient for low-cardinality attributes.

**Characteristics:**

- Each value represented by a bitmap.
- Efficient for read-heavy workloads.

**Advantages:**

- Fast query processing.
- Reduced storage for low-cardinality attributes.

**Disadvantages:**

- Not suitable for high-cardinality attributes.
- Maintenance overhead.

**Example:**

- Indexing customer gender or marital status.

#### 4.4.2 Full-Text Indexes

- Full-text indexes allow efficient searching of text data, supporting keyword searches and phrase matching.

**Characteristics:**

- Indexes words or phrases in text fields.
- Supports complex text queries.

**Advantages:**

- Efficient text search and retrieval.
- Advanced search features.

**Disadvantages:**

- Significant storage and maintenance.
- Handling natural language complexity.

**Example:** Indexing product descriptions in an online store.

#### 4.4.3 Spatial Indexes

- Spatial indexes are used for indexing spatial data, enabling efficient querying of geometric and geographic data.

**Characteristics:**

- Supports spatial queries.
- Uses structures like R-trees and quad-trees.

**Advantages:**

- Efficient spatial data retrieval.
- Supports complex spatial queries.

**Disadvantages:**

- Complex implementation and maintenance.
- Additional storage for spatial structures.

**Example:**

- Indexing geographic locations in a GIS system.

**4.5 SUMMARY:**

Indexing structures are essential for optimizing data retrieval in DBMS. Various indexing techniques, including single-level ordered indexes, multilevel indexes, dynamic multilevel indexes using B-trees and B+ trees, indexes on multiple keys, and other specialized indexes, offer different advantages and trade-offs. Understanding these techniques allows database designers to choose the most appropriate indexing strategy based on specific application requirements, ensuring efficient and effective data management. This comprehensive overview highlights the importance of indexing structures in achieving high performance and scalability in modern database systems.

**4.6 TECHNICAL TERMS:**

B Tree, B++ Tree, Indexing, Single level indexing, Multi-level indexing, Special Indexing.

**4.7 SELF ASSESSMENT QUESTIONS:****Essay Questions:**

- 1) Illustrate about Single level indexing?
- 2) Describe about Indexing methods?
- 3) Explain about Special Indexing methods?

**Short Notes:**

- 1) Write about B Trees?
- 2) Define Multi-level Indexing?
- 3) List advantages and disadvantages full text indexing?

**4.8 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.

## **LESSON-5**

### **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.

#### **STRUCTURE:**

- 5.1. Introduction**
- 5.2. Entity Types and Entity Sets**
- 5.3. Attributes and Keys**
- 5.4. Relationship Types, Relationship Sets and Roles**
- 5.5. Structural Constraints**
- 5.6. Weak Entity Types**
- 5.7. Relationship Types of Degree Higher than Two in DBMS**
- 5.8. Refining the ER Design for the Company Database**
- 5.9. Summary**
- 5.10. Technical Terms**
- 5.11. Self-Assessment Questions**
- 5.12. Suggested Readings**

#### **5.1 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.

## 5.2 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.'

### 5.2.1 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.

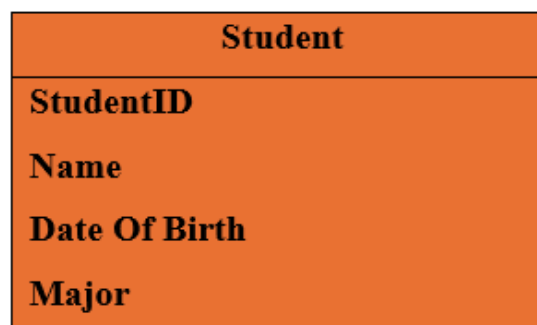


Fig. 5.1 Entity Type: Student

### 5.2.2 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:

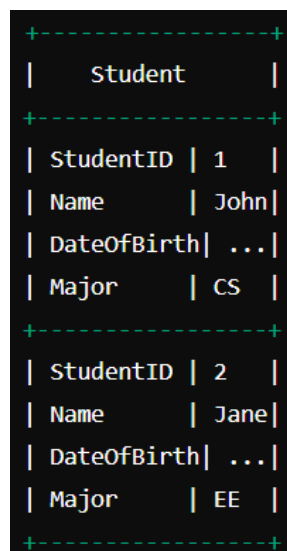
- **Homogeneous Collection:** Contains entities of the same entity type.
- **Dynamic:** The number of entities in an entity set can change over time as entities are added, modified, or removed.

#### 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.



Student			
StudentID	1	Name	John
DateOfBirth	...	Major	CS
StudentID	2	Name	Jane
DateOfBirth	...	Major	EE

Fig. 5.2 Entity Set of Student Object

### 5.3 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.

#### 5.3.1 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.

**5.3.2 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:**

- **Primary Key:** 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.
- **Composite Key:** 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.

- **Candidate Key:** 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.
  - **Alternate Key:** 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.
- **Foreign Key:** 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.
- **Super Key:** 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.
  - **Unique Key:** 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:**
    - Like 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:

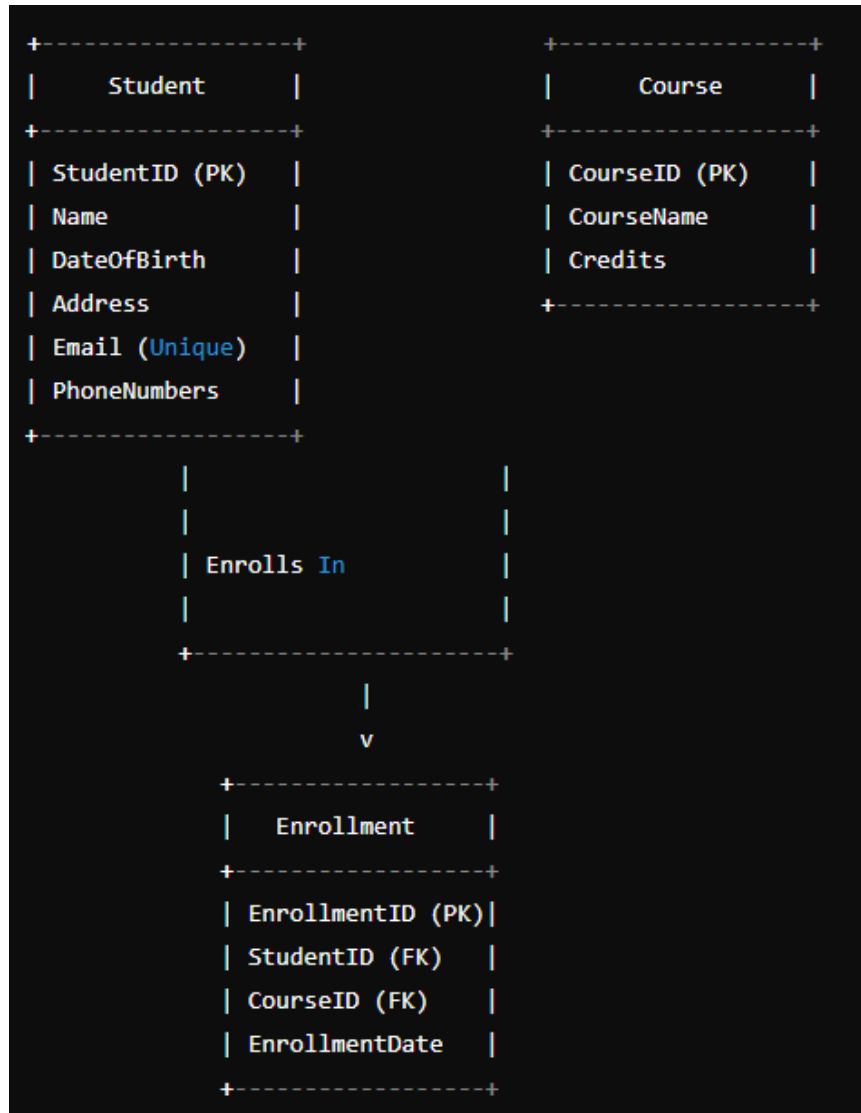
- Primary keys are underlined in entity representations.
- Foreign keys are represented with a dashed line connecting the two related entities.

### 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. 5.3 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.

## 5.4 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.

### 5.4.1 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.

### 5.4.2 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.

### 5.4.3 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.

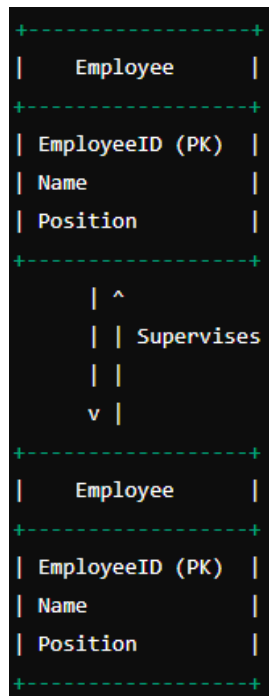
### Employee Database:

#### Entities:

- Employee (EmployeeID, Name, Position)

#### Relationships:

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



**Fig. 5.4 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.

## 5.5 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.

**5.5.1 Domain Constraints:** 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)
);

```

**5.5.2 Entity Integrity Constraints:** 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)  
);
```

**5.5.3 Referential Integrity Constraints:** 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)  
);
```

**5.5.4 Unique Constraints:** 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  
);
```

**5.5.5 Check Constraints:** 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)  
);
```

**5.5.6 Not Null Constraints:** Ensure that a column cannot have null values.

- **Example:** An employee's name cannot be null.
- **Implementation:** Not null constraints

```
CREATE TABLE Employees (  
    name VARCHAR(50) NOT NULL,  
    employee_id INT PRIMARY KEY,  
    salary DECIMAL(10, 2),  
    CHECK (salary >= 1500)  
);
```

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

**5.5.7 Default Constraints:** 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'  
);
```

## 5.6 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.

### 5.6.1 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.



### 5.6.2 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.

### 5.7 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)  
);
```

## 5.8 REFINING THE ER DESIGN FOR THE COMPANY DATABASE:

The steps for refining the Entity-Relationship (ER) design for the **COMPANY database**. It provides guidance on identifying initial entity types, attributes, and relationships, as well as refining the design to ensure it aligns with the requirements.

### Key Elements in Refining the ER Design:

**1. Identifying Initial Entity Types:** Common entities for a COMPANY database include:

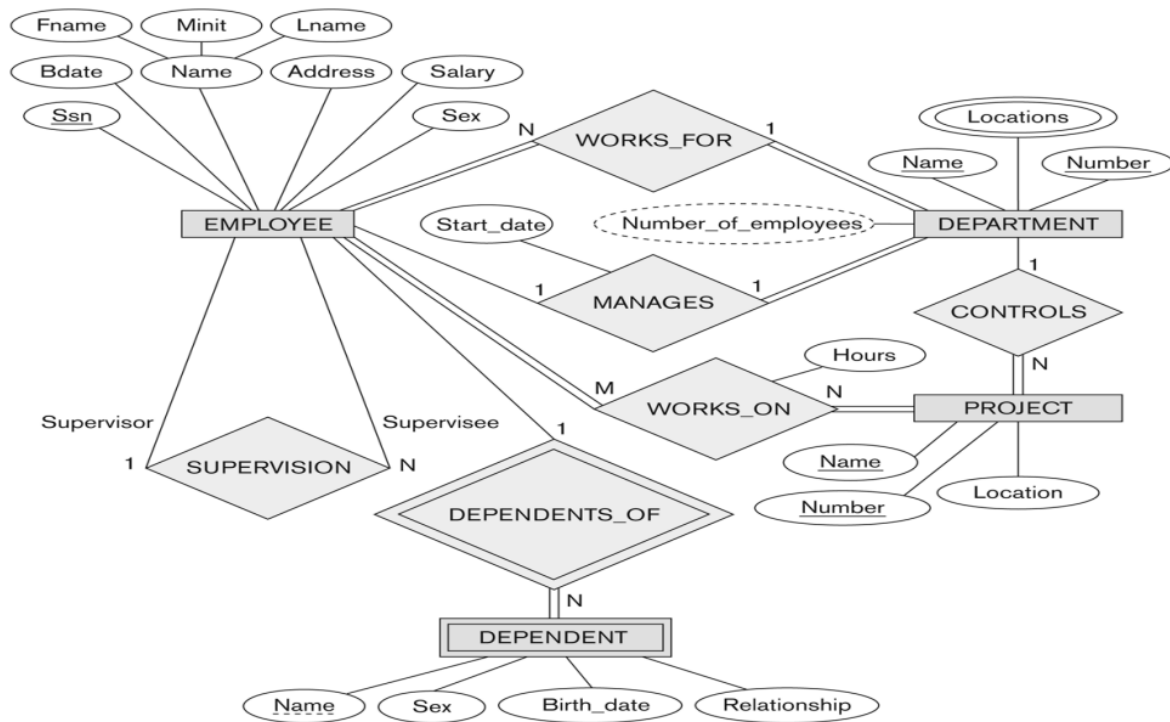
- **Employee:** Attributes like Name, SSN, Address, and Job\_Title.
- **Department:** Attributes like Dept\_Name, Dept\_Number, and Manager\_SSN.
- **Project:** Attributes such as Project\_Name, Project\_Number, and Location.
- **Dependent:** Attributes such as Dependent\_Name, Relationship, and DOB.

**2. Adding Relationships:**

- **Works\_For:** Connects Employee and Department.
- **Manages:** Links a Manager (Employee) to their Department.
- **Works\_On:** Links Employee and Project with an additional attribute for Hours\_Worked.
- **Dependent\_of:** Links Dependent to an Employee.

**3. Design Refinements:**

- Ensuring the diagram captures all constraints, such as cardinalities and participation (e.g., every employee must work in a department, while some may not work on projects).
- Introducing composite or multivalued attributes if required.
- Handling recursive relationships (e.g., an employee supervising another employee).



**Fig. 5.5 Refined ER diagram for Company Database**

## 5.9 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.

## 5.10 TECHNICAL TERMS:

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

## 5.11 SELF ASSESSMENT QUESTIONS:

### Essay questions:

- 1) Illustrate about Entity and Entity Types?
- 2) Describe about Relationship Types?
- 3) Explain about Company Database ER Model?
- 4) Illustrate Refined ER Diagram for Company Database?

**Short Notes:**

- 1) Write about Weak Entity?
- 2) Define Key Constraints?
- 3) Explain about Cardinality?

**5.12 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. U. Surya Kameswari**

## **LESSON-6**

### **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.

#### **STRUCTURE:**

- 6.1. Introduction**
- 6.2. Subclasses, Superclasses and Inheritance**
- 6.3. Specialization and Generalization**
- 6.4. Constraints and Characteristics of Specialization and Generalization Hierarchies**
- 6.5. Modelling of Union Types Using Categories**
- 6.6. Example University EER Schema**
- 6.7. Design Choices and Formal Definitions**
- 6.8. Summary**
- 6.9. Technical Terms**
- 6.10. Self-Assessment Questions**
- 6.11. Suggested Readings**

#### **6.1. 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.

## 6.2 SUBCLASSES, SUPERCLASSES, AND INHERITANCE:

### 6.2.1 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 DateOfBirth 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.

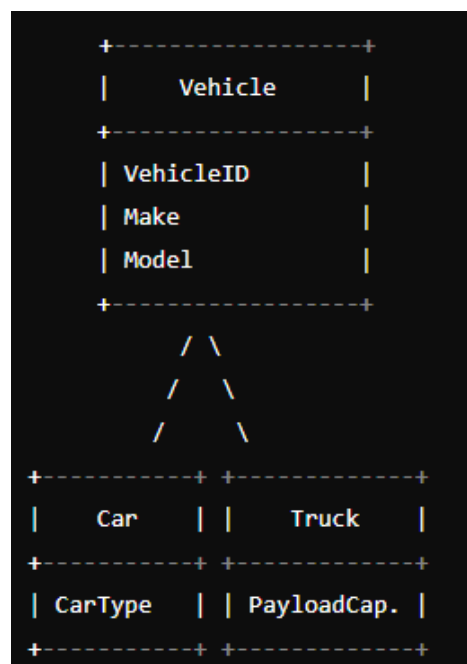


Fig. 6.1 Subclass in EER Diagram

### 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.

#### 6.2.2 Super classes

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.

#### 6.2.3 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.

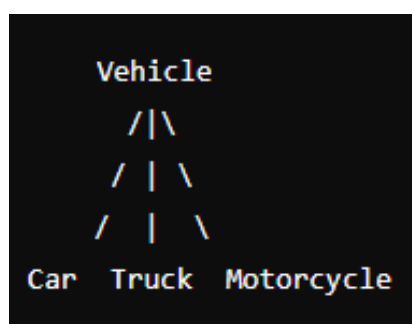


Fig. 6.2 Inheritance in EER Diagram

#### 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

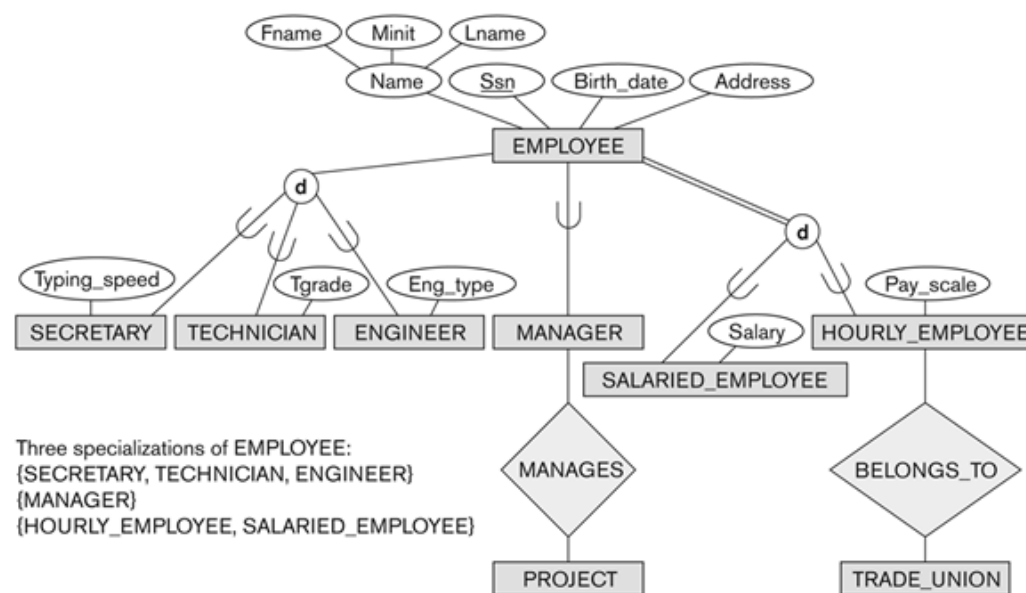
## 4. Establish Inheritance Relationships:

- 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.



**Fig. 6.3 EER diagram with sub classes and specialization**

An Enhanced Entity-Relationship (EER) diagram for an Employee Database using subclasses and specialization adds complexity and granularity to represent hierarchical and specialized data relationships.

### 6.3. SPECIALIZATION AND GENERALIZATION:

#### 6.3.1 Specialization

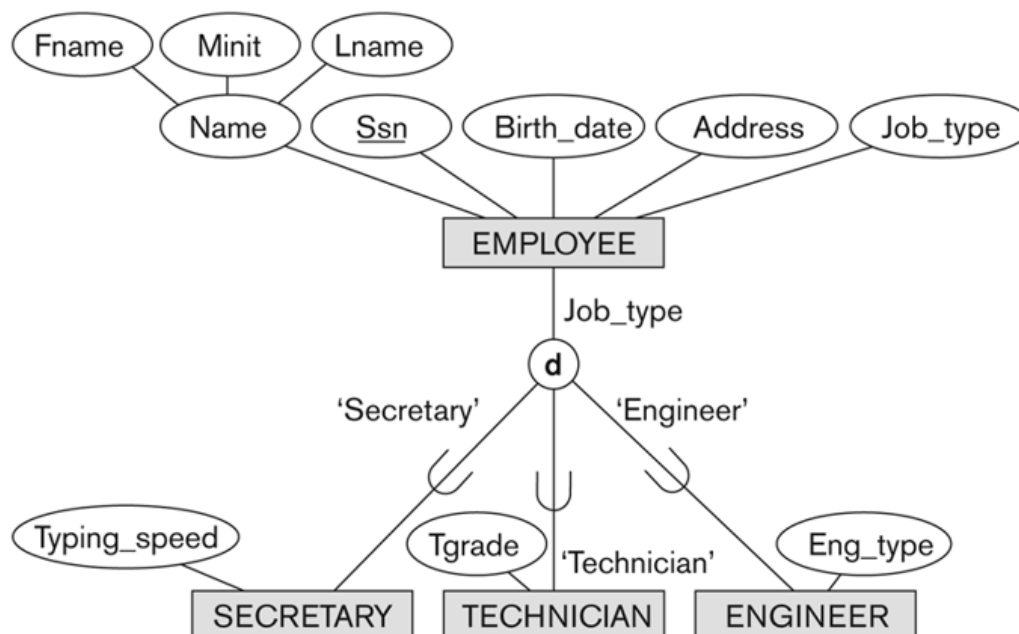
Specialization in EER (Extended Entity-Relationship) modeling involves creating lower-level entity types (subclasses) from a higher-level entity type (superclass) based on some distinguishing characteristics. This process is a top-down approach where a general entity is divided into more specific entities.

**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.



**Fig. 6.4 EER diagram with specialization on attribute Job\_type**

Enhanced Entity-Relationship (EER) Diagram for an Employee Database with specialization based on the attribute Job\_Type, we use attribute-based specialization to classify employees into different categories based on their roles. Here's how the design would look:

Super Class: EMPLOYEE

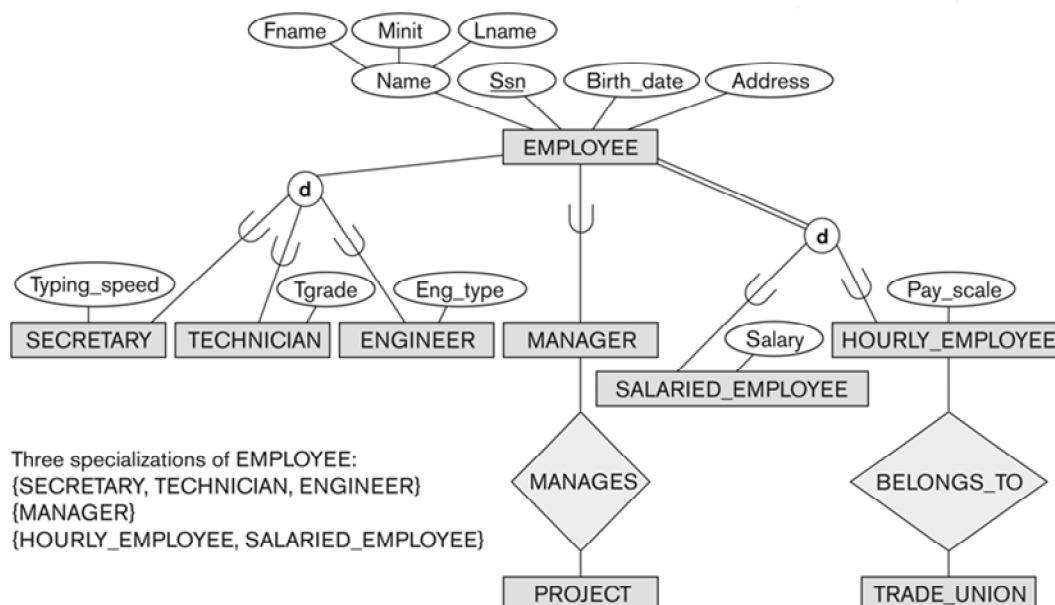
Sub Classes: SECRETARY, ENGINEER AND TECHNICIAN

### Specialization:

- Disjoint specialization based on Job\_Type:
  - If Job\_Type = " Secretary", the employee belongs to the SECRETARY subclass.
  - If Job\_Type = "Engineer", the employee belongs to the ENGINEER subclass.
  - If Job\_Type = " Technician", the employee belongs to the TECHNICIAN subclass.
- **Total Specialization:** Every Employee must belong to one subclass.

### Specialization Constraints

- **Disjoint Constraint:** An employee can belong to only one subclass (Manager, Engineer, or Intern).
- **Participation Constraint:** Specialization is **total** (every employee has a job type).



**Fig. 6.5 EER diagram with Sub class and specialization**

### 6.3.2 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

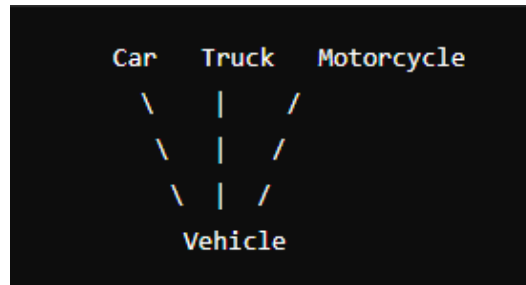
- **Higher-Level Entity (Superclass):** The generalized entity that encompasses the shared attributes and relationships of the lower-level entities.
- **Lower-Level Entities (Subclasses):** The specific entities that are combined to form the higher-level entity. Each subclass may have its own unique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

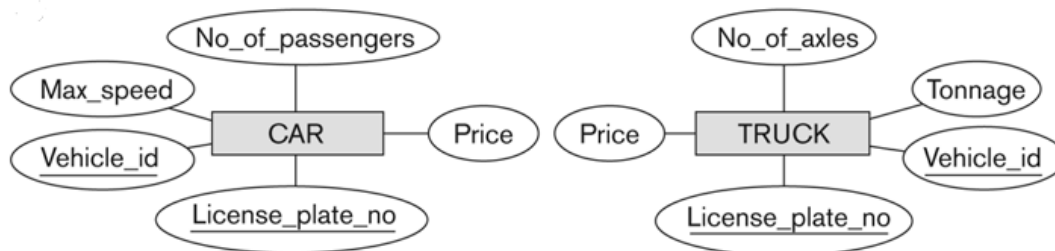
1. **Identify Common Attributes:**
  - Common attributes: vehicle\_id, license\_plate\_number, manufacturer, model
2. **Define the Higher-Level Entity (Superclass):**
  - Superclass: Vehicle
  - Attributes: vehicle\_id, license\_plate\_number, manufacturer, model
3. **Define Lower-Level Entities (Subclasses):**
  - 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).



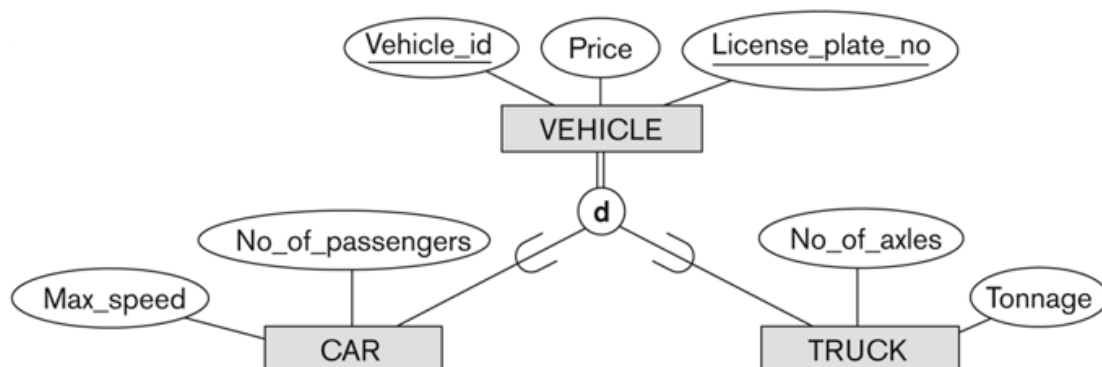
**Fig. 6.6 Generalization in EER Diagram**

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.



**Fig. 6.7 Entities CAR and TRUCK**

- CAR, TRUCK generalized into VEHICLE
- We can view {CAR, TRUCK} as a specialization of VEHICLE
- Alternatively, we can view VEHICLE as a generalization of CAR and TRUCK
- Arrow pointing to the generalized superclass represents a generalization shown in Fig 6.7.



**Fig. 6.8 Generalization of CAR and TRUCK into super class Truck**

Arrow pointing to the generalized superclass represents a generalization.

## 6.4. CONSTRAINTS AND CHARACTERISTICS OF SPECIALIZATION AND GENERALIZATION HIERARCHIES:

### 6.4.1 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.

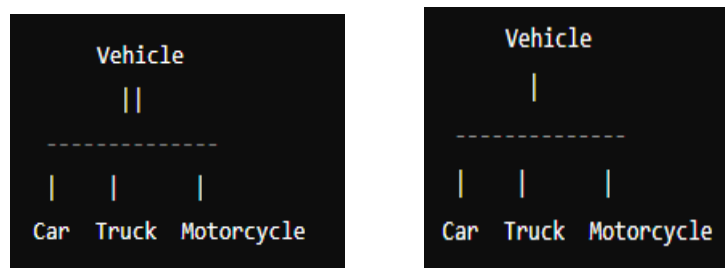


Fig. 6.9 (a) Total Specialization (b) Partial Specialization

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

### 6.4.2 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.

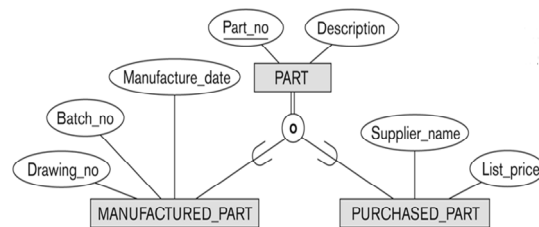
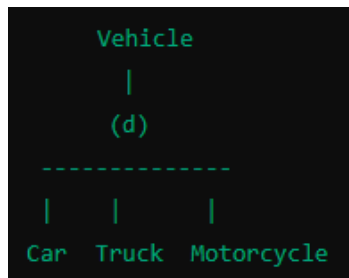


Fig. 6.10 (a) Disjoint (Exclusive) Constraint

(b) Overlapping Constraint

### 6.4.3 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:

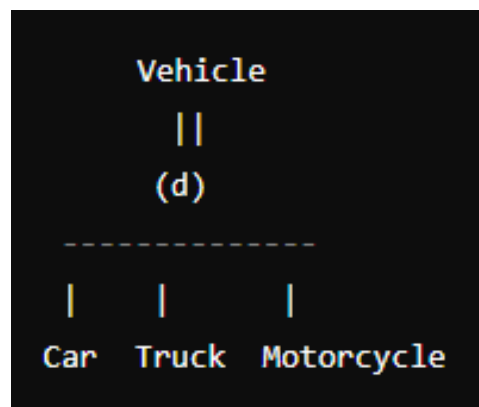
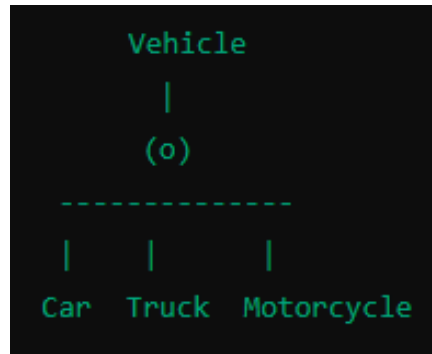


Fig. 6.11 Total Completeness and Disjoint Constraint

**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. 6.12 Partial Completeness and Overlapping Constraint**

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

**6.5. MODELING OF UNION TYPES USING CATEGORIES:**

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

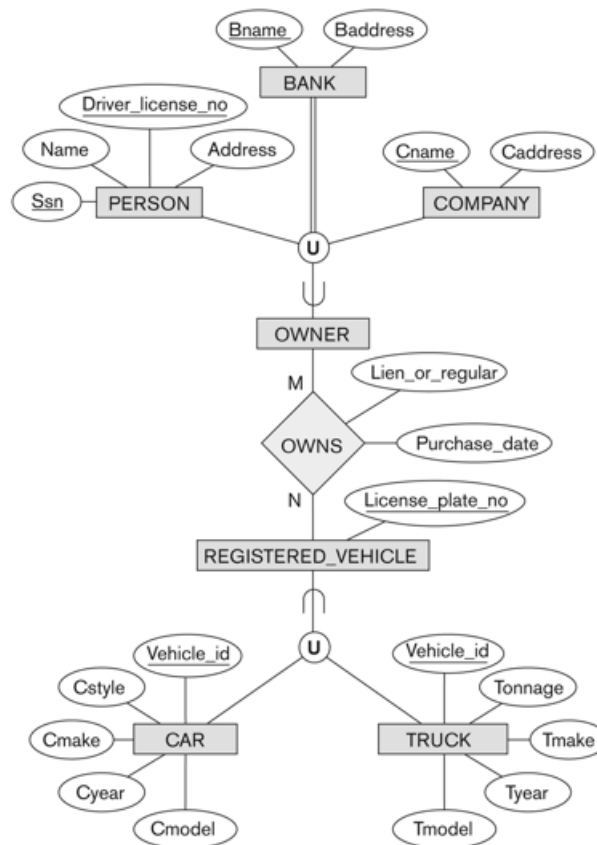
**Example:** A Member entity could be a union of Student, Teacher, and Alumni, allowing the Member to inherit attributes from any of these entities.

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.





**Fig. 6.13 Two Categories OWNER and REGISTERED\_VEHICLE**

### Union Entity:

Create a VehicleType 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.

## 6.6. 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.

### 6.6.1 Entities and Relationships

#### Entities:

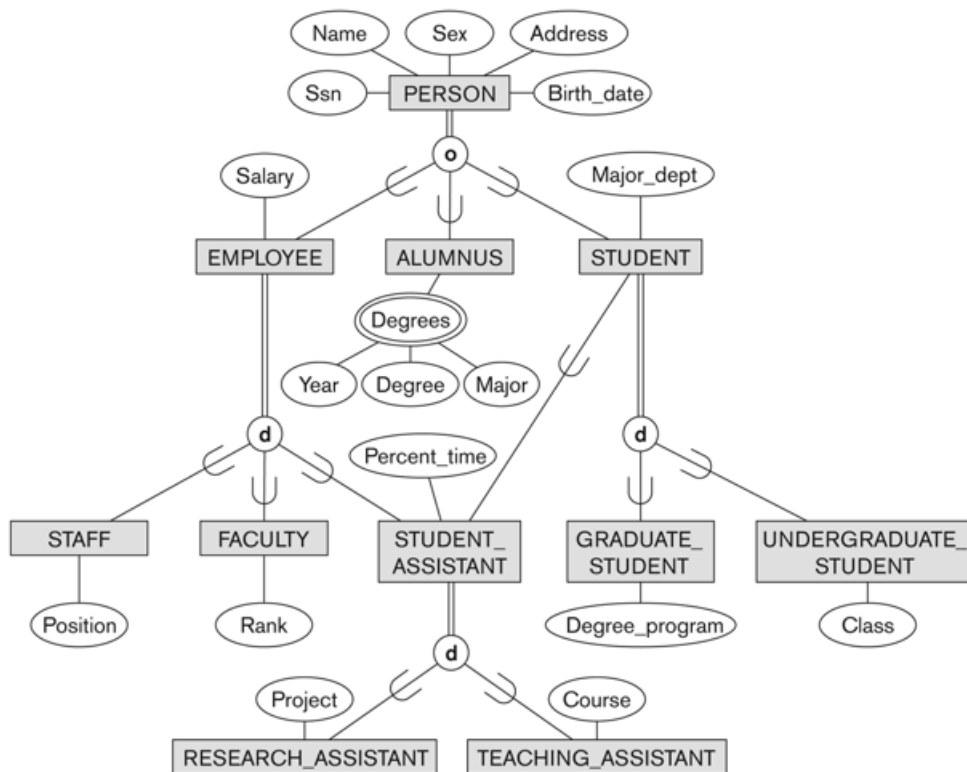
- Student (StudentID, Name, DateOfBirth, Major, Year, GPA)
- Professor (ProfessorID, Name, Department, Title, OfficeNumber)
- Course (CourseID, CourseName, Credits, Department)
- Department (DepartmentID, DepartmentName, Building)
- Classroom (RoomNumber, Building, Capacity)
- Enrollment (EnrollmentID,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

**6.6.2 Schema Diagram**

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



**Fig. 6.14 EER Diagram University Database**

This EER diagram covers key aspects of a university system, ensuring clear relationships between students, professors, courses, departments, and classrooms.

- In specialization, start with an entity type and then define subclasses of the entity type by successive specialization called a top-down conceptual refinement process.
- In generalization, start with many entity types and generalize those that have common properties Called a bottom-up conceptual synthesis process.
- In practice, a combination of both processes is usually employed.

## **6.7. DESIGN CHOICES AND FORMAL DEFINITIONS:**

### **6.7.1 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.

### **6.7.2 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.

## **6.8. 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.

## **6.9. TECHNICAL TERMS:**

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

## **6.10. SELF ASSESSMENT QUESTIONS:**

### **Essay questions:**

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

### **Short Notes:**

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

**6.11. 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 (6<sup>th</sup> ed.). Addison-Wesley.
- 7) Silberschatz, A., Korth, H.F., & Sudarshan, S. (2011). Database System Concepts (6<sup>th</sup> ed.). McGraw-Hill.

**Dr. U. Surya Kameswari**

## **LESSON-7**

# **THE RELATIONAL DATA MODEL AND RELATIONAL DATABASE CONSTRAINTS**

### **AIMS AND OBJECTIVES:**

The primary goal of this chapter is to understand the concept of Relational Data Model and Relational Database Constraints. The chapter began Relational Model Concepts, Relational Model Constraints and Relational Database Schemas, Update Operations, Transactions and Dealing with Constraint Violations. After completing this chapter, the student will understand The Relational Data Model and Relational Database Constraints. Also know about Update Operations, Transactions and Dealing with Constraint Violations at end of chapter.

### **STRUCTURE:**

- 7.1. Introduction**
- 7.2. Relational Data Model Concepts**
- 7.3. Relational Model Constraints and Relational Database Schemas**
- 7.4. Update Operations**
- 7.5. Transactions and Dealing with Constraint Violations.**
- 7.6. Summary**
- 7.7. Technical Terms**
- 7.8. Self-Assessment Questions**
- 7.9. Suggested Readings**

### **7.1. INTRODUCTION:**

The relational data model, introduced by E.F. Codd in 1970, revolutionized database management by providing a structured framework for organizing data. Unlike previous hierarchical and network models, the relational model relies on simple yet powerful concepts of tables (relations), which allow for flexibility, scalability, and ease of use. This chapter delves into the core concepts, constraints, and operations of the relational model, emphasizing the importance of maintaining data integrity through various constraints and how to handle violations during database transactions.

The chapter first covered began with Relational Model Concepts, Relational Model Constraints and Relational Database Schemas, Update Operations, Transactions and Dealing with Constraint Violations.

## 7.2. RELATIONAL MODEL CONCEPTS:

### 7.2.1 Data Structures in the Relational Model

The primary data structure in the relational model is the relation (table). Each relation consists of tuples (rows) and attributes (columns).

- ❖ **Relation:** A relation is a table with columns and rows. Each table represents an entity or a relationship among entities. In a relational database, data is organized into relations, which are called tables.
  - **Relation Schema:** Defines the structure of a relation, including its name and the name and type of each column.
  - **Relation Instance:** A set of tuples (rows) that conform to the relation schema at any given point in time.

#### Example:

Students (Ssn, Name, Home\_phone, Adress, Gpa)

- ❖ **Tuple:** A single row in a table.
- ❖ **Attribute:** A column in a table, representing a data field.
- ❖ **Domain:** The set of permissible values for an attribute.

#### Example:

- Domain of GPA might be real numbers between 0.0 and 4.0.
- ❖ **Degree:** The number of attributes in a relation.
- ❖ **Cardinality:** The number of tuples in a relation.

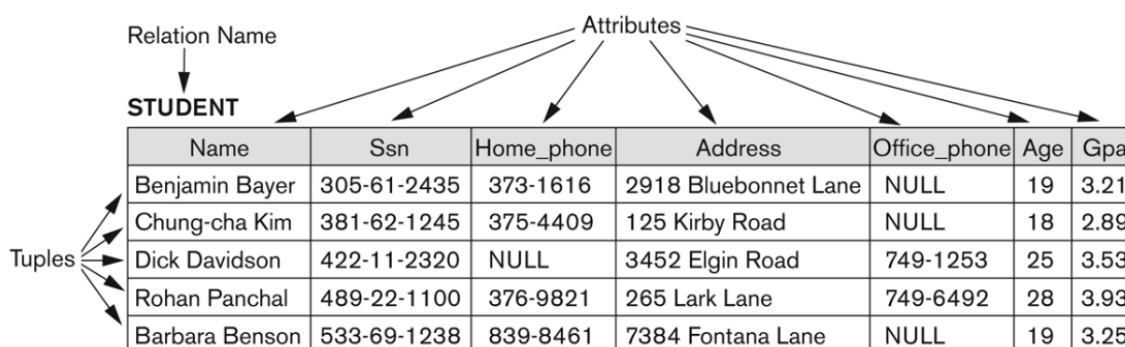


Fig. 7.1 Attribute and Tuple in STUDENT Relation

### Advantages of the Relational Model

- ❖ **Simplicity:** The tabular structure is intuitive and easy to understand.
- ❖ **Flexibility:** Easy to modify schema without affecting existing applications.

- ❖ **Data Integrity:** Ensured through the use of constraints and normalization.
- ❖ **Data Independence:** Logical and physical data independence is maintained.
- ❖ **SQL:** A powerful query language that allows for complex queries and operations.

### 7.3. RELATIONAL MODEL CONSTRAINTS AND RELATIONAL DATABASE SCHEMAS:

Relational model constraints and relational database schemas are essential for maintaining data integrity, consistency, and validity. Constraints such as domain, key, entity integrity, and referential integrity ensure the correctness of the data. A well-designed relational database schema provides a structured blueprint for organizing data and defining relationships between tables. Normalization helps in reducing redundancy and dependency, while denormalization can improve query performance. Understanding these concepts is crucial for designing efficient and reliable relational databases.

#### 7.3.1 Relational Model Constraints

Constraints are rules applied to data in a relational database to ensure accuracy and consistency. The primary types of constraints in the relational model include domain constraints, key constraints, entity integrity constraints, and referential integrity constraints.

##### ❖ Domain Constraints

Domain constraints define the permissible values for an attribute. They restrict the data type and the range of values that can be stored in an attribute.

```
CREATE TABLE Students (  
    StudentID INTEGER,  
    Name VARCHAR(50),  
    Major VARCHAR(50),  
    Year INTEGER CHECK (Year >= 1 AND Year <= 4),  
    GPA FLOAT CHECK (GPA >= 0.0 AND GPA <= 4.0)  
);
```

##### ❖ Key Constraints

Key constraints ensure that a set of attributes uniquely identifies each tuple in a relation. The primary key constraint enforces uniqueness and non-nullability of the primary key attribute(s).

##### ➤ Primary Key:

```
CREATE TABLE Students (  
    StudentID INTEGER PRIMARY KEY,  
    Name VARCHAR(50),  
    Major VARCHAR(50),
```

Year INTEGER,  
 GPA FLOAT  
 );

Example: Consider the CAR relation schema:

- CAR(State, Reg#, SerialNo, Make, Model, Year)
- Primary Key= SerialNo

➤ **Superkey:**

- Any key is a superkey (but not vice versa)
- Any set of attributes that includes a key is a superkey
- A minimal superkey is also a key

**Example:** CAR has two keys:

- Key1 = {State, Reg#}
- Key2 = {SerialNo}
- Both are also superkeys of CAR
- {SerialNo, Make} is a superkey but not a key.

If a relation has several candidate keys, one is chosen arbitrarily to be the primary key. The primary key attributes are underlined.

**Example:** Consider the CAR relation schema:

- CAR(State, Reg#, SerialNo, Make, Model, Year)
- We chose SerialNo as the primary key

**CAR**

<u>License_number</u>	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

**Fig. 7.2 Two candidate keys in CAR Relation: License\_number and Engine\_serial\_number**



### ❖ Entity Integrity Constraints

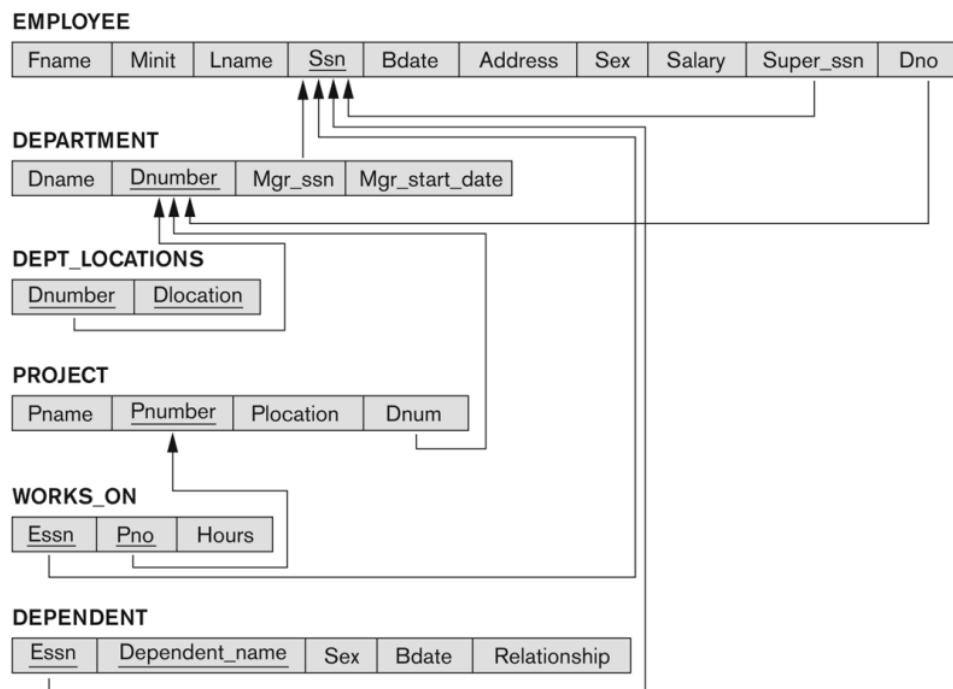
Entity integrity constraints ensure that the primary key of a table is unique and not null. This is crucial for uniquely identifying each record in a table.

```
CREATE TABLE Students (  
    StudentID INTEGER PRIMARY KEY,  
    Name VARCHAR(50) NOT NULL,  
    Major VARCHAR(50),  
    Year INTEGER,  
    GPA FLOAT  
);
```

### ❖ Referential Integrity Constraints

Referential integrity constraints ensure that foreign keys correctly and consistently reference primary keys in related tables. This maintains the logical consistency between tables.

```
CREATE TABLE Departments (  
    DepartmentID INTEGER PRIMARY KEY,  
    DepartmentName VARCHAR(50)  
);  
  
CREATE TABLE Students (  
    StudentID INTEGER PRIMARY KEY,  
    Name VARCHAR(50),  
    MajorID INTEGER,  
    Year INTEGER,  
    GPA FLOAT,  
    FOREIGN KEY (MajorID) REFERENCES Departments (DepartmentID)  
);
```



**Fig. 7.3 Referential Integrity Constraints on the COMPANY Relation**

### 7.3.2 Relational Database Schemas

A relational database schema defines the structure of a relational database, including the tables, attributes, data types, and constraints. It provides a blueprint for how data is organized and how relationships between tables are maintained.

Consider a university database schema with the following tables: Students, Courses, Enrollments, and Professors.

```
CREATE TABLE Students (
    StudentID INTEGER PRIMARY KEY,
    Name VARCHAR(50) NOT NULL,
    Major VARCHAR(50),
    Year INTEGER CHECK (Year >= 1 AND Year <= 4),
    GPA FLOAT CHECK (GPA >= 0.0 AND GPA <= 4.0)
);

CREATE TABLE Courses (
    CourseID INTEGER PRIMARY KEY,
    CourseName VARCHAR(100) NOT NULL,
    Credits INTEGER CHECK (Credits > 0)
);

CREATE TABLE Enrollments (
```

```

EnrollmentID INTEGER PRIMARY KEY,
StudentID INTEGER,
CourseID INTEGER,
Grade VARCHAR(2),
FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);

CREATE TABLE Professors (
  ProfessorID INTEGER PRIMARY KEY,
  Name VARCHAR(50) NOT NULL,
  Department VARCHAR(50),
  Title VARCHAR(50)
);

```

**EMPLOYEE**

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

**DEPARTMENT**

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

**DEPT\_LOCATIONS**

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

**PROJECT**

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

**WORKS\_ON**

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

**DEPENDENT**

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

**Fig. 7.4 Schema Diagram for the COMPANY Relation**

## 7.4 UPDATE OPERATIONS:

In a relational database, update operations modify data within the tables (relations) to reflect changes in the real-world system being modeled. There are three primary update operations:

### ➤ INSERT Operation

- Adds new tuples (rows) to a relation.
- Rules for Validity:

- The values for each attribute must match the domain of that attribute.
- The primary key must be unique for the new tuple to avoid duplicates.
- Referential integrity must be maintained; if a foreign key exists, it must reference a valid tuple in the parent table.

```
INSERT INTO Employee (Emp_ID, Name, Job_Type)
VALUES (101, 'Alice', 'Engineer');
```

#### ➤ **DELETE Operation**

- Removes existing tuples from a relation.
- **Rules for Validity:**
  - If the tuple to be deleted is referenced by a foreign key in another table, the deletion may be restricted or cascade (based on constraints).
  - Deleting a tuple must not violate database constraints.

```
DELETE FROM Employee
WHERE Emp_ID = 101;
```

#### ➤ **UPDATE Operation**

- Modifies attribute values for existing tuples.
- **Rules for Validity:**
  - New values must respect attribute domains.
  - Primary key changes must avoid duplication and maintain unique identification.
  - Referential integrity should be preserved when modifying foreign key values.

```
UPDATE Employee
SET Job_Type = 'Manager'
WHERE Emp_ID = 102;
```

#### ➤ **SELECT Operation**

While not strictly an "update operation," the **SELECT** statement is often combined with update operations to identify specific tuples to modify.

```
UPDATE Employee
SET Salary = Salary * 1.1
WHERE Job_Type = 'Engineer';
```

## 7.5. TRANSACTIONS AND DEALING WITH CONSTRAINT VIOLATIONS:

In a relational database, each operation-**INSERT**, **DELETE**, or **UPDATE**-can lead to various types of constraint violations. Here's a breakdown of the possible violations for each operation:

### ❖ **INSERT Operation**

The **INSERT** operation adds new data to a table. Violations can occur if the new data does not meet the database constraints.

- **NOT NULL Violation:**
  - Inserting a row without providing values for columns defined as NOT NULL.
- **UNIQUE Violation:**
  - Inserting a duplicate value into a column or set of columns with a UNIQUE constraint.
- **PRIMARY KEY Violation:**
  - Attempting to insert a row with a duplicate or NULL value in the PRIMARY KEY column(s).
- **FOREIGN KEY Violation:**
  - Inserting a value in a child table that does not exist in the referenced parent table.
- **CHECK Constraint Violation:**
  - Inserting a value that does not satisfy the condition defined in a CHECK constraint (e.g., age < 18).
- **Domain Constraint Violation:**
  - Inserting a value that falls outside the predefined domain (data type, format, or range).

#### **Example:**

```
FOREIGN KEY violation: Dept_ID=5 does not exist in Department table
INSERT INTO Employee (Emp_ID, Name, Dept_ID) VALUES (101, 'Alice', 5);
```

### ❖ **DELETE Operation**

The **DELETE** operation removes rows from a table. Violations arise primarily due to the presence of foreign key dependencies.

- **FOREIGN KEY Violation:**
  - Deleting a parent row that is referenced by a child table without enabling ON DELETE CASCADE or handling dependencies.

#### **Example:**

```
-- FOREIGN KEY violation: Cannot delete a department that has employees
DELETE FROM Department WHERE Dept_ID = 1;
```

### ❖ UPDATE Operation

The **UPDATE** operation modifies existing rows. Violations can occur if the updated values conflict with constraints.

- **NOT NULL Violation:**
  - Updating a NOT NULL column to NULL.
- **UNIQUE Violation:**
  - Updating a column to a value that already exists in another row when the column has a UNIQUE constraint.
- **PRIMARY KEY Violation:**
  - Modifying the PRIMARY KEY column(s) to create a duplicate or NULL value.
- **FOREIGN KEY Violation:**
  - Changing a foreign key value to one that does not exist in the referenced table.
- **CHECK Constraint Violation:**
  - Updating a column to a value that fails a CHECK constraint.

#### Example:

```
-- UNIQUE violation: Both employees cannot have the same Emp_ID
UPDATE Employee SET Emp_ID = 102 WHERE Name = 'Alice';
```

### ❖ 4. SELECT Operation

Though **SELECT** is generally non-intrusive, errors can arise in scenarios like:

- Violating **security constraints** by attempting to access restricted columns or rows.
- Querying with invalid data types or formats.

#### Preventive Strategies

1. **Validations at Application Layer:**
  - Check inputs before execution of operations.
2. **Database Triggers:**
  - Automate checks and corrective actions at the database level.
3. **Transaction Management:**
  - Use transactions (BEGIN, COMMIT, ROLLBACK) to ensure atomicity.
4. **Database Design:**
  - Ensure a well-normalized schema with appropriately defined constraints.

By designing for and testing against these violations, you can maintain database integrity and minimize runtime errors.

## 7.6. SUMMARY:

The relational data model, with its foundation in relations, keys, and constraints, provides a robust framework for managing structured data. Understanding relational algebra and calculus is crucial for effective querying. Constraints play a vital role in maintaining data integrity, and handling update operations with respect to these constraints ensures consistency and reliability in a relational database. Transactions and ACID properties are fundamental to preserving the integrity and consistency of databases during concurrent and complex operations.

This chapter provides a comprehensive overview of the relational data model, its constraints, and how to manage database integrity and transactions, offering essential knowledge for database design and management.

## 7.7. TECHNICAL TERMS:

Relational Model, Shema, Relationship, Relational Algebra, Relational Calculus, insert, select, projection, Union, Find, Natural Join.

## 7.8. SELF ASSESSMENT QUESTIONS:

### Essay questions:

- 1) Illustrate about Relational Calculus?
- 2) Describe about Relational Algebra?
- 3) Explain about Relational Model concepts?

### Short Notes:

- 1) Write about keys?
- 2) Define Attributes?
- 3) Explain about Relational Shema?

## 7.9. 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". 8<sup>th</sup> Edition. Addison-Wesley.
- 3) Silberschatz, A., Korth, H.F., & Sudarshan, S. (2010). "Database System Concepts". 6<sup>th</sup> Edition. McGraw-Hill.
- 4) Ullman, J.D., & Widom, J. (2008). "A First Course in Database Systems". 3<sup>rd</sup> Edition. Pearson.

## LESSON-8

### 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
- 8.2. Unary Relational Operations
- 8.3. Relational Algebra Operations from Set Theory
- 8.4. Binary Relational Operations
- 8.5. Additional Relational Operations
- 8.6. Examples of Queries in Relational Algebra
- 8.7. Summary
- 8.8. Technical Terms
- 8.9. Self-Assessment Questions
- 8.10. Suggested Readings

#### 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 ( $\sigma$ ):** 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:**

- $\sigma$  SALARY>30,000 (EMPLOYEE) retrieves employees earning more than 30,000.
- $\sigma$ DNO = 4 (EMPLOYEE)
- Can handle complex conditions using logical operators (AND, OR, NOT).
- In general, the select operation is denoted by  $\sigma$ < selection condition > (R) where the symbol  $\sigma$  (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:**

- $\sigma$  (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 ( $\pi$ ):** 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:

- $\pi$  FNAME, LNAME, SALARY(EMPLOYEE) retrieves names and salaries of employees.
- The general form of the project operation is  $\pi$ <attribute list >(R) where  $\pi$  (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:**  $\pi$  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 ( $\rho$ ):** 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:  $\rho$ (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  $R \cup S$ , 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.

$DEP5\_EMPS \leftarrow \sigma_{DNO=5} (EMPLOYEE)$

$RESULT1 \leftarrow \pi_{SSN}(DEP5\_EMPS)$

$RESULT2(SSN) \leftarrow \pi_{SUPERSSN}(DEP5\_EMPS)$

$RESULT \leftarrow RESULT1 \cup RESULT2$

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. 8.3 Union Operation among STUDENT and INSTRUCTOR Relations**

#### ❖ INTERSECTION Operation

The result of this operation, denoted by  $R \cap S$ , 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. 8.4 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. 8.5 Set Difference Operation among STUDENT and INSTRUCTOR Relations**

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

$$R \cup S = S \cup R, \text{ and } R \cap S = S \cap R$$

- Both union and intersection can be treated as n-ary operations applicable to any number of relations as both are associative operations; that is

$$R \cup (S \cap T) = (R \cup S) \cap T, \text{ and } (R \cap S) \cap T = R \cap (S \cap T)$$

- 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  $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$  is a relation  $Q$  with degree  $n + m$  attributes  $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ , 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  $|R \times S|$  will have  $nR * nS$  tuples.
- The two operands do NOT have to be "type compatible".

**Example:**

FEMALE\_EMPS  $\leftarrow \sigma_{SEX='F'}(EMPLOYEE)$

EMPNAMES  $\leftarrow \pi_{FNAME, LNAME, SSN}(FEMALE\_EMPS)$

EMP\_DEPENDENTS  $\leftarrow EMPNAMES \times DEPENDENT$

FEMALE_EMPS	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	Alicia	J	Zelays	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	888662255	4
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES	FNAME	LNAME	SSN
	Alicia	Zelays	999887777
	Jennifer	Wallace	987654321
	Joyce	English	453453453

EMP_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	...
	Alicia	Zelays	999887777	333445555	Alice	F	1986-04-05	...
	Alicia	Zelays	999887777	333445555	Theodore	M	1983-10-25	...
	Alicia	Zelays	999887777	333445555	Joy	F	1958-05-03	...
	Alicia	Zelays	999887777	987654321	Abner	M	1942-02-28	...
	Alicia	Zelays	999887777	123456789	Michael	M	1988-01-04	...
	Alicia	Zelays	999887777	123456789	Alice	F	1988-12-30	...
	Alicia	Zelays	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. 8.6 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  $R(A_1, A_2, \dots, A_n)$  and  $S(B_1, B_2, \dots, B_m)$  is:

$$R \bowtie \langle \text{join condition} \rangle S$$

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.

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

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. 8.7 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. 8.8 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.

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

$$\text{TEMP} \leftarrow (\text{EMPLOY} \text{ EE } \bowtie_{\text{SSN}=\text{MGRSSN}} \text{DEPARTMENT})$$

$$\text{RESULT} \leftarrow \pi_{\text{FNAME}, \text{MINIT}, \text{DNAME}}(\text{TEMP})$$

- **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.  
 $\text{JSMITH\_SSN}(\text{ESSN}) \leftarrow \pi_{\text{SSN}}(\sigma_{\text{NAME}='John' \text{ AND } \text{LNAME}='Smith'}(\text{EMPLOYEE}))$   
 $\text{JSMITH\_PROJ} \leftarrow \pi_{\text{P\_NO}}(\text{JSMITH\_SSN} * \text{WORKS\_ON})$

WORKS\_ON2  $\leftarrow \pi_{\text{ESSN,P\_NO}}(\text{WORKS\_ON})$

DIV\_HERE(SSN)  $\leftarrow \text{WORKS\_ON2} \div \text{JSMITH PROJ}$

RESULT  $\leftarrow \pi_{\text{NAME, LNAME}}(\text{EMPLOYEE} * \text{DIV\_HERE})$

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. 8.9 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. 8.10 Result of AVG Aggregate Operation**

### ❖ OUTER JOIN

- **LEFT OUTER JOIN:**  $R_3(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m) \leftarrow R_1(A_1, A_2, \dots, A_n) \bowtie_{\langle \text{JOIN COND} \rangle} R_2(B_1, B_2, \dots, B_m)$

✓ This operation keeps every tuple  $t$  in left relation  $R_1$  in  $R_3$ , and fills “NULL” for attributes  $B_1, B_2, \dots, B_m$  if the join condition is not satisfied for  $t$ .

✓ **Example,**

$TEMP \leftarrow (EMPLOYEE \bowtie_{SSN=MGRSSN} DEPARTMENT)$   
 $RESULT \leftarrow \pi_{FNAME, MINIT, DNAME}(TEMP)$

- **RIGHT OUTER JOIN:** similar to LEFT OUTER JOIN, but keeps every tuple  $t$  in right relation  $R_2$  in the resulting relation  $R_3$ .

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

- **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.



- $R_3(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m, C_1, C_2, \dots, C_p) \leftarrow R_1(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m) \text{ OUTER UNION } R_2(A_1, A_2, \dots, A_n, C_1, C_2, \dots, C_p)$
- The list of compatible attributes are represented only once in  $R_3$ .
- Tuples from  $R_1$  and  $R_2$  with the same values on the set of compatible attributes are represented only once in  $R_3$
- In  $R_3$ , 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  $R_3(\text{NAME, SSN, DEPT, ADVISOR, RANK})$

#### 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.

$$\begin{aligned} RESEARCH\_DEPT &\leftarrow \sigma_{DNAME='Research'}(DEPARTMENT) \\ RESEARCH\_EMPS &\leftarrow (RESEARCH\_DEPT \bowtie_{DNUMBER=DNO} (EMPLOYEE)) \\ RESULT &\leftarrow \pi_{FNAME,LNAME,ADDRESS}(RESEARCH\_EMPS) \end{aligned}$$

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

$$\begin{aligned} STAFFORD\_PROJS &\leftarrow \sigma_{PLOCATION='Stafford'}(PROJECT) \\ CONTR\_DEPT &\leftarrow (STAFFORD\_PROJS \bowtie_{DNUM=DNUMBER} (DEPARTMENT)) \\ PROJ\_DEPT\_MGR &\leftarrow (CONTR\_DEPT \bowtie_{MGRSSN=SSN} (EMPLOYEE)) \\ RESULT &\leftarrow \pi_{PNUMBER,DNUM,LNAME,ADDRESS,BDATE}(PROJ\_DEPT\_MGR) \end{aligned}$$

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

$$\begin{aligned} DEPT5\_PROJS(PNO) &\leftarrow \pi_{PNUMBER}(\sigma_{DNUM=5}(PROJECT)) \\ EMP\_PROJ(SSN,PNO) &\leftarrow \pi_{ESSN,PNO}(WORKS\_ON) \\ RESULT\_EMP\_SSNS &\leftarrow EMP\_PROJ \div DEPT\_PROJS \\ RESULT &\leftarrow \pi_{LNAME,FNAME}(RESULT\_EMP\_SSNS * EMPLOYEE) \end{aligned}$$

- 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.

$$\begin{aligned} SMITHS(ESSN) &\leftarrow \pi_{SSN}(\sigma_{LNAME='Smith'}(EMPLOYEE)) \\ SMITH\_WORKER\_PROJ &\leftarrow \pi_{PNO}(WORKS\_ON * SMITHS) \\ MGRS &\leftarrow \pi_{LNAME,DNUMBER}(EMPLOYEE \bowtie_{SSN=MGRSSN} DEPARTMENT) \\ SMITH\_MANAGED\_DEPTS(DNUM) &\leftarrow \pi_{DNUMBER}(\sigma_{LNAME='Smith'}(MGRS)) \\ SMITH\_MGR\_PROJS(PNO) &\leftarrow \pi_{PNUMBER}(SMITH\_MANAGED\_DEPTS * PROJECT) \\ RESULT &\leftarrow (SMITH\_WORKER\_PROJS \cup SMITH\_MGR\_PROJS) \end{aligned}$$

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

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

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

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

- Retrieve the names of employees who have no dependents.

$$ALL\_EMPS \leftarrow \pi_{SSN}(EMPLOYEE)$$

$$EMPS\_WITH\_DEPS(SSN) \leftarrow \pi_{ESSN}(DEPENDENT)$$

$$EMPS\_WITHOUT\_DEPS \leftarrow (ALL\_EMPS - EMPS\_WITH\_DEPS)$$

$$RESULT \leftarrow \pi_{LNAME, FNAME}(EMPS\_WITHOUT\_DEPS * EMPLOYEE)$$

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

$$MGRS(SSN) \leftarrow \pi_{MGRSSN}(DEPARTMENT)$$

$$EMPS\_WITH\_DEPS(SSN) \leftarrow \pi_{ESSN}(DEPENDENT)$$

$$MGRS\_WITH\_DEPS \leftarrow (MGRS \cap EMPS\_WITH\_DEPS)$$

$$RESULT \leftarrow \pi_{LNAME, FNAME}(MGRS\_WITH\_DEPS * EMPLOYEE)$$

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

$$EMPS\_DEPS \leftarrow$$

$$(EMPLOYEE \bowtie_{SSN=ESSN\ AND\ SEX=SEX\ AND\ FNAME=DEPENDENT\_NAME} DEPENDENT)$$

$$RESULT \leftarrow \pi_{LNAME, FNAME}(EMPS\_DEPS)$$

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

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

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

$$RESULT \leftarrow$$

$$DNO \wp_{SUM\ SALARY, MAXIMUM\ SALARY, MINIMUM\ SALARY, AVERAGE\ SALARY}(EMPLOYEE)$$

**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." 8<sup>th</sup> 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". 3<sup>rd</sup> Edition. Pearson.

**Dr. U. Surya Kameswari**

## LESSON-9

### 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**
- 9.2. The Tuple Relational Calculus (TRC)**
- 9.3. The Domain Relational Calculus (DRC)**
- 9.4. Summary**
- 9.5. Technical Terms**
- 9.6. Self-Assessment Questions**
- 9.7. Suggested Readings**

**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:

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

Where  $t_1, t_2, \dots, t_n, t_{n+1}, t_{n+2}, \dots, t_{n+m}$  are tuple variables, each  $A_i$  is an attribute of the relation on which  $t_i$  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  $(F_1 \text{ and } F_2)$ ,  $(F_1 \text{ or } F_2)$ ,  $\text{not}(F_1)$ ,  $\text{not}(F_2)$ .

And

- \*  $(F_1 \text{ and } F_2)$  is TRUE if both  $F_1$  and  $F_2$  are TRUE; otherwise, it is FALSE.
- \*  $(F_1 \text{ or } F_2)$  is FALSE if both  $F_1$  and  $F_2$  are FALSE; otherwise, it is TRUE.
- \*  $\text{not}(F_1)$  is TRUE if  $F_1$  is FALSE; it is FALSE if  $F_1$  is TRUE.
- \*  $\text{not}(F_2)$  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** ( $\exists$ ) 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** ( $\forall$ ) 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 ( $\exists$ ) and universal ( $\forall$ ) 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 ( $\exists$ ):** 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:

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

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

**Syntax:**

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

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

**Example in Tuple Relational Calculus (TRC):**

Find employees who are in all departments:

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

➤ **Comparison Between Existential and Universal Quantifiers**

Aspect	Existential Quantifier ( $\exists$ )	Universal Quantifier ( $\forall$ )
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 $\exists$ (e.g., $\exists x P(x)$ )	Represented by $\forall$ (e.g., $\forall x P(x)$ )
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	$\exists t (t \in Employee \wedge t.salary > 50000)$ : Checks if there exists an employee earning more than \$50,000.	$\forall t (t \in Employee \rightarrow t.age > 18)$ : 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 $\exists x P(x)$ is $\forall x \neg P(x)$ .	The negation of $\forall x P(x)$ is $\exists x \neg P(x)$ .
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.

$$\{t.FNAME, t.LNAME, t.ADDRESS \mid EMPLOYEE(t) \text{ and } (\exists d) (DEPARTMENT(d) \text{ and } d.DNAME = 'Research' \text{ and } d.DNUMBER = t.DNO)\}$$

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

$$\{p.PNUMBER, p.DNUM, m.LNAME, m.BDATE, m.ADDRESS \mid PROJECT(p) \text{ and } EMPLOYEE(m) \text{ and } p.PLOCATION = 'Stafford' \text{ and } ((\exists d)(DEPARTMENT(d) \text{ and } p.DNUM = d.DNUMBER \text{ and } d.MGRSSN = m.SSN))\}$$

- 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 :

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

- A formula is made up of atoms.
  - An atom of the form  $R(x_1, x_2, \dots, x_j)$  (or simply  $R(x_1x_2 \dots x_j)$ ), 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  $\langle x_1, x_2, \dots, x_j \rangle$  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  $x_1, x_2, \dots, x_j$  are assigned values corresponding to a tuple of  $R$ , then the atom is TRUE.
  - An atom of the form  $x_i \text{ op } x_j$ , 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  $x_i \text{ op } c$  or  $c \text{ op } x_j$ , 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 ( $\wedge$ ), disjunction ( $\vee$ ), negation ( $\neg$ ), and quantifiers like existential ( $\exists$ ) and universal ( $\forall$ ) 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.

$$\{ e.name \mid \exists e\_id \exists d\_id (Employee(e\_id, e.name, e.salary) \wedge Works(e\_id, d\_id) \wedge d\_id = 'D2') \}$$
**Explanation:**

- e.name: The result we want to retrieve, which is the employee's name.
- $\exists e\_id \exists d\_id$ : 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.

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

IS EQUIVALENT TO:

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

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.,  $\wedge$ ,  $\vee$ ,  $\neg$ ) and quantifiers ( $\exists$ ,  $\forall$ ), 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 ( $\exists$ ) and universal ( $\forall$ ) quantifiers, along with logical operators like conjunction ( $\wedge$ ), disjunction ( $\vee$ ), and negation ( $\neg$ ), 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". 8<sup>th</sup> Edition. Addison-Wesley.
- 3) Silberschatz, A., Korth, H.F., & Sudarshan, S. (2010). "Database System Concepts". 6<sup>th</sup> Edition. McGraw-Hill.
- 4) Ullman, J.D., & Widom, J. (2008). "A First Course in Database Systems". 3<sup>rd</sup> Edition. Pearson.

**Dr. Neelima Guntupalli**

## **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.

#### **STRUCTURE:**

- 10.1 Introduction**
- 10.2 SQL**
- 10.3 SQL Data Definitions and Data Types**
- 10.4 Specifying Constraints in SQL**
- 10.5 Schema Change Statements in SQL**
- 10.6 Basic Queries in SQL**
- 10.7 More Complex SQL Queries**
- 10.8 Insert, Delete, And Update Statements in SQL**
- 10.9 Triggers in SQL**
- 10.10 Views in SQL**
- 10.11 Summary**
- 10.12 Technical Terms**
- 10.13 Self-Assessment Questions**
- 10.14 Suggested Readings**

#### **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 CREATE, 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:

1. **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
2. **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 Materialized 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". 8<sup>th</sup> Edition. Addison-Wesley.
- 3) Silberschatz, A., Korth, H.F., & Sudarshan, S. (2010). "Database System Concepts". 6<sup>th</sup> Edition. McGraw-Hill.
- 4) Ullman, J.D., & Widom, J. (2008). "A First Course in Database Systems". 3<sup>rd</sup> Edition. Pearson.

**Dr. Neelima Guntupalli**

## **LESSON-11**

### **FUNCTIONAL DEPENDENCIES AND NORMALIZATION FOR RELATIONAL DATABASES**

#### **AIMS AND OBJECTIVES:**

The primary goal of this chapter is to understand the concept of Functional Dependencies and Normalization for Relational Databases. The chapter began 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. After completing this chapter, the student will understand Functional Dependencies and Normalization for Relational Databases.

#### **STRUCTURE:**

- 11.1. Introduction**
- 11.2. Informal Design Guidelines for Relation Schemas**
- 11.3. Functional Dependencies**
- 11.4. Normal Forms Based on Primary Keys**
- 11.5. Boyce-Codd Normal Form (BCNF)**
- 11.6. Summary**
- 11.7. Technical Terms**
- 11.8. Self-Assessment Questions**
- 11.9. Suggested Readings**

#### **11.1. 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.

The chapter first covered began with 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

## 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.

**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.

**This can lead to update anomalies:**

- **Insertion Anomaly:** Difficulty in adding new data due to missing information.
- **Update Anomaly:** Inconsistencies arising from updating data in multiple places.
- **Deletion Anomaly:** Unintended loss of data when a record is deleted.

**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.

**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.

By adhering to these informal design guidelines, database designers can create schemas that are intuitive, maintainable, and free from common data anomalies, ultimately leading to more reliable and efficient databases.

## 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  $X \rightarrow Y$ ) 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

- **Trivial FD:** A functional dependency  $X \rightarrow Y$  is trivial if Y is a subset of X.
- **Non-trivial FD:** An FD is non-trivial if Y is not a subset of X.

- ❖ **Trivial Functional Dependencies** A functional dependency  $X \rightarrow Y$  is considered trivial if the set of attributes  $Y$  is a subset of the set of attributes  $X$ . In other words, a trivial functional dependency is one where the dependent attributes are already included in the determinant. Trivial functional dependencies do not provide any new information about the data because they are inherently satisfied by the definition of functional dependencies.

**Example:** For a relation  $R(A,B,C)$ , the functional dependency  $\{A,B\} \rightarrow A$  is trivial because  $A$  is a subset of  $\{A,B\}$ .

- ❖ **Non-trivial Functional Dependencies** A functional dependency  $X \rightarrow Y$  is non-trivial if the set of attributes  $Y$  is not a subset of the set of attributes  $X$ . Non-trivial functional dependencies are significant because they indicate meaningful relationships between different attributes in the database and are essential for understanding the structure and constraints of the data.

**Example:** For a relation  $R(A,B,C)$ , the functional dependency  $A \rightarrow B$  is non-trivial because  $B$  is not a subset of  $A$ .

Non-trivial functional dependencies are crucial for the normalization process as they help identify and eliminate redundancies and update anomalies in the database schema.

### 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.

### 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  $X \rightarrow Y$ .
- **Augmentation:** If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$  for any  $Z$ .
- **Transitivity:** If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$ .

### Finding the Closure

The closure of a set of FDs, denoted as  $F^+$ , is found by repeatedly applying Armstrong's Axioms to the given set  $F$  until no new FDs can be derived.

### Example

Consider a relation  $R(A,B,C)$  with the following FDs:

1.  $A \rightarrow B$
2.  $B \rightarrow C$



To find the closure  $F^+$  of the set  $F = \{A \rightarrow B, B \rightarrow C\}$ :

1. Start with the given FDs:
  - $A \rightarrow B$  and  $B \rightarrow C$
  - $B \rightarrow C$  and  $C \rightarrow C$
2. Apply Transitivity to derive a new FD:
  - Since  $A \rightarrow B$  and  $B \rightarrow C$ , by Transitivity,  $A \rightarrow C$ .
3. The closure  $F^+$  includes:
  - $A \rightarrow B$  and  $B \rightarrow C$
  - $B \rightarrow C$  and  $C \rightarrow C$
  - $A \rightarrow C$  and  $C \rightarrow C$

The closure  $F^+$  represents all the functional dependencies that can be inferred from the initial set  $F$ . 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  $R(A,B,C,D)$  with the following functional dependencies:

1.  $A \rightarrow B$
2.  $C \rightarrow D$

To decompose this relation, we can follow these steps:

1. **Identify Functional Dependencies:** The given FDs are  $A \rightarrow B$  and  $C \rightarrow D$ .
2. **Check Normal Form:** Let's assume the relation  $R$  is not in BCNF because each FD does not have a superkey on the left-hand side.
3. **Decompose the Relation:**
  - Based on  $A \rightarrow B$ :
    - Create  $R_1(A,B)$ .
  - Based on  $C \rightarrow D$ :
    - Create  $R_2(C,D)$ .
  - Ensure that remaining attributes are appropriately placed to preserve all functional dependencies:
    - Combine the remaining attributes into another relation if needed:  $R_3(A,C)$ .

Thus, the original relation  $R(A,B,C,D)$  is decomposed into:

- $R_1(A,B)$
- $R_2(C,D)$
- $R_3(A,C)$

### 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  $R$  into  $R_1$  and  $R_2$ :

- The decomposition is lossless if  $R_1 \cap R_2$  forms a superkey for either  $R_1$  or  $R_2$ .

### 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:

- $R_1(A,B)$
- $R_2(C,D)$
- $R_3(A,C)$
- Lossless Join:  $R_1 \cap R_3 = \{A\}$  and  $R_2 \cap R_3 = \{C\}$ . Since  $A$  and  $C$  can act as keys in their respective decomposed relations, the join is lossless.
- Dependency Preservation: The original dependencies  $A \rightarrow B$  and  $C \rightarrow D$  are maintained in  $R_1$  and  $R_2$ .

## 11.4. 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).

### 11.4.1 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

**Fig. 11.1** the Courses column contains multiple values, violating 1NF.

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

**Fig. 11.2** To Convert this table to 1NF, We split the multi-valued column into separate rows:

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.

#### 11.4.2 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:**

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

**Fig. 11.3 Consider a table Enrollments that is in 1NF but not in 2NF:**

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

**Fig. 11.4 1NF to 2NF: remove partial dependencies****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 StudentName is fully dependent on StudentID.
- The Courses table ensures that CourseName and Instructor are fully dependent on CourseID.
- 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.

**11.4.3 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

**Enrollments 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

**Fig. 10.5 2NF to 3NF**

**In this normalized structure:**

- The Instructors table stores the instructor information, ensuring InstructorName is fully dependent on InstructorID.
- 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

### 11.5. BOYCE-CODD NORMAL FORM (BCNF):

**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  $X \rightarrow Y$ ,  $X$  must be a superkey (a set of attributes that uniquely identify a tuple in a relation).

#### 11.5.1 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.  $CourseID \rightarrow InstructorCourseID \setminus to InstructorCourseID \rightarrow Instructor$
2.  $Instructor \rightarrow RoomInstructor \setminus to RoomInstructor \rightarrow Room$

The composite key here can be either  $CourseIDCourseIDCourseID$  or  $InstructorInstructorInstructor$ , as each uniquely identifies a course offering.

**Identifying BCNF Violation**

In this case,  $Instructor \rightarrow RoomInstructor \setminus to RoomInstructor \rightarrow Room$  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.

**11.5.2 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

**Fig. 11.6 Examples and Decomposition into BCNF**

**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

**Fig. 11.7 Result of BCNF**

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.

#### **11.6. SUMMARY:**

Functional dependencies and normalization are fundamental principles in relational database design that ensure data integrity, reduce redundancy, and prevent update anomalies. By systematically applying normalization processes, such as achieving 1NF, 2NF, 3NF, and BCNF, database designers can create robust, efficient, and scalable databases. These normalized schemas not only improve data consistency but also enhance query performance and maintainability. Understanding and implementing these concepts are essential for developing reliable and effective database systems that can adapt to changing data requirements and business rules.

#### **11.7. TECHNICAL TERMS:**

Functional dependencies, normalization, 1NF, 2NF, 3NF, BCNF, Reliability, Decomposition.

#### **11.8. 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

#### **11.9. 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". 8<sup>th</sup> Edition. Addison-Wesley.
- 3) Silberschatz, A., Korth, H.F., & Sudarshan, S. (2010). "Database System Concepts". 6<sup>th</sup> Edition. McGraw-Hill.
- 4) Ullman, J.D., & Widom, J. (2008). "A First Course in Database Systems". 3<sup>rd</sup> Edition. Pearson.

## **LESSON-12**

### **RELATIONAL DATABASE DESIGN ALGORITHMS AND FURTHER DEPENDENCIES**

#### **AIMS AND OBJECTIVES:**

The primary goal of this chapter is to understand the concept of Relational Database Design Algorithms and Further Dependencies. The chapter began 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. After completing this chapter, the student will understand Relational Database Design Algorithms and Further Dependencies

#### **STRUCTURE:**

- 12.1. Introduction**
- 12.2. Properties of Relational Decompositions**
- 12.3. Algorithms for Relational Database Schema Design**
- 12.4. Multivalued Dependencies and Fourth Normal Form**
- 12.5. Join Dependencies and Fifth Normal Form**
- 12.6. Inclusion Dependencies**
- 12.7. Other Dependencies and Normal Forms**
- 12.8. Summary**
- 12.9. Technical Terms**
- 12.10. Self-Assessment Questions**
- 12.11. Suggested Readings**

#### **12.1. INTRODUCTION:**

In relational database design, advanced concepts and algorithms are crucial for creating efficient, reliable, and scalable databases. While basic normalization (up to BCNF) addresses many common issues, further dependencies and sophisticated algorithms are needed to handle more complex scenarios and ensure optimal database performance.

Understanding and applying advanced dependencies, such as multivalued and join dependencies, along with robust design algorithms, are essential for maintaining data integrity and minimizing redundancy. These advanced techniques ensure that databases can efficiently manage intricate data relationships and constraints.

The chapter first covered began with 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.

## 12.2. PROPERTIES OF RELATIONAL DECOMPOSITIONS:

### 12.2.1 Lossless 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  $R$  into two or more relations  $R_1, R_2, \dots, R_n$  is said to have the lossless join property if, by joining these decomposed relations, we can exactly recreate the original relation  $R$  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.

### 12.2.2 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.

### 12.2.3 Desirability 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:**

Consider a relation  $R(A,B,C,D)$  with functional dependencies:

1.  $A \rightarrow B$  and  $B \rightarrow A$
2.  $C \rightarrow D$  and  $D \rightarrow C$

If we decompose  $R$  into  $R_1(A,B)$  and  $R_2(C,D)$ :

- The decomposition is desirable if it satisfies the lossless join property and dependency preservation.
- Joining  $R_1$  and  $R_2$  should reconstruct the original relation without introducing spurious tuples.
- The functional dependencies  $A \rightarrow B$  and  $C \rightarrow D$  should be enforceable in  $R_1$  and  $R_2$  respectively.

Desirable decomposition is fundamental to relational database design, ensuring that the benefits of normalization are fully realized while maintaining the integrity and performance of the database system.

**12.3. 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 schema is efficient, free from redundancy, and maintains data integrity. Key algorithms include those for testing the lossless join property, ensuring dependency preservation, and performing heuristic-based schema optimization.

**12.3.1 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  $R$  be decomposed into two or more relations  $R_1, R_2, \dots, R_n$ .

#### 2. Construct the Join Dependency Matrix:

- Create a matrix where each row represents an attribute in the original relation  $R$ , and each column represents a decomposed relation  $R_i$ .
- Initialize the matrix with zeros.

#### 3. Mark the Attributes:

- For each decomposed relation  $R_i$ , mark the columns corresponding to the attributes present in  $R_i$ .

#### 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:

$A \rightarrow B$  to  $B \rightarrow A$

$B \rightarrow C$  to  $C \rightarrow B$

#### 2. Decompose $R$ into $R_1(A,B)$ and $R_2(B,C)$ .

##### Identify the Decomposition:

- $R$  is decomposed into  $R_1(A,B)$  and  $R_2(B,C)$ .

### Construct the Join Dependency Matrix:

	$R_1(A, B)$	$R_2(B, C)$
A	0	0
B	0	0
C	0	0

#### 3. Mark the Attributes:

- For  $R_1(A,B)$ :
  - 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

#### 4. 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.

### 12.3.2 Algorithm 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 R into a set of relations R1, R2, ..., Rn.

**3. Project Functional Dependencies:**

- For each decomposed relation  $R_i$ , compute the projection of  $F$  on  $R_i$ , denoted as  $F_i$ . The projection of  $F$  on  $R_i$  includes all functional dependencies in  $F$  that involve only attributes of  $R_i$ .

**4. Compute the Closure:**

- Compute the closure of the union of the projected dependencies  $F_1 \cup F_2 \cup \dots \cup F_n$ , denoted as  $(F_1 \cup F_2 \cup \dots \cup F_n)^+$ .

**5. Check for Dependency Preservation:**

- Verify that every functional dependency in  $F$  is included in the closure  $(F_1 \cup F_2 \cup \dots \cup F_n)^+$ . If all dependencies in  $F$  are present in the closure, then the decomposition preserves dependencies.

**Example**

Consider a relation  $R(A,B,C)$  with functional dependencies:

1.  $A \rightarrow B$
2.  $B \rightarrow C$

Decompose  $R$  into  $R_1(A,B)$  and  $R_2(B,C)$ .

**1. Identify the Functional Dependencies:**

- $F = \{A \rightarrow B, B \rightarrow C\}$

**2. Decompose the Relation:**

- Decomposed relations are  $R_1(A,B)$  and  $R_2(B,C)$

**3. Project Functional Dependencies:**

- For  $R_1(A,B)$ :
  - Projection  $F_1 = \{A \rightarrow B\}$
- For  $R_2(B,C)$ :
  - Projection  $F_2 = \{B \rightarrow C\}$

**4. Compute the Closure:**

- $F_1 \cup F_2 = \{A \rightarrow B, B \rightarrow C\}$
- Compute the closure  $(F_1 \cup F_2)^+$ :
  - Start with  $\{A \rightarrow B, B \rightarrow C\}$
  - From  $A \rightarrow B$  and  $B \rightarrow C$ , by transitivity, derive  $A \rightarrow C$
  - $(F_1 \cup F_2)^+ = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$



### 5. Check for Dependency Preservation:

- Verify that every functional dependency in  $F$  is in  $(F_1 \cup F_2)^+$  :
  - $A \rightarrow B$  is in  $(F_1 \cup F_2)^+$
  - $B \rightarrow C$  is in  $(F_1 \cup F_2)^+$
  - 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.

### 12.3.3 Heuristic Algorithms for Schema Design

Heuristic algorithms for schema design provide practical approaches to database normalization and schema optimization. These algorithms use rule-of-thumb strategies to decompose relations, aiming to balance the benefits of normalization with the practical considerations of performance and complexity. While they may not always produce the theoretically optimal solution, heuristic algorithms are valuable for efficiently managing real-world database design tasks.

### 12.4. MULTIVALUED DEPENDENCIES AND FOURTH NORMAL FORM:

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  $A \twoheadrightarrow B$  implies that for each value of  $A$ , there is a set of values for  $B$  that can vary independently of other attributes.

#### 12.4.1 Examples of MVDs

Consider a relation  $R(\text{Student}, \text{Course}, \text{Hobby})$

- A student can enroll in multiple courses and have multiple hobbies independently.
- If "John" is enrolled in "Math" and "Science" and has hobbies "Reading" and "Swimming," then the MVD  $\text{Student} \twoheadrightarrow \text{Course}$  and  $\text{Student} \twoheadrightarrow \text{Hobby}$  exist.

#### 12.4.2 Fourth Normal Form (4NF) and Decomposition

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.

### Example of Decomposition to 4NF

Given the relation  $R(\text{Student}, \text{Course}, \text{Hobby})$  with MVDs  $\text{Student} \twoheadrightarrow \text{Course}$  and  $\text{Student} \twoheadrightarrow \text{Hobby}$  :

#### 1. Original Relation:

Student	Course	Hobby
John	Math	Reading
John	Math	Swimming
John	Science	Reading
John	Science	Swimming

#### 2. Decompose into Two Relations:

- $R_1(\text{Student}, \text{Course})$ :

Student	Course
John	Math
John	Science

- $R_2(\text{Student}, \text{Hobby})$ :

Student	Hobby
John	Reading
John	Swimming

**Fig. 12.1 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.

## 12.5. JOIN DEPENDENCIES AND FIFTH NORMAL FORM:

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.

### 12.5.1 Examples of JDs

Consider a relation  $R(A,B,C,D)$  with a JD such that  $R$  can be decomposed into  $R_1(A,B)$ ,  $R_2(B,C)$ , and  $R_3(C,D)$  and perfectly reconstructed by joining  $R_1$ ,  $R_2$ , and  $R_3$ .

#### 1. Original Relation:

A	B	C
1	X	10
2	Y	20
1	X	20
2	Y	10

#### 2. Decomposed Relations:

- $R_1(A, B)$ :

A	B
1	X
2	Y

- $R_2(B, C)$ :

B	C
X	10
Y	20
X	20
Y	10

- $R_3(A, C)$ :

A	C
1	10
2	20
1	20
2	10

Fig. 12.2 Result of JD with 4NF

### 12.5.2 Fifth Normal Form (5NF) and Decomposition

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.

#### Example of Decomposition to 5NF

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. 12.3 Result of 5NF**

## 12.6. 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.

Inclusion Dependencies can be formally defined as follows: Given two relations  $R1$  and  $R2$ , 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  $R1[A] \subseteq R2[B]$ .

### 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.

### 12.6.1 Applications and Examples

Inclusion dependencies are used to enforce referential integrity, such as ensuring that every order in an Orders table references an existing customer in a Customers table.

#### Enforcing Referential Integrity:

**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)  
);
```

### 12.6.2 Enforcement of Inclusion Dependencies

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, INs help prevent data anomalies and enforce relationships between different entities in the database schema.

## 12.7. OTHER DEPENDENCIES AND NORMAL FORMS:

### 12.7.1 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.

#### 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.

### 12.7.2 Template Dependencies

Template dependencies define patterns of constraints that generalize functional dependencies and other types of constraints, providing a flexible way to model complex relationships.

#### Benefits

- **Complex Integrity Constraints:** Allows for the specification of complex rules that govern the relationships between attributes.
- **Data Quality:** Ensures high data quality by enforcing specific patterns and combinations of values.
- **Business Logic Enforcement:** Directly supports the enforcement of business rules within the database schema.

Template dependencies provide a powerful mechanism for capturing and enforcing complex integrity constraints in relational databases. By specifying allowable patterns of attribute values, template dependencies ensure that data adheres to defined business rules and relationships, enhancing data quality and integrity.



### 12.7.3 Generalized Dependency Constraints

These constraints extend beyond traditional dependencies to capture more complex data relationships and integrity rules in a relational database.

#### Benefits

- **Enhanced Data Integrity:** Provides a robust framework for enforcing a wide range of integrity constraints, ensuring high data quality and consistency.
- **Flexibility:** Allows for the modeling of complex and nuanced relationships that go beyond simple FDs, MVDs, and JDs.
- **Comprehensive Rule Enforcement:** Supports the enforcement of comprehensive business rules directly within the database schema.

Generalized dependency constraints offer a powerful and flexible approach to maintaining data integrity and enforcing complex relationships within a relational database. By incorporating a broad range of dependency types, these constraints ensure that the database accurately reflects the underlying business rules and data relationships, leading to a more reliable and effective database design.

### 12.8. SUMMARY:

Review of the main concepts covered, including advanced dependencies, normalization forms, and design algorithms. Emphasizing the significance of understanding and applying these concepts for creating efficient, reliable, and scalable databases. Concluding remarks on the critical role of advanced relational database design in modern data management. This chapter provides an in-depth exploration of advanced relational database design principles, algorithms, and dependencies, equipping readers with the knowledge to design highly efficient and reliable database schemas.

### 12.9. TECHNICAL TERMS:

Advanced dependencies, Loss less property, 4NF, 5NF, Join Dependency, Decomposition, Data Integrity.

### 12.10. SELF ASSESSMENT QUESTIONS:

#### Essay questions:

- 1) Illustrate about 4NF
- 2) Describe about 5NF
- 3) Explain about Join Dependency

#### Short Notes:

- 1) Write Lossless property
- 2) Write algorithm for Dependency Preservation

**12.11. 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". 8<sup>th</sup> Edition. Addison-Wesley.
- 3) Silberschatz, A., Korth, H. F., & Sudarshan, S. (2010). "Database System Concepts". 6<sup>th</sup> Edition. McGraw-Hill.
- 4) Ullman, J.D., & Widom, J. (2008). "A First Course in Database Systems". 3<sup>rd</sup> Edition. Pearson.

**Dr. Neelima Guntupalli**

## LESSON-13

# INTRODUCTION TO TRANSACTION PROCESSING CONCEPTS AND THEORY

### AIMS AND OBJECTIVES:

The primary aim of this chapter, “Introduction to Transaction Processing Concepts and Theory,” is to provide a comprehensive understanding of the fundamental concepts, principles, and theories related to database transaction processing. It seeks to equip readers with the knowledge necessary to comprehend how transactions are managed in database systems to ensure consistency, reliability, and efficiency, even in complex environments involving concurrency and potential failures.

### The specific objectives of this chapter are:

- 1) Introduction to Transaction Processing
- 2) Understanding Transaction and System Concepts
- 3) Exploration of Desirable Transaction Properties
- 4) Characterization of Schedules Based on Recoverability
- 5) Characterization of Schedules Based on Serializability

### STRUCTURE:

#### 13.1. Introduction

#### 13.2. Introduction to Transaction Processing

#### 13.3. Transaction and System Concepts

#### 13.4. Desirable Properties of Transactions

#### 13.5. Characterizing Schedules based on Recoverability

#### 13.6. Characterizing schedules based on Serializability

#### 13.7. Summary

#### 13.8. Technical Terms

#### 13.9. Self-Assessment Questions

#### 13.10. Suggested Readings

### 13.1. INTRODUCTION:

The lesson “*Introduction to Transaction Processing Concepts and Theory*” lays the foundation for understanding the mechanisms and principles underpinning database transaction processing. It introduces the concept of transactions, which represent a sequence of operations performed as a single logical unit of work in a database system. These operations must ensure consistency, durability, and atomicity even in the presence of system

failures or concurrent user interactions. The chapter also delves into the key system concepts that support transaction processing, such as concurrency control, recovery mechanisms, and the roles of schedules in coordinating transactions. These concepts are essential to maintaining the integrity and reliability of databases in real-world applications.

Additionally, the chapter explores the desirable properties of transactions, often encapsulated by the ACID properties (Atomicity, Consistency, Isolation, and Durability). It provides a detailed analysis of how transaction schedules can be characterized based on recoverability, ensuring that systems can revert to a consistent state after a failure. Another critical focus is on serializability, which determines whether a schedule of transactions ensures correctness by being equivalent to a serial execution. By examining these theoretical frameworks, the chapter equips readers with the tools to analyze and design robust transaction processing systems that meet the demands of modern, multi-user environments.

### **13.2. INTRODUCTION TO TRANSACTION PROCESSING:**

Transaction processing is a fundamental aspect of Database Management Systems (DBMS) that ensures the consistent, reliable, and efficient execution of operations in multi-user environments. A transaction is defined as a sequence of one or more operations performed on a database, treated as a single logical unit of work. These operations often involve retrieving, updating, or modifying data, and their execution must maintain the integrity of the database. Transaction processing is essential for applications that require concurrent access to shared data while preventing conflicts, data loss, or inconsistencies caused by system failures or concurrency issues.

Key concepts in transaction processing include the **ACID properties**-Atomicity, Consistency, Isolation, and Durability-which guarantee that transactions are executed reliably. Atomicity ensures that a transaction is either fully completed or entirely undone. Consistency ensures that the database remains in a valid state before and after a transaction. Isolation prevents concurrent transactions from interfering with each other, and Durability ensures that once a transaction is committed, its results are permanently recorded, even in the event of system crashes. To achieve these goals, DBMSs employ mechanisms like concurrency control, locking, and recovery strategies, ensuring robust transaction processing in diverse application scenarios.

In the context of transaction processing within a DBMS, the handling of transactions varies depending on whether the system operates in a single-user or multiuser environment. Concurrency, a critical aspect of multiuser systems, introduces additional challenges and solutions to maintain the integrity and efficiency of the database.

#### **Single-User System**

In a single-user system, only one transaction is processed at a time. There is no need for concurrency control, as the system inherently avoids conflicts between transactions by

sequentially executing them. This simplifies transaction management since issues like data inconsistencies or race conditions are non-existent. However, single-user systems are less efficient in environments requiring high throughput, as they cannot leverage the benefits of parallelism or handle multiple users simultaneously.

### Multiuser System

A multiuser system allows multiple transactions to be executed simultaneously by different users. This improves system efficiency and resource utilization, enabling higher throughput and responsiveness. However, it also introduces challenges related to **concurrency control**. Without proper management, concurrent transactions can lead to problems such as:

- **Lost Updates:** When two transactions simultaneously update the same data, one update may overwrite the other.
- **Dirty Reads:** When a transaction reads uncommitted data from another transaction, leading to inconsistencies if the latter is rolled back.
- **Non-repeatable Reads:** When data retrieved in one operation is changed by another transaction before the initial transaction completes.
- **Phantom Reads:** When new data is inserted or deleted by another transaction, causing inconsistencies in subsequent queries.

### Concurrency in Transaction Processing

Concurrency refers to the ability of the DBMS to allow multiple transactions to execute in parallel while ensuring the consistency and isolation of each transaction. This is achieved through **scheduling algorithms** and mechanisms like locking, timestamp ordering, and multiversion concurrency control. The goal of concurrency control is to ensure **serializability**, where the outcome of concurrent transactions is equivalent to their serial execution. Additionally, recovery mechanisms are implemented to ensure that transactions can recover gracefully from system crashes, preserving the database's integrity.

Effective transaction processing in multiuser environments is crucial for modern applications that require high performance and data consistency, such as e-commerce platforms, banking systems, and enterprise resource planning (ERP) systems.

#### ❖ Transaction

A transaction is a sequence of one or more operations (such as queries or updates) that are treated as a single, indivisible unit of work. Transactions are often initiated to maintain data consistency during operations like transferring money between accounts, placing an order in e-commerce, or updating inventory levels. A transaction must follow these key properties:

- **Atomicity:** Ensures that all operations within a transaction are completed or none at all.
- **Consistency:** Guarantees that the database transitions from one valid state to another.
- **Isolation:** Ensures that transactions do not interfere with each other.
- **Durability:** Makes sure the changes made by a committed transaction are permanent.

**❖ Basic operations are read and write**

Transactions often involve read and write operations:

- Read Operation: Retrieves data from the database (e.g., fetching account balance).
- Write Operation: Modifies data in the database (e.g., updating account balance after a withdrawal).

These operations are the building blocks of transactions. In a multiuser environment, improper handling of these operations can lead to data inconsistencies, such as dirty reads, lost updates, or non-repeatable reads.

In transaction processing, the operations `read_item(X)` and `write_item(X)` are fundamental primitives that define how a transaction interacts with a database. These operations ensure that the data manipulation is structured and allows the DBMS to maintain consistency, especially in concurrent transaction environments.

**➤ read\_item(X)**

The `read_item(X)` operation retrieves the value of a data item `X` from the database into a variable in the transaction's local memory. It does not alter the database; it only accesses the required data.

**Steps of read\_item(X):**

1. The database retrieves the value of `X`.
2. The value is copied into a variable in the transaction's workspace.

**For example:**

If a transaction needs to check the balance of an account, `X`, it performs a `read_item(X)` operation.

**➤ write\_item(X)**

The `write_item(X)` operation updates the value of a data item `X` in the database with the value from the transaction's local memory. This operation modifies the database and must ensure that changes are consistent.

**Steps of write\_item(X):**

1. The transaction modifies `X` in its local memory (workspace).
2. The updated value of `X` is written back to the database.

**For example:**

If a transaction withdraws an amount from an account, `X`, it updates `X` using a `write_item(X)` operation.

### Example Sequence of Operations

Consider a transaction T1 that transfers \$100 from account A to account BBB:

1. read\_item(A): Read the balance of account A.
2. write\_item(A): Update the balance of A after subtracting \$100.
3. read\_item(B): Read the balance of account B.
4. write\_item(B): Update the balance of B after adding \$100.

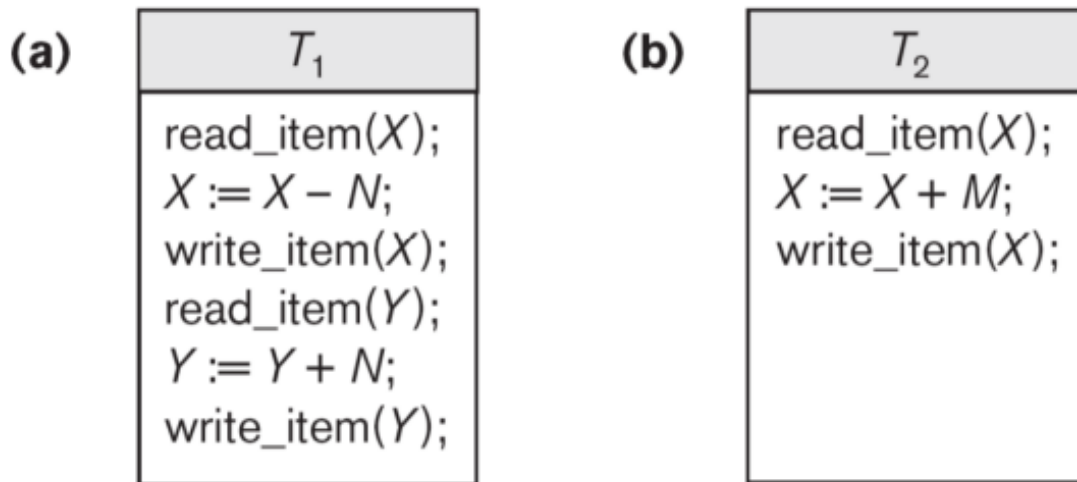


Fig. 13.1 Two sample transactions T1 and T2

#### ❖ Why Concurrency Control is Needed:

In a multiuser environment, concurrency arises when multiple transactions execute simultaneously. While concurrency improves system performance and resource utilization, it also introduces the potential for conflicts and data inconsistencies. Common issues include:

- **Lost Updates:** Two transactions simultaneously modify the same data, causing one update to overwrite the other.
- **Dirty Reads:** A transaction reads uncommitted changes made by another transaction.
- **The Incorrect Summary Problem:** If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

Concurrency control mechanisms, such as locking protocols, timestamp ordering, and multiversion concurrency control (MVCC), ensure that concurrent transactions produce results equivalent to serial execution, preserving database consistency.

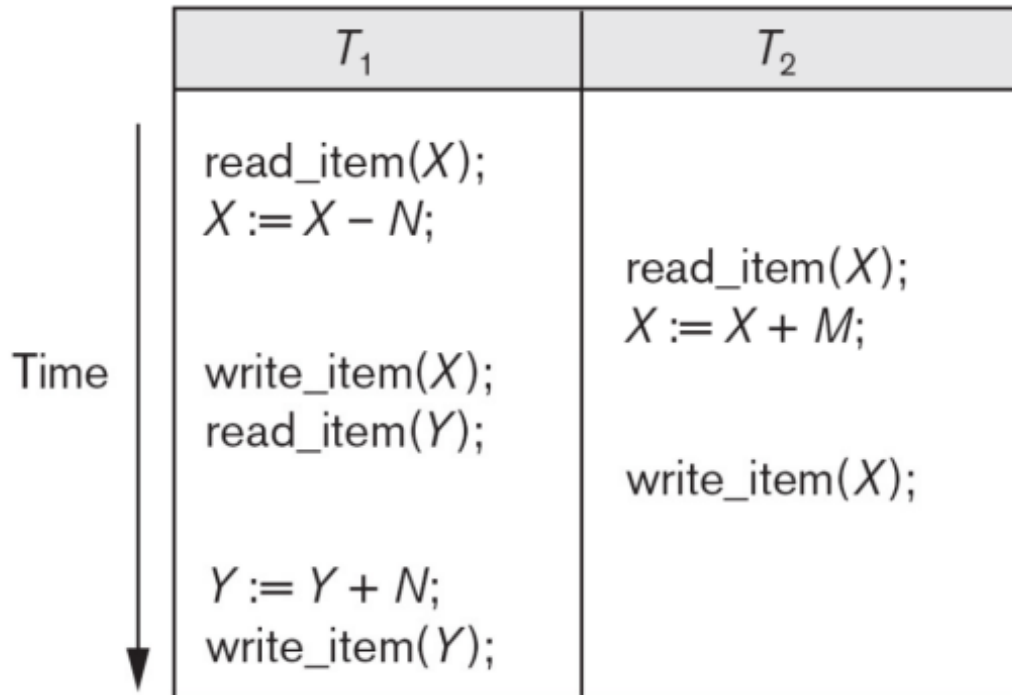


Fig. 13.2 The Lost update problem:  $X$  has wrong value its updated  $T_1$  failed

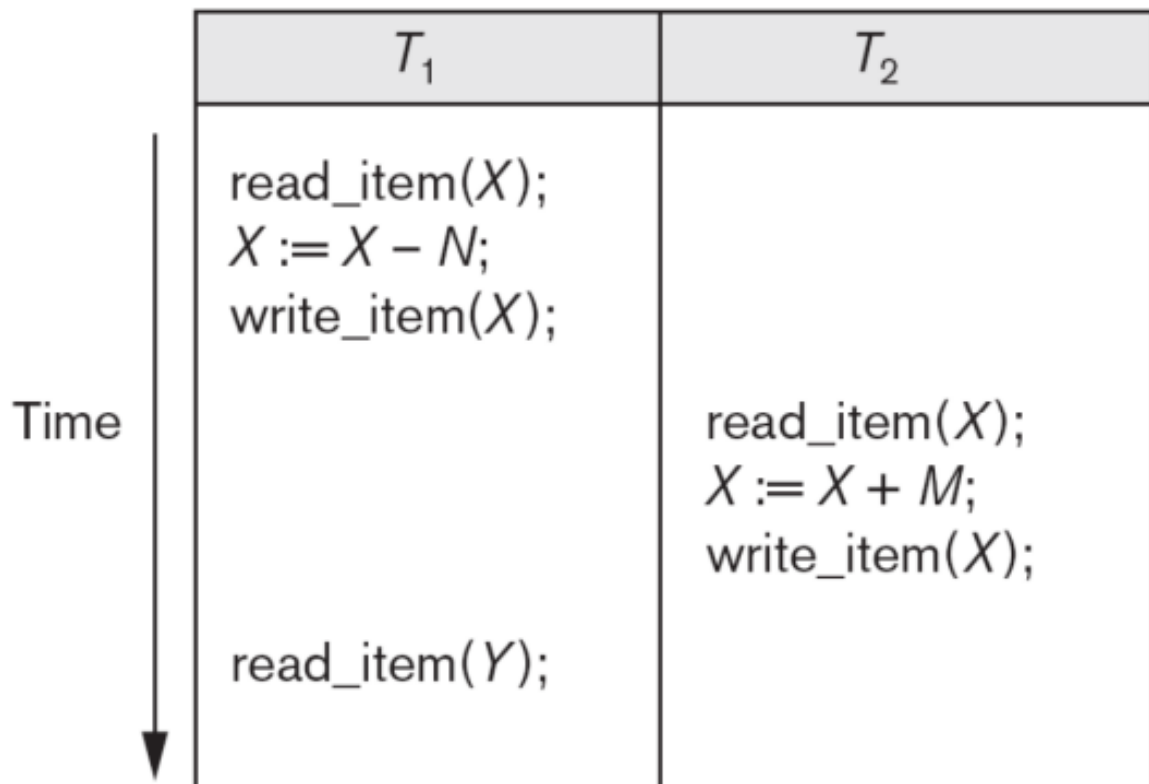


Fig. 13.3 The Dirty Read:  $T_1$  failed had old value  $T_2$  read incorrect  $X$  value



$T_1$	$T_3$
<pre> read_item(X); X := X - N; write_item(X);  read_item(Y); Y := Y + N; write_item(Y); </pre>	<pre> sum := 0; read_item(A); sum := sum + A; . . . read_item(X); sum := sum + X; read_item(Y); sum := sum + Y; </pre>

**Fig. 13.4 The Incorrect Summary Problem**

#### ❖ Why Recovery is Needed

Database recovery is essential to protect data integrity in the event of failures such as system crashes, power outages, or software bugs. Recovery mechanisms are required to ensure:

- **Atomicity:** Incomplete transactions are rolled back to avoid partial updates.
- **Durability:** Committed transactions remain persistent despite failures.
- **Consistency:** The database is restored to a valid state after recovery.

Without recovery mechanisms, data loss, corruption, or inconsistencies may occur, severely impacting the reliability of the database system. Techniques like transaction logs, checkpoints, and redo/undo operations are commonly employed to facilitate effective recovery.

### 13.3. TRANSACTION AND SYSTEM CONCEPTS:

Transactions are the basic units of work in a DBMS that must be executed in a way that ensures the consistency and reliability of the database. The system must manage transactions effectively to prevent issues such as partial updates or conflicts in a multiuser environment.

Various states, recovery techniques, and system components like logs and recovery mechanisms ensure that transactions adhere to the ACID properties.

#### ❖ **Transaction States**

A transaction goes through several states during its lifecycle:

1. **Active:** The transaction is being executed and operations are ongoing.
2. **Partially Committed:** All operations of the transaction are executed, but it is not yet permanently saved (commit pending).
3. **Committed:** The transaction is successfully completed, and changes are permanently saved to the database.
4. **Failed:** The transaction cannot proceed due to an error or system failure.
5. **Aborted:** The transaction is terminated, and all changes made during it are rolled back.
6. **Terminated:** The transaction has completed, either successfully (committed) or unsuccessfully (aborted).

#### ❖ **Recovery Manager Keeps Track of the Following Operations**

The recovery manager is responsible for maintaining the integrity and consistency of the database in the event of failures. It tracks:

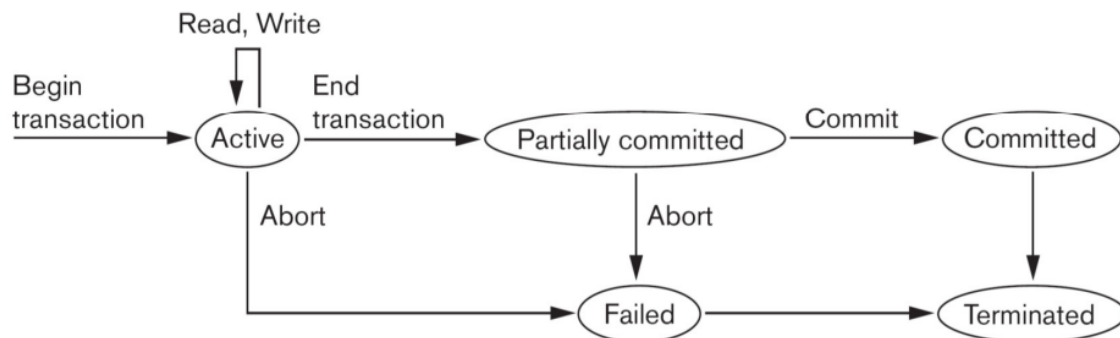
1. **Transaction Start:** When a transaction begins.
2. **Read and Write Operations:** To log the data items accessed or modified.
3. **Transaction Commit or Abort:** To record the final status of the transaction.
4. **Checkpointing:** Periodically saving the current state of the database to facilitate recovery.
5. **Undo and Redo Actions:** For rolling back incomplete transactions or reapplying changes of committed transactions.

#### ❖ **Recovery Techniques**

Recovery techniques are methods to restore the database to a consistent state after a failure. Common techniques include:

1. **Deferred Update:** Changes made by transactions are not written to the database until the transaction commits.
2. **Immediate Update:** Changes are written to the database as soon as they occur, with a log to ensure recovery in case of failure.
3. **Shadow Paging:** A separate copy (shadow) of the database pages is maintained, which replaces the original only after a successful transaction commit.

4. **Checkpointing:** A snapshot of the current database state is periodically recorded, reducing the amount of log processing required during recovery.



**Fig. 13.5 States for Transaction Execution State transition Diagram**

#### ❖ The System Log

The system log is a sequential record of all transaction operations and system activities. It is essential for recovery and typically stores:

1. Transaction Start and End Records.
2. Before-Image and After-Image: The state of data items before and after updates.
3. Commit and Abort Records: To indicate transaction status.

#### Types of log record:

- [start\_transaction,T]: Records that transaction T has started execution.
- [write\_item,T,X,old\_value,new\_value]: Records that transaction T has changed the value of database item X from old\_value to new\_value.
- [read\_item,T,X]: Records that transaction T has read the value of database item X.
- [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- [abort,T]: Records that transaction T has been aborted.

#### ❖ Recovery Using Log Records

The system log facilitates two primary recovery actions:

1. **Undo Operations:** Rolling back uncommitted transactions to revert changes using the *before-images* stored in the log.
2. **Redo Operations:** Reapplying committed transactions to ensure durability using the *after-images* stored in the log.

#### ❖ Commit Point of a Transaction

The commit point is the stage where all changes made by a transaction are permanently recorded in the database. Once a transaction reaches its commit point:

1. It is considered successfully completed.
2. Its effects are guaranteed to persist even in the case of a system failure.
3. The system writes a "commit" record to the log to indicate the transaction's final state.

The commit point is crucial for ensuring atomicity and durability, as it marks the transition from a transient to a permanent state of data updates.

#### 13.4. DESIRABLE PROPERTIES OF TRANSACTIONS:

- **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation:** A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation:** A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
- **Durability or permanency:** Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

#### 13.5. CHARACTERIZING SCHEDULES BASED ON RECOVERABILITY:

In transaction processing, a schedule refers to the sequence in which transactions' operations (read and write) are executed, particularly in a multi-user environment. The recoverability of a schedule determines whether the system can restore a consistent database state after a failure. A recoverable schedule ensures that committed transactions do not lead to inconsistencies, even if some transactions must be rolled back due to errors or system crashes.

##### ❖ Transaction Schedule

- ❖ A transaction schedule is an arrangement of the operations (read and write) of a set of transactions such that the order of operations in each transaction is preserved. Schedules are critical in concurrent environments because they affect the consistency and recoverability of the database.

##### Properties of a Schedule:

1. Operations within each transaction must appear in the same order as in the original transaction.
  2. Operations from different transactions may interleave to improve concurrency.
  3. Schedules are evaluated based on their correctness and whether they preserve consistency, often using concepts like serializability and recoverability.
- **Schedules Classified Based on Recoverability**

Schedules can be classified into three categories based on their recoverability:

##### ➤ Recoverable Schedules

A schedule is recoverable if:

- A transaction  $T_i$  commits only after all transactions  $T_j$ , from which  $T_i$  has read data, have committed.
- This ensures that  $T_i$  does not depend on uncommitted or invalid data.

**For example:**

**T1: R(A), W(A), Commit**

**T2: R(A), W(B), Commit**

Here, T2 commits only after T1, making the schedule recoverable.

#### ➤ Cascadeless Schedules

A schedule is cascadeless if:

- Transactions do not read uncommitted data from other transactions.
- Cascadeless schedules avoid the cascading rollback problem, where one transaction's failure causes a chain of rollbacks.

**For example:**

**T1: W(A)**

**T2: R(A) (after T1 commits)**

T2 waits for T1 to commit before reading A, ensuring no cascading rollbacks.

#### ➤ Strict Schedules

A schedule is strict if:

- Transactions do not read or write a data item until all transactions that previously wrote to the data item have either committed or aborted.
- Strict schedules simplify recovery by ensuring that uncommitted changes are never accessed.

**For example:**

**T1: W(A), Commit**

**T2: R(A) (only after T1 commits)**

T2 does not access A until T1 has committed, ensuring strictness.

Schedule Type	Key Characteristics	Example	Recovery Impact
Recoverable	Transactions can commit only after transactions they depend on have committed.	$T1 \rightarrow T2$ where $T1$ commits before $T2$ .	Safe recovery, but may require undo operations.
Cascadeless	No transaction reads uncommitted data from another transaction.	$T1 \rightarrow T2$ , both commit in order.	Prevents cascading rollbacks.
Strict	No transaction reads or writes data from another unless that transaction has committed.	$T1 \rightarrow T2$ , both commit.	Simplifies recovery and guarantees consistency.
Unrecoverable	A transaction commits while relying on uncommitted data from another transaction.	$T1 \rightarrow T2$ where $T2$ commits before $T1$ .	Leads to potential inconsistencies and difficult recovery.

**Fig. 13.6 Summary of Schedule Classifications Based on Recoverability**

### Significance of Recoverability in Schedules

- **Avoiding Inconsistencies:** Recoverable schedules prevent the database from reaching an inconsistent state by ensuring that committed transactions depend only on valid data.
- **Simplifying Recovery:** Cascadeless and strict schedules reduce the complexity of recovery by minimizing or eliminating cascading rollbacks.
- **Concurrency vs. Safety:** While maximizing concurrency is desirable, it must be balanced with the need for recoverable and consistent schedules to ensure system reliability.

By categorizing schedules based on recoverability, DBMSs ensure that even in failure scenarios, data consistency and transaction integrity are maintained.

### 13.6. CHARACTERIZING SCHEDULES BASED ON SERIALIZABILITY:

Serializability is a key concept in transaction processing that evaluates whether a given schedule of concurrent transactions is correct. A schedule is serializable if its execution results in a database state equivalent to one produced by a serial schedule, where transactions are executed sequentially without interleaving. This ensures consistency in concurrent transaction processing.

**Serializability** is a key property in transaction processing that ensures a schedule (a sequence of operations from multiple transactions) is equivalent to a serial schedule in terms of the final outcome, even if the transactions are interleaved. A schedule is **serializable** if its execution produces the same results as some serial execution of those transactions. Ensuring serializability is crucial for maintaining database consistency and correctness in multi-transaction environments.

Schedules can be classified based on serializability into different types, primarily based on **conflict serializability** and **view serializability**. Let's explore these classifications in detail:

- **Conflict Serializability**

A schedule is **conflict-serializable** if it can be transformed into a serial schedule by swapping non-conflicting operations. The key here is that **conflict** arises when two operations from different transactions are on the same data item and at least one of them is a write operation.

#### Key Characteristics of Conflict Serializability:

- Two operations are in conflict if they involve the same data item, and at least one of the operations is a write.
- A schedule is conflict-serializable if there exists a serial schedule that is equivalent to it in terms of the final database state.

- **Conflict equivalence** is based on the ability to swap non-conflicting operations without changing the final outcome.

**Conflicting Operations:**

1. **Read-write conflict:** When one transaction writes a data item while another reads the same item.
2. **Write-write conflict:** When two transactions write to the same data item.

**Example of Conflict-Serializable Schedule:**

**T1: W(A)**

**T2: W(B)**

**T1: W(B)**

**T2: R(A)**

In this schedule, the read and write operations from T2 and T1 on A and B can be reordered (by swapping T2:R(A) and T1:W(B) to produce a serial schedule, meaning it is conflict-serializable.

- **View Serializability**

A schedule is **view-serializable** if it produces the same final result as some serial execution of the same transactions, considering the data read by each transaction and the transactions' commit points. Unlike conflict serializability, view serializability considers the **read-from relationship** and **commit dependencies** between transactions.

**Key Characteristics of View Serializability:**

- A schedule is view-equivalent to a serial schedule if the read-from relationships are maintained, and the final database state matches the serial execution.
- View serializability is broader than conflict serializability. All conflict-serializable schedules are view-serializable, but the reverse is not always true.

**View Equivalence:**

- **Read-from Relationship:** For every data item X, if T<sub>j</sub> reads X from T<sub>i</sub> this relationship must be preserved in the equivalent serial schedule.
- **Commit Dependency:** Transactions must commit in the same order as they would in a serial schedule.

**Example of View-Serializable Schedule:**

**T1: W(A), Commit**

**T2: R(A), Commit**

**T3: R(A), Commit**

This schedule may be view-equivalent to the serial schedule where T1 runs first, followed by T2, and T3. The **read-from** relationship between T2 and T3 is maintained, making this view-serializable.

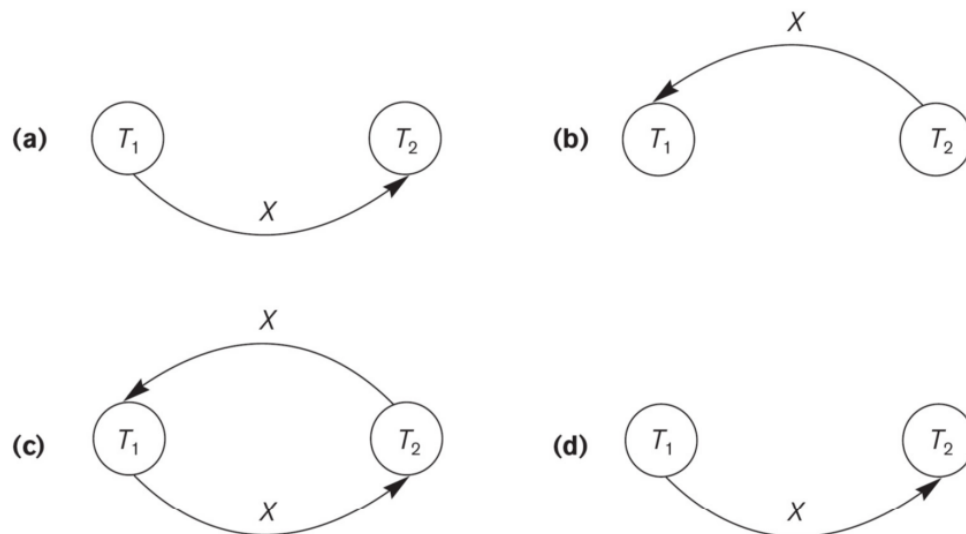
- **Testing for Conflict Serializability**

To test whether a schedule is conflict-serializable, the **precedence graph** (or **serializability graph**) is typically used. This graph is constructed by:

- **Nodes:** Representing each transaction.
- **Edges:** Directed edges between transactions representing conflicts (i.e., when two transactions perform conflicting operations on the same data item).

**Algorithm:**

- Looks at only read\_Item (X) and write\_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from  $T_i$  to  $T_j$  if one of the operations in  $T_i$  appears before a conflicting operation in  $T_j$
- The schedule is serializable if and only if the precedence graph has no cycles.



**Fig. 13.7 Precedence Graph (a) serial schedule A (b) serial schedule B (c) not serializable schedule C (d) serializable, equivalent to schedule A, schedule D**

- **Other Types of Equivalence of Schedules**

In addition to conflict serializability and view serializability, there are other types of equivalence that relate to the ordering of transactions, including:

- **Serializable Schedule:** A schedule that is either conflict-serializable or view-serializable is considered serializable. The final goal is to ensure that the interleaved execution of transactions does not violate database consistency.



- **Causal Serializability:** A schedule is causally serializable if it preserves the causal dependencies between transactions. This is a stricter form of serializability, considering the interdependencies between transactions based on their read/write operations.

Schedule Type	Key Characteristics	Example	Testing Mechanism
<b>Conflict-Serializable</b>	Can be transformed into a serial schedule by swapping non-conflicting operations.	$T1 : W(A), T2 : W(B), T1 : W(B), T2 : R(A)$	Precedence graph (check for cycles).
<b>View-Serializable</b>	Produces the same final state as a serial schedule, considering read-from relationships and commit dependencies.	$T1 : W(A), T2 : R(A)$ , both commit in order.	More complex than conflict serializability, based on read-from relationships.
<b>Serializable</b>	Includes both conflict-serializable and view-serializable schedules.	Any conflict-serializable schedule is serializable.	Includes both conflict and view serializability tests.
<b>Causal Serializability</b>	Preserves the causal dependencies between transactions.	Similar to view serializability but with stricter causality rules.	Involves tracking causality of transaction operations.

**Fig. 13.8 Summary of Schedule Classifications Based on Serializability**

The classification of schedules based on serializability is fundamental to ensuring that transaction processing systems behave correctly in a multi-user environment. Conflict-serializability is the most used approach due to its simpler nature and efficient testing using precedence graphs. View-serializability provides a broader concept of equivalence but is more complex to test. Ultimately, ensuring serializability helps maintain database consistency, which is essential for reliable transaction processing systems.

### 13.7. SUMMARY:

The **Introduction to Transaction Processing Concepts and Theory** covers the fundamental principles behind managing transactions within a database management system (DBMS). It begins by defining **transaction processing** and explaining key concepts, such as **transactions** (units of work) and **systems** (the environment in which transactions execute). The chapter highlights the **desirable properties of transactions**, including **atomicity**, **consistency**, **isolation**, and **durability** (ACID properties), which ensure the integrity and reliability of transactions. It also delves into **recoverability**, outlining how schedules (sequences of operations) are classified to prevent inconsistencies, and **serializability**, which ensures that interleaved transactions produce results equivalent to serial executions. Overall, the chapter introduces essential concepts for ensuring that multiple transactions can be executed concurrently without compromising database consistency and correctness.

**13.8. TECHNICAL TERMS:**

- Transaction
- Atomicity
- Consistency
- Isolation
- durability
- recoverability
- schedules
- serializability

**13.9. SELF ASSESSMENT QUESTIONS:****Essay questions:**

- 1) Explain the fundamental concepts of transaction processing and its importance in DBMS. How do transactions interact with the system?
- 2) Discuss the desirable properties of transactions (ACID properties). How do these properties ensure the reliability and consistency of database systems?
- 3) Describe schedules in transaction processing. How are schedules classified based on recoverability, and why is recoverability important for maintaining database consistency?
- 4) Explain conflict serializability and its significance in ensuring that a transaction schedule produces the same result as a serial execution of transactions. Provide an example to illustrate conflict serializability.
- 5) How does view serializability differ from conflict serializability? Discuss the advantages and disadvantages of using view serializability in transaction processing.
- 6) Define the concept of serializability in transaction processing. Discuss the relationship between conflict serializability and view serializability in ensuring database consistency during concurrent transaction execution.

**Short Notes:**

- 1) What is transaction processing in database management systems (DBMS)?
- 2) Define a transaction in the context of DBMS.
- 3) What are the ACID properties in transaction processing?
- 4) Explain the concept of atomicity in a transaction.
- 5) What is the difference between a single-user system and a multiuser system in transaction processing?

**13.10. 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". 8<sup>th</sup> Edition. Addison-Wesley.
- 3) Silberschatz, A., Korth, H.F., & Sudarshan, S. (2010). "Database System Concepts". 6<sup>th</sup> Edition. McGraw-Hill.
- 4) Ullman, J.D., & Widom, J. (2008). "A First Course in Database Systems". 3<sup>rd</sup> Edition. Pearson.

**Mrs. Appikatla Pushpa Latha**

# LESSON-14

## CONCURRENCY CONTROL TECHNIQUES

### AIMS AND OBJECTIVES:

The aim of studying **Concurrency Control Techniques** in database management systems (DBMS) is to understand how to manage multiple transactions executing concurrently without compromising the consistency, integrity, and isolation of the data. Concurrency control ensures that the database remains in a valid state even when transactions are interleaved, preventing issues like race conditions, deadlocks, and inconsistencies.

By the end of the study of Concurrency Control Techniques, the learner should be able to:

1. Explain the importance of concurrency control in a multi-user database system.
2. Understand and apply **Two-Phase Locking (2PL)** techniques for ensuring serializability and preventing conflicts in concurrent transactions.
3. Describe the principles behind **Timestamp Ordering** and its application for controlling concurrency and ensuring transaction ordering.
4. Analyze **Multiversion Concurrency Control (MVCC)** as a method to improve performance in read-heavy environments and reduce contention.
5. Evaluate **Validation Concurrency Control** and its suitability for applications with low conflict rates.
6. Discuss the concept of **Granularity** in data locking and understand the benefits and challenges of **Multiple Granularity Locking** in managing concurrency.
7. Compare the strengths and weaknesses of different concurrency control techniques and understand their practical applications in DBMS design and transaction management.

### STRUCTURE:

- 14.1 Introduction
- 14.2 Two Phase Locking Techniques for Concurrency Control
- 14.3 Concurrency Control Based on Timestamp Ordering
- 14.4 Multiversion Concurrency control techniques
- 14.5 Validation concurrency control Techniques
- 14.6 Granularity of Data Items and multiple Granularity Locking.
- 14.7 Summary
- 14.8 Technical Terms
- 14.9 Self-Assessment Questions
- 14.10 Suggested Readings

#### 14.1 INTRODUCTION:

The chapter on **Concurrency Control Techniques** explores various methods used to manage the simultaneous execution of transactions in a database system. **Two-Phase Locking (2PL)** is one of the foundational techniques for ensuring serializability, where transactions acquire locks on data items in a first phase and release them in a second phase. This protocol ensures

that once a transaction releases a lock, it cannot acquire any more, thus preventing conflicts and maintaining transaction consistency. The chapter also discusses **Concurrency Control Based on Timestamp Ordering**, which resolves conflicts by assigning timestamps to transactions. This method ensures that transactions are executed in a logical order based on their timestamps, guaranteeing serializability without the need for locking. Additionally, **Multiversion Concurrency Control (MVCC)** is introduced as a technique that allows multiple versions of a data item to exist, reducing contention and improving performance, particularly in read-heavy workloads where transactions can access different versions of the same data item without blocking each other.

The chapter also covers **Validation Concurrency Control Techniques**, which involve validating a transaction's read and write operations before committing them, ensuring that the transaction would not violate serializability if executed in isolation. This technique minimizes the need for locks, making it useful in systems with low contention or when transactions are mostly independent. Furthermore, the concept of **Granularity of Data Items** is examined, where the size of data units (such as rows, pages, or entire tables) determines the level of locking. **Multiple Granularity Locking** is introduced as a method that allows for locks at different levels of granularity, offering a trade-off between fine-grained control and efficient resource usage. By understanding these techniques, the chapter equips readers with a comprehensive toolkit for ensuring the proper execution of concurrent transactions while maintaining database integrity, efficiency, and isolation in multi-user environments.

## 14.2. TWO PHASE LOCKING TECHNIQUES FOR CONCURRENCY CONTROL:

❖ **Two-Phase Locking (2PL)** is a concurrency control protocol used in database management systems (DBMS) to ensure **serializability** and to avoid conflicts between concurrently executing transactions. The essential components of the 2PL protocol are:

### 1. Locking Phase:

- In the first phase, called the **growing phase**, a transaction can acquire locks on data items, but it cannot release any locks. The transaction requests locks as needed, ensuring that no other transactions can modify the data items it holds a lock on.

### 2. Unlocking Phase:

- In the second phase, called the **shrinking phase**, the transaction releases the locks it holds and cannot acquire any new locks. Once a transaction starts releasing locks, it enters the shrinking phase, and no further locking is allowed. This prevents a situation where a transaction could change its set of locked items after releasing a lock, maintaining the order and ensuring consistency.

The key property of 2PL is that it guarantees **serializability**, ensuring that the interleaving of transactions will produce a result equivalent to some serial execution of those transactions.

However, 2PL may lead to **deadlocks** (if two transactions are waiting for each other to release locks) and **lock contention**.

```
B:if LOCK (X) = 0 (*item is unlocked*)
  then LOCK (X) ← 1 (*lock the item*)
  else begin
    wait (until lock (X) = 0) and
    the lock manager wakes up the transaction);
  goto B
end;
```

**Fig. 14.1 Two phase Locking Example**

#### ❖ Two-Phase Locking Algorithm

The 2PL protocol can be described in terms of an algorithm for managing locks during the execution of a transaction:

##### 1. **Transaction Initialization:**

- A transaction starts, and the DBMS assigns it a unique transaction identifier (TID).

##### 2. **Growing Phase** (Acquiring Locks):

- During the growing phase, the transaction requests and acquires locks on the data items it needs to access.
- **Lock types:** Common lock types are **shared locks** (S-locks) for read operations and **exclusive locks** (X-locks) for write operations.
- The transaction can acquire multiple locks but cannot release any locks during this phase.

##### 3. **Shrinking Phase** (Releasing Locks):

- Once the transaction releases a lock, it enters the shrinking phase. From this point, the transaction cannot request any new locks.
- It can only release the locks it holds as it completes its operations.

##### 4. **Commit or Abort:**

- When the transaction finishes its operations and releases all the locks, it commits to the database (making the changes permanent). If the transaction fails or is aborted, it is rolled back, and any changes made are discarded.

#### **Algorithm:**

##### 1. **Start Transaction:**

- Begin transaction  $T_i$  (with transaction ID  $TID_i$ ).

## 2. Growing Phase:

- While the transaction is executing, it can request locks on data items.
  - **Request Lock:** If the transaction requires access to a data item (e.g., read or write), it requests the appropriate lock (shared or exclusive) on that item.
  - **Lock Granted:** If no conflicting locks are held by other transactions, the requested lock is granted.
  - **Wait for Lock:** If another transaction holds a conflicting lock on the data item, the requesting transaction waits for the lock to be released.

## 3. Shrinking Phase:

- Once the transaction releases its first lock, it enters the shrinking phase, where it cannot acquire any new locks.
  - **Release Lock:** After the transaction completes the operation on a data item, it releases the lock.
  - **Commit:** Once all operations are complete and the transaction has released all locks, it commits.

## 4. Transaction Completion:

- The transaction is either committed (changes are permanent) or aborted (changes are discarded).

```
B: if LOCK (X) = "unlocked" then
begin LOCK (X) ← "read-locked";
  no_of_reads (X) ← 1;
end
else if LOCK (X) ← "read-locked" then
  no_of_reads (X) ← no_of_reads (X) +1
else begin wait (until LOCK (X) = "unlocked" and
  the lock manager wakes up the transaction);
  go to B
end;
```

**Fig. 14.2 Two phase Locking with read\_lock**

```

if LOCK (X) = "write-locked" then
  begin LOCK (X) ← "unlocked";
    wakes up one of the transactions, if any
  end
else if LOCK (X) ← "read-locked" then
  begin
    no_of_reads (X) ← no_of_reads (X) -1
    if no_of_reads (X) = 0 then
      begin
        LOCK (X) = "unlocked";
        wake up one of the transactions, if any
      end
    end
  end;
end;

```

Fig. 14.3 Two phase Locking with write\_lock

<u>T1</u>	<u>T2</u>	<u>Result</u>
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	

Fig. 14.4 Two phase Locking with Serializable

<u>T1</u>	<u>T2</u>	<u>Result</u>
read_lock (Y);		X=50; Y=50
read_item (Y);		Nonserializable because it.
unlock (Y);		violated two-phase policy.
	read_lock (X);	
	read_item (X);	
	unlock (X);	
	write_lock (Y);	
	read_item (Y);	
	Y:=X+Y;	
	write_item (Y);	
	unlock (Y);	
write_lock (X);		
read_item (X);		
X:=X+Y;		
write_item (X);		
unlock (X);		

Time ↓

Fig. 14.5 Two phase Locking with Non-serializable



**T'1**

```

read_lock (Y);
read_item (Y);
write_lock (X);
unlock (Y);
read_item (X);
X:=X+Y;
write_item (X);
unlock (X);

```

**T'2**

```

read_lock (X);
read_item (X);
Write_lock (Y);
unlock (X);
read_item (Y);
Y:=X+Y;
write_item (Y);
unlock (Y);

```

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

**Fig. 14.6 Two phase Locking with Deadlock**

❖ **Two-Phase Locking Policy and the Two Locking Algorithms: Basic and Conservative**

The Two-Phase Locking (2PL) policy is a widely used protocol in database management systems to ensure serializability and avoid conflicts between concurrently executing transactions. It generates two different locking algorithms: Basic Two-Phase Locking and Conservative Two-Phase Locking. These algorithms differ mainly in when transactions acquire locks and how they manage the timing of lock requests.

➤ **Basic Two-Phase Locking (Basic 2PL)**

The Basic Two-Phase Locking protocol follows the standard 2PL rule, where each transaction goes through two phases: the growing phase (where it can acquire locks) and the shrinking phase (where it can only release locks). The critical characteristic of basic 2PL is that a transaction can acquire locks during the growing phase and release them during the shrinking phase, but it is not restricted from acquiring additional locks during the growing phase as long as no locks are released.

**Key Characteristics:**

1. **Growing Phase:** The transaction can acquire locks on the data items it requires, but it cannot release any locks during this phase.
2. **Shrinking Phase:** Once the transaction releases a lock, it enters the shrinking phase. After this point, the transaction is not allowed to acquire any new locks.

**Advantages:**

- **Simple and Flexible:** Basic 2PL is easy to implement and allows for flexibility in acquiring and releasing locks during the growing phase.
- **Ensures Serializability:** This approach guarantees that the transaction schedule will be serializable, as no transaction can acquire new locks after releasing any lock.

**Disadvantages:**

- **Deadlocks:** Since transactions are allowed to acquire locks until the point of release, this can lead to circular waiting situations, causing deadlocks.
- **Lock Contention:** Transactions that acquire and hold locks for long durations may block other transactions, leading to delays.

**➤ Conservative Two-Phase Locking (Conservative 2PL)**

The Conservative Two-Phase Locking protocol, sometimes referred to as Strict Two-Phase Locking, is a more restrictive version of basic 2PL. In this variant, a transaction must acquire all the locks it needs before it begins its operations, and it cannot release any locks until the very end of its execution. This means that the transaction enters the growing phase, acquires all required locks, and only after completing all its operations can it release the locks during the shrinking phase.

**Key Characteristics:**

1. **Lock Acquisition:** The transaction must acquire all the locks it will need for its entire execution before it starts performing any operations.
2. **No Lock Release:** The transaction does not release any locks until it has completed all its operations and is ready to commit.
3. **Shrinking Phase:** Similar to basic 2PL, after the locks are released, the transaction cannot acquire new locks.

**Advantages:**

- **Avoids Deadlocks:** Since the transaction acquires all locks at once and does not release any locks until the end, the potential for deadlocks is minimized.
- **Simpler Deadlock Resolution:** With all locks acquired upfront, there is no risk of circular waiting, making it easier to manage transaction synchronization.

**Disadvantages:**

- **Reduced Concurrency:** Because the transaction holds all locks for the entire duration, it may block other transactions from accessing the locked data items, leading to lower system throughput and reduced concurrency.
- **More Lock Contention:** Transactions that hold many locks for the entire duration increase lock contention, making the system less efficient, especially in high-concurrency environments.

**Table 14.1 Comparison of Basic 2PL and Conservative 2PL**

<b>Feature</b>	<b>Basic Two-Phase Locking</b>	<b>Conservative Two-Phase Locking</b>
Locking Behavior	Acquires locks as needed during the growing phase; can release locks during the growing phase.	Acquires all locks upfront before performing any operations and releases them only at the end.
Lock Acquisition	Locks are acquired progressively as the transaction executes.	All required locks are acquired at the beginning of the transaction.
Deadlock Risk	Higher risk due to flexible lock acquisition.	Lower risk due to the need to acquire all locks upfront.
Concurrency	Higher concurrency since locks are released during execution.	Lower concurrency due to holding all locks for the entire duration of the transaction.
Implementation Complexity	Easier to implement and more flexible.	More restrictive, but simpler in avoiding deadlocks.
Performance Impact	May have better performance due to reduced waiting times.	Can cause more lock contention and performance degradation.

While Basic Two-Phase Locking (2PL) offers flexibility and higher concurrency by allowing transactions to acquire locks as needed, it risks deadlocks and lock contention. On the other hand, Conservative Two-Phase Locking is more restrictive, avoiding deadlocks by acquiring all locks upfront but sacrificing concurrency. Each algorithm has its strengths and weaknesses, and the choice between them depends on the specific requirements of the application, including the need for concurrency, transaction complexity, and tolerance for deadlocks.

### **14.3. CONCURRENCY CONTROL BASED ON TIMESTAMP ORDERING:**

Concurrency Control Based on Timestamp Ordering (TO) ensures the serializability of transactions in a database system by using timestamps to enforce an order of transaction operations. This method guarantees that transactions are executed in a way that results in a serializable schedule, meaning the execution order of transactions will be equivalent to some serial order of those transactions.

There are various versions of Timestamp Ordering, including Basic Timestamp Ordering, Strict Timestamp Ordering, and Thomas's Write Rule. Each of these approaches modifies the basic timestamp ordering protocol to handle certain edge cases or to improve performance.

#### **➤ Basic Timestamp Ordering**

In Basic Timestamp Ordering (BTO), each transaction is assigned a unique timestamp when it starts. The system ensures that the order of operations adheres to the timestamp order, meaning the transaction with the earlier timestamp should execute its operations before transactions with later timestamps.

**1. Read Operation:**

- A transaction  $T_i$  can read data item  $X$  if and only if:
  - The last write operation on  $X$  occurred before  $T_i$ 's timestamp, i.e., the write timestamp of  $X$  must be earlier than  $T_i$ 's timestamp.
  - If  $T_i$  timestamp is later than the write timestamp of  $X$ , then the read operation is rejected, and  $T_i$  is aborted.

**2. Write Operation:**

- A transaction  $T_i$  can write to data item  $X$  if and only if:
  - No transaction that started after  $T_i$  has read or written to  $X$ . Specifically, the read timestamp and write timestamp of  $X$  must be less than  $T_i$ 's timestamp.
  - If there is a conflict (i.e., a read or write operation from another transaction with a later timestamp),  $T_i$  write operation is rejected and  $T_i$  is aborted.

**Disadvantages:**

- Transaction Abort: If a transaction attempts to perform an operation that conflicts with the timestamp order, it is aborted. This can lead to performance overhead due to frequent transaction rollbacks and restarts.
- System Load: Handling aborts and restarts for transactions can place significant load on the system, especially when conflicts are frequent.

**❖ Strict Timestamp Ordering**

Strict Timestamp Ordering (STO) is a stricter version of the basic timestamp ordering protocol that aims to prevent certain kinds of conflicts and reduce the chances of transaction abortion.

**1. Write-Read Conflict Handling:**

- In strict timestamp ordering, a transaction  $T_i$  can write to a data item  $X$  only if no other transaction has read  $X$  after  $T_i$  started. This guarantees that once a transaction has written a value, no other transaction can read it until the current transaction has been committed.

**2. Read-Write Conflict Handling:**

- A transaction  $T_i$  can read a data item  $X$  only if no other transaction has written to  $X$  after  $T_i$  started.

**➤ Strict Timestamp Ordering Rules:**

- 1. Read Operation:** A transaction  $T_i$  can read data item  $X$  if and only if the timestamp of  $T_i$  is greater than the write timestamp of  $X$ . Moreover, no transaction with a later timestamp can write to  $X$  after  $T_i$  has read it.

- 2. Write Operation:** A transaction  $T_i$  can write to data item  $X$  only if no other transaction has read or written to  $X$  after  $T_i$  started.

### Advantages of Strict Timestamp Ordering:

- Prevents Lost Updates: By ensuring that a transaction does not read a data item written by another transaction with a later timestamp, it prevents potential issues like lost updates.
- Consistency: Guarantees a higher level of consistency as it prevents any write-read conflicts from happening after a write.

### Disadvantages:

- Increased Abort Frequency: Because the rules are stricter, transactions are more likely to be aborted when there is contention, leading to reduced system throughput.
- Lower Concurrency: The system may suffer from reduced concurrency due to the more rigid constraints imposed by strict ordering.

### ❖ Thomas's Write Rule

Thomas's Write Rule is an optimization to the basic timestamp ordering protocol that seeks to reduce unnecessary transaction abortions by introducing a more relaxed condition for write operations. It was proposed by C. Thomas in 1978 to improve performance in systems where many transactions tend to be aborted under basic timestamp ordering.

- Thomas's Write Rule allows a write operation to proceed even if the write timestamp of a data item is less than the timestamp of the writing transaction, provided that no other transaction has read the data item after the current transaction's write operation.
- If a transaction  $T_i$  wants to write to data item  $X$ , and the write timestamp of  $X$  is less than  $T_i$ 's timestamp, but no other transaction has read  $X$  after  $T_i$  started, then the write operation will be allowed, even if it conflicts with a later timestamp.

This rule prevents unnecessary aborts of transactions and improves concurrency by allowing transactions to proceed with writes when the conflicts are not significant (i.e., no transaction has read the data).

### Example:

- Transaction  $T_1$  writes to data item  $X$ , with  $TS(T_1) = 100$ .
- Transaction  $T_2$  tries to write to  $X$ , but its  $TS(T_2) = 150$ .
- If no other transaction has read  $X$  after  $T_1$ 's write,  $T_2$ 's write operation will be allowed, even though its timestamp is later than  $T_1$ 's. According to basic timestamp ordering,  $T_2$  would be aborted, but Thomas's rule permits it.

**Advantages:**

- **Improved Performance:** By allowing certain write operations to proceed without aborting transactions, Thomas's Write Rule can significantly improve system throughput and reduce the frequency of transaction rollbacks.
- **Better Concurrency:** It allows more transactions to execute concurrently by relaxing the constraints on writes.

**Disadvantages:**

- **Potential Inconsistencies:** The rule may lead to situations where a transaction writes a value that later turns out to conflict with another transaction's operation. This could introduce inconsistencies if not handled carefully.

**Table 14.2 Summary of Timestamp Ordering Approaches**

Feature	Basic Timestamp Ordering (BTO)	Strict Timestamp Ordering (STO)	Thomas's Write Rule
Timestamp Assignment	Unique timestamp for each transaction.	Same as BTO.	Same as BTO.
Read Operation	Can read if $TS(T) >$ write timestamp.	Same as BTO, but stricter rules on subsequent writes.	Same as BTO.
Write Operation	Can write if $TS(T) >$ read and write timestamps.	Same as BTO, but stricter rules on subsequent reads.	Allows write even if write timestamp is earlier, as long as no read after.
Transaction Abort	Aborts transaction if conflicting operations occur.	More aborts than BTO due to stricter constraints.	Fewer aborts, more relaxed constraints.
Concurrency	Moderate, some aborts.	Lower due to strict rules.	Higher, allows more concurrency.

**14.4. MULTIVERSION CONCURRENCY CONTROL TECHNIQUES:**

This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected. Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

**❖ Multiversion technique based on timestamp ordering**

In MVCC with Timestamp Ordering, each transaction is assigned a unique timestamp when it begins, and the database maintains multiple versions of data items corresponding

to different transaction timestamps. This approach allows transactions to read older versions of data, which helps avoid conflicts that could arise if transactions were forced to wait for locks or the latest data.

### Algorithm

1. **Transaction Timestamp:** Each transaction  $T_i$  is assigned a unique timestamp  $TS(T_i)$  when it starts. The timestamp reflects the "logical time" of the transaction.
2. **Versioning of Data:** Each data item  $XXX$  can have multiple versions, each version being associated with a transaction that created it. The version has a timestamp indicating when the transaction modified the data item.
3. **Read and Write Operations:** Transactions can read older versions of a data item, but when writing, a new version of the data item is created with the transaction's timestamp.
4. **Visibility:** When a transaction reads a data item, it reads the most recent version of the item that was committed before the transaction's timestamp.

```
def read_operation(T_i, X):
```

```
    if  $TS(T_i) < W(X)$ : # If transaction's timestamp is earlier than the write timestamp
        return False # Abort the transaction
```

```
    else:
```

```
        read_version = get_version(T_i, X) # Find the most recent committed version before
         $TS(T_i)$ 
```

```
        return read_version
```

```
def write_operation(T_i, X):
```

```
    if  $TS(T_i) < W(X)$ : # Transaction timestamp must be later than last write timestamp
        return False # Abort the transaction
```

```
    else:
```

```
        if  $TS(T_i) > R(X)$ : # Make sure no transaction has read X after  $TS(T_i)$ 
```

```
            create_new_version(T_i, X) # Create a new version with  $T_i$ 's timestamp
```

```
            return True
```

```
    else:
```

```
        return False # Abort transaction, conflict occurs
```

```
def commit_transaction(T_i):
```

```
    if not conflicts_with_other_transactions(T_i):
```

```
        mark_as_committed(T_i)
```

```
        return True # Transaction successfully committed
```

```
    else:
```

```
        return False # Transaction aborted
```

```
def abort_transaction(T_i):
    rollback_operations(T_i)
    return False # Transaction aborted
```

### Conditions for Serializability

The goal of **MVCC with Timestamp Ordering** is to ensure **serializability** by respecting the following conditions:

1. **Write-Write Conflicts:** A transaction should only write to a data item if no other transaction that started after it has written to the same item.
2. **Read-Write Conflicts:** A transaction should not read a data item written by a transaction that started after it.
3. **Read-Read Conflicts:** Multiple transactions can read the same data item concurrently without conflict, as long as no transaction is writing to it.

By following these rules, the system ensures that the **schedule of transactions** generated by MVCC is **serializable**, meaning that the outcome is equivalent to the outcome of some serial execution order of transactions.

The **MVCC with Timestamp Ordering** algorithm is an effective concurrency control technique that ensures **serializability** by enforcing an ordering based on transaction timestamps. It allows for high concurrency by permitting transactions to access multiple versions of data concurrently while avoiding blocking. However, the system's complexity and the need for version management must be carefully considered when implementing this technique in large-scale database systems.

#### ➤ Multiversion Two-Phase Locking Using Certify Locks

Multiversion Two-Phase Locking (MV2PL) is an extension of the traditional Two-Phase Locking (2PL) protocol that combines the benefits of MVCC with the locking protocol. In MV2PL, the system maintains multiple versions of data items and uses certify locks to ensure serializability and prevent conflicts while allowing higher concurrency.

#### Steps:

1. X is the committed version of a data item.
2. T creates a second version X' after obtaining a write lock on X.
3. Other transactions continue to read X.
4. T is ready to commit so it obtains a certify lock on X'.
5. The committed version X becomes X'.
6. T releases its certify lock on X', which is X now.



Compatibility tables for

	Read	Write		Read	Write	Certify
Read	yes	no	Read	yes	no	no
Write	no	no	Write	no	no	no
			Certify	no	no	no
read/write locking scheme			read/write/certify locking scheme			

**Fig. 14.7 Compatibility table for Locking Schemes**

#### 14.5. VALIDATION CONCURRENCY CONTROL TECHNIQUES::

**Validation Concurrency Control (VCC)** is a technique used to manage concurrency in database systems by validating transactions at the end of their execution, rather than using locking mechanisms or timestamps. This method helps avoid conflicts between transactions by ensuring that transactions are **serializable** while maintaining high concurrency. VCC is based on the idea that each transaction goes through three phases: **Read Phase**, **Validation Phase**, and **Write Phase**.

##### Three Phases of Validation Concurrency Control

###### 1. Read Phase:

- In the **read phase**, a transaction is allowed to **read** data without any restrictions. During this phase, the transaction can perform any number of **read operations** on data items, and no validation or checking is done on whether the transaction will eventually conflict with others.
- The transaction continues to read the database until it reaches the end of the read phase or wants to perform a write operation. The key here is that the transaction is not yet considered to have made any modifications to the database, so no conflicts are checked during the reading process.
- During this phase, other transactions can continue reading or writing to the database, allowing for high concurrency and minimal blocking.

###### 2. Validation Phase:

- Once the transaction finishes reading and is about to perform write operations, it enters the **validation phase**. In this phase, the system checks whether the transaction's actions are **conflict-free** and if the transaction can be committed without violating the database's consistency.
- The validation process ensures that:
  - **No write-write conflicts:** If the transaction wants to write to a data item, it must verify that no other transaction has already written to the same data item during the read phase.
  - **No read-write conflicts:** If the transaction reads a data item, it must ensure that no other transaction has written to that item after the transaction began reading it.

- If conflicts are found, the transaction is **aborted** and rolled back to ensure the database remains consistent.

In essence, the validation phase is responsible for checking whether the transaction's operations can be serialized in a way that maintains consistency and does not cause any inconsistencies in the final database state.

### 3. Write Phase:

- If the transaction passes the validation phase, it enters the **write phase**, where it can **commit** and apply its changes to the database.
- The changes made by the transaction are written to the database, and it is considered **committed**. At this point, the transaction's updates become visible to other transactions.
- If the transaction fails the validation phase (because of conflicts), it is aborted and must restart. If a restart occurs, the transaction will go back to the read phase and reattempt its operations.

### Example of Validation Concurrency Control

Consider the following transactions:

- **Transaction T1:** Reads data items A and B, then attempts to write to A.
- **Transaction T2:** Reads data item B, then attempts to write to B.

### Step-by-Step Process:

#### 1. T1 and T2 begin execution:

- Both transactions read A and B in the **read phase**.
- **T1** reads A and B.
- **T2** reads B.

#### 2. Validation Phase:

- **T1's Validation:** T1 wants to write to A. It checks if another transaction (like T2) has written to A after T1's read. If T2 wrote to A, T1 would be invalidated and aborted.
- **T2's Validation:** T2 wants to write to B. It checks if another transaction (like T1) has written to B after T2's read. If T1 wrote to B, T2 would be invalidated and aborted.

#### 3. Write Phase:

- If both transactions pass validation, they move to the **write phase** and commit their changes.

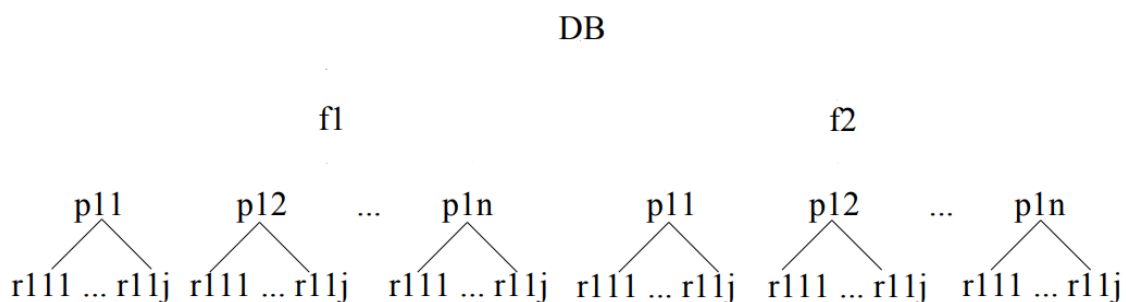
Validation Concurrency Control techniques ensure serializability by **validating transactions** during a **validation phase** before committing changes to the database. This method allows for high concurrency, as transactions can read data without blocking and can only be validated at the end of their execution. While this can lead to significant performance improvements, especially in systems with low contention, it may result in frequent transaction abortions and rollbacks in environments with high conflict rates. Therefore, the effectiveness of VCC depends largely on the nature of the workload and the frequency of conflicts between transactions.

#### 14.6. GRANULARITY OF DATA ITEMS AND MULTIPLE GRANULARITY LOCKING:

**Multiple Granularity Locking** is a locking protocol that allows transactions to acquire locks at different levels of granularity simultaneously. The key idea behind this technique is to allow transactions to lock data at a **coarse granularity** when high concurrency is not necessary, and at a **fine granularity** when access to specific data items needs to be controlled more carefully.

For example, a transaction could lock an entire table when it wants to perform broad operations like reading or writing a range of rows. At the same time, it could lock individual rows or fields when it needs to perform more specific operations on particular records or attributes.

The **Multiple Granularity Locking** approach reduces the overhead associated with fine-grained locking by allowing coarser locks when possible, which minimizes the need for a large number of locks. However, it still provides the necessary isolation and concurrency when more detailed control is required.



**Fig. 14.8 A hierarchy of granularity from coarse (database) to fine (record).**

To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:

- Intention-shared (IS): indicates that a shared lock(s) will be requested on some descendent nodes(s).

- Intention-exclusive (IX): indicates that an exclusive lock(s) will be requested on some descendent node(s).
- Shared-intention-exclusive (SIX): indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

	IS	IX	S	SIX	X	Intention-shared (IS) Intention-exclusive (IX) Shared-intention-exclusive (SIX)
IS	yes	yes	yes	yes	no	
IX	yes	yes	no	no	no	
S	yes	no	yes	no	no	
SIX	yes	no	no	no	no	
X	no	no	no	no	no	

**Fig. 14.9 compatibility matrix: Multiple Granularity Locking**

The set of rules which must be followed for producing serializable schedule are

1. The lock compatibility must adhered to.
2. The root of the tree must be locked first, in any mode..
3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.
4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
6. T can unlock a node, N, only if none of the children of N are currently locked by T.

#### 14.7. SUMMARY:

The **Introduction to Transaction Processing Concepts and Theory** covers the fundamental principles behind managing transactions within a database management system (DBMS). It begins by defining **transaction processing** and explaining key concepts, such as **transactions** (units of work) and **systems** (the environment in which transactions execute). The chapter highlights the **desirable properties of transactions**, including **atomicity**, **consistency**, **isolation**, and **durability** (ACID properties), which ensure the integrity and reliability of transactions. It also delves into **recoverability**, outlining how schedules (sequences of operations) are classified to prevent inconsistencies, and **serializability**, which ensures that interleaved transactions produce results equivalent to serial executions. Overall, the chapter introduces essential concepts for ensuring that multiple transactions can be executed concurrently without compromising database consistency and correctness.

**14.8. TECHNICAL TERMS:**

- Serializability
- Concurrency Control
- Two-Phase locking
- Lock Granularity
- Lock Compatibility Matrix

**14.9. SELF ASSESSMENT QUESTIONS:****Essay questions:**

- 1) Explain the working of the Two-Phase Locking protocol and its role in ensuring serializability.
- 2) Describe Basic Timestamp Ordering, Strict Timestamp Ordering, and Thomas's Write Rule with examples.
- 3) Discuss the implementation of Multiversion Concurrency Control based on Timestamp Ordering and its advantages over traditional locking mechanisms.
- 4) Explain the phases of Validation Concurrency Control and evaluate its effectiveness in low-conflict environments.
- 5) Analyze the concept of granularity in data items and the working of multiple granularity locking, including its benefits and challenges.

**Short Notes:**

- 1) What are the two phases of the Two-Phase Locking protocol?
- 2) What is the purpose of timestamps in concurrency control?
- 3) How does MVCC improve performance in read-heavy environments?
- 4) Name the three phases of Validation Concurrency Control.
- 5) What is the role of a lock hierarchy in multiple granularity locking?

**14.10. 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". 8<sup>th</sup> Edition. Addison-Wesley.
- 3) Silberschatz, A., Korth, H.F., & Sudarshan, S. (2010). "Database System Concepts". 6<sup>th</sup> Edition. McGraw-Hill.
- 4) Ullman, J.D., & Widom, J. (2008). "A First Course in Database Systems". 3<sup>rd</sup> Edition. Pearson.

## LESSON-15

# DISTRIBUTED DATABASES AND CLIENT SERVER ARCHITECTURES

### AIMS AND OBJECTIVES:

The chapter aims to provide a comprehensive understanding of distributed database concepts, design principles, and client-server architectures. It focuses on equipping learners with the knowledge to design, manage, and optimize distributed database systems while understanding the significance of client-server architectures in modern data management.

By the end of this chapter, learners should be able to:

- 1) Understand Distributed Database Concepts
- 2) Explain Data Fragmentation, Replication, and Allocation Techniques
- 3) Identify Types of Distributed Database Systems
- 4) Analyze Distributed Database Design Techniques
- 5) Discuss 3-Tier Client-Server Architecture

### STRUCTURE:

#### 15.1. Introduction

#### 15.2. Distributed Database Concepts

#### 15.3. Data Fragmentation, Replication and Allocation

#### 15.4. Types of Distributed Database Systems

#### 15.5. 3-Tier Client-Server Architecture

#### 15.6. Summary

#### 15.7. Technical Terms

#### 15.8. Self-Assessment Questions

#### 15.9. Suggested Readings

### 15.1. INTRODUCTION:

Distributed databases and client-server architectures are pivotal in modern data management, enabling organizations to manage and access data efficiently across multiple locations. A **distributed database** is a system where data is stored across multiple sites or nodes, interconnected through a network, yet appears to users as a single cohesive database. This approach ensures scalability, fault tolerance, and location transparency, making it ideal for enterprises operating on a global scale. Key concepts such as **data fragmentation**, **replication**, and **allocation techniques** form the foundation of distributed database design, enabling optimal distribution of data to balance performance, reliability, and cost.

In addition to distributed databases, the chapter explores **client-server architectures**, focusing on the widely adopted **3-tier client-server model**. This architecture divides system functionality into three layers: presentation, application, and data, ensuring better scalability, maintainability, and performance. Learners will also examine various types of distributed database systems, including **homogeneous** and **heterogeneous systems**, to understand their differences and applications. By integrating distributed database concepts with client-server designs, this chapter provides a robust framework for building resilient and efficient data management systems suited to modern

## 15.2. DISTRIBUTED DATABASE CONCEPTS:

A **Distributed Database System (DDBS)** is a collection of multiple, logically interrelated databases distributed across various locations and interconnected by a communication network. Unlike centralized systems, where all data resides at a single site, distributed databases distribute data and processing across multiple nodes while maintaining the appearance of a single, unified database to users.

### Key Characteristics of Distributed Database Systems:

1. **Distribution of Data:** Data is physically stored across multiple locations, enabling access and processing closer to the point of use.
2. **Transparency:**
  - **Location Transparency:** Users can query the database without knowing its physical location.
  - **Replication Transparency:** Users are unaware of data replication across sites.
  - **Failure Transparency:** The system can recover from site or network failures without affecting user operations.
  - **Concurrency Transparency:** Multiple users can perform operations simultaneously without interference.
3. **Scalability:** Distributed databases can scale horizontally by adding more nodes to the system, handling growing amounts of data and users.
4. **Autonomy:** Different sites in a distributed database system can operate independently, managing their local data while participating in the larger network.

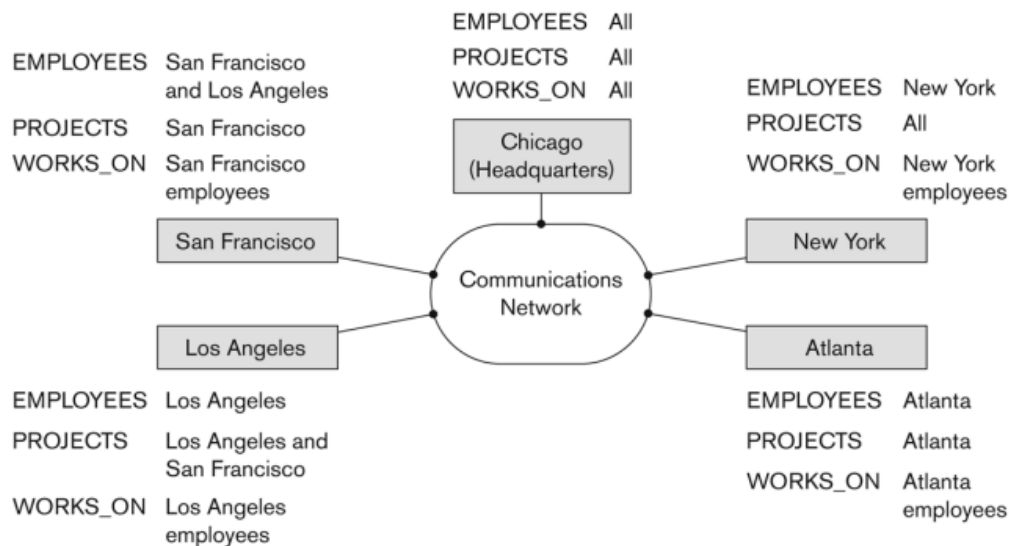
### Components of a Distributed Database System

1. **Database Servers:** Manage and store data at individual sites.
2. **Communication Network:** Connects the sites and enables data exchange between them.

- 3. Distributed Database Management System (DDBMS):** A software layer that manages distributed databases, ensuring consistency, transparency, and efficient data access.

### Advantages of Distributed Database Systems

- 1. Improved Reliability:** Even if a site fails, other sites can continue to function.
- 2. Enhanced Performance:** By processing data locally, distributed databases reduce network traffic and improve query response times.
- 3. Scalability:** Adding new nodes to a distributed database is simpler than expanding a centralized system.
- 4. Data Localization:** Data can be stored close to where it is most frequently accessed, reducing latency.



**Fig. 15.1 Data Distribution and Replication**

### Challenges of Distributed Database Systems:

- 1. Complexity:** Designing, managing, and maintaining distributed databases is more complex than centralized systems.
- 2. Synchronization Issues:** Keeping data consistent across multiple sites requires sophisticated protocols.
- 3. Communication Overhead:** The need for frequent data exchanges between sites can increase network traffic and response times.
- 4. Security Concerns:** Distributing data across multiple sites increases the attack surface, making security management more challenging.



### Applications for Distributed Database Systems:

- 1. Global Enterprises:** Companies with offices worldwide use distributed databases to store and manage local and global data seamlessly.
- 2. Cloud-Based Systems:** Distributed databases underpin many cloud storage solutions, ensuring high availability and scalability.
- 3. E-Commerce Platforms:** Distributed databases support real-time inventory updates, customer data storage, and transaction management.

By understanding distributed database systems, organizations can design robust, efficient, and scalable data management solutions to meet the demands of modern applications.

### 15.3. DATA FRAGMENTATION, REPLICATION, AND ALLOCATION:

In distributed database design, data fragmentation, replication, and allocation are critical techniques for organizing data across multiple sites to optimize performance, reliability, and availability.

#### ❖ Data Fragmentation

Data fragmentation involves dividing a database into smaller, logical pieces called fragments, which are distributed across various sites. Fragmentation improves performance by enabling queries to access only relevant subsets of data and ensures that data is stored closer to where it is needed.

#### Types of Data Fragmentation

##### ➤ Horizontal Fragmentation:

- Divides a relation (table) into subsets of rows (tuples) based on specified predicates.
- Each fragment contains a subset of rows, and collectively, the fragments represent the entire table.
- Example:  
A table EMPLOYEE can be horizontally fragmented by DEPARTMENT:
  - EMPLOYEE\_DEPT1: All employees in department 1.
  - EMPLOYEE\_DEPT2: All employees in department 2.

##### ➤ Vertical Fragmentation:

- Divides a relation into subsets of columns (attributes), with a common key included in each fragment to enable reconstruction.
- Example:  
A table EMPLOYEE(ID, Name, Salary, Department) can be vertically fragmented into:
  - Fragment 1: EMPLOYEE\_BASIC(ID, Name)
  - Fragment 2: EMPLOYEE\_JOB(ID, Salary, Department)

➤ **Mixed (Hybrid) Fragmentation:**

- Combines horizontal and vertical fragmentation.
- A relation is first horizontally fragmented, and the resulting fragments are further divided vertically (or vice versa).
- **Example:**  
A table EMPLOYEE is horizontally fragmented by DEPARTMENT, and each horizontal fragment is vertically fragmented into attributes Name and Salary.

**Representation of Fragmentation**

➤ **Horizontal Fragmentation Representation**

A relation R can be divided into fragments R1, R2, ..., Rn, based on predicates:

- $R1 = \sigma(p1)(R)$
- $R2 = \sigma(p2)(R)$
- ...

Where  $R = R1 \cup R2 \cup \dots \cup Rn$ .

- **Complete horizontal fragmentation:** A set of horizontal fragments whose conditions C1, C2, ..., Cn include all the tuples in R- that is, every tuple in R satisfies (C1 OR C2 OR ... OR Cn).
- **Disjoint complete horizontal fragmentation:** No tuple in R satisfies (Ci AND Cj) where  $i \neq j$ .
- To reconstruct R from horizontal fragments a UNION is applied.

➤ **Vertical Fragmentation Representation**

A relation R is divided into fragments R1 and R2:

- $R1 = \pi(A1, A2, K)(R)$
- $R2 = \pi(A3, A4, K)(R)$

Where K is the primary key to reconstruct R:

- $R = R1 \bowtie R2$ .
- Complete vertical fragmentation v A set of vertical fragments whose projection lists L1, L2, ..., Ln include all the attributes in R but share only the primary key of R. In this case the projection lists satisfy the following two conditions:

$$L1 \cup L2 \cup \dots \cup Ln = \text{ATTRS}(R)$$

$Li \cap Lj = \text{PK}(R)$  for any  $i, j$ , where  $\text{ATTRS}(R)$  is the set of attributes of R and  $\text{PK}(R)$  is the primary key of R.

To reconstruct R from complete vertical fragments a OUTER UNION is applied.

### ➤ **Mixed (Hybrid) Fragmentation Representation**

A relation R is first horizontally fragmented, then each horizontal fragment is vertically fragmented:

- Horizontal:  $R1 = \sigma(p1)(R)$ ,  $R2 = \sigma(p2)(R)$
- Vertical:  $R1a = \pi(A1, A2, K)(R1)$ ,  $R1b = \pi(A3, A4, K)(R1)$ .

By effectively combining fragmentation, replication, and allocation, distributed database systems achieve a balance between performance, reliability, and resource utilization.

### ❖ **Data Replication**

Replication involves creating and maintaining copies of fragments at multiple sites. Replication ensures high availability, fault tolerance, and improved query performance by providing local access to frequently used data.

### **Types of Replication**

1. Full Replication:
  - Every site contains a complete copy of the database.
  - Improves availability but introduces high storage and update synchronization overhead.
2. Partial Replication:
  - Only selected fragments are replicated across sites based on usage patterns.
  - Balances availability and storage requirements.
3. No Replication:
  - Each fragment is stored at only one site.
  - Reduces storage and synchronization cost but may impact availability.

### ❖ **Data Allocation**

Data allocation determines how fragments (or replicated fragments) are distributed across sites in the network.

### **Strategies for Allocation**

1. Centralized Allocation:
  - All fragments are stored at a single site.
  - Simple to manage but lacks the benefits of distribution.
2. Partitioned Allocation:
  - Fragments are distributed across multiple sites, with each fragment located at one site.
3. Replicated Allocation:
  - Combines fragmentation and replication to place copies of fragments at multiple sites.

#### 15.4. TYPES OF DISTRIBUTED DATABASE SYSTEMS:

Distributed database systems can be categorized based on the degree of homogeneity among the participating systems and how they manage interactions across multiple databases.

##### ❖ Homogeneous Distributed Database Systems

In a **homogeneous distributed database system**, all participating sites use the same DBMS software, schema, and structure, making them consistent in data representation and management.

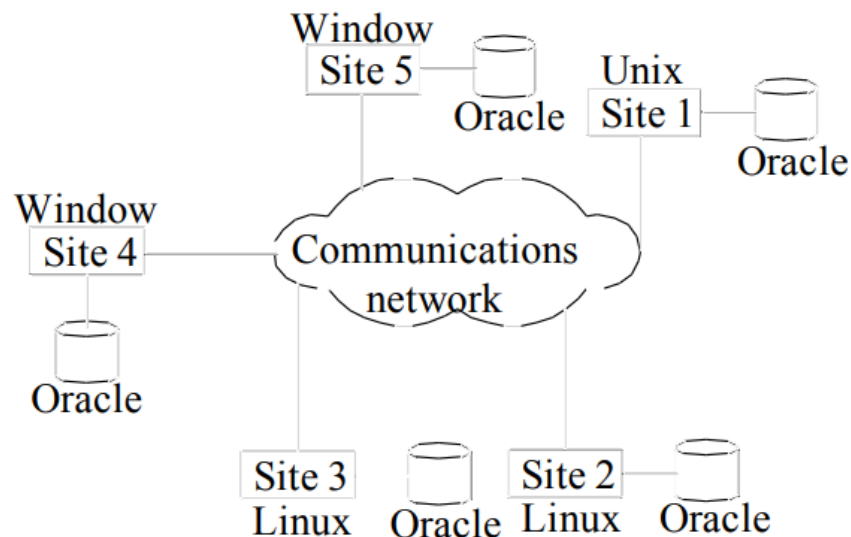


Fig. 15.2 Homogeneous Distributed Database Systems

##### Characteristics:

- Same DBMS across all sites.
- Uniform schema structure.
- Simplified communication and data exchange.
- Easy integration and management.

##### Advantages:

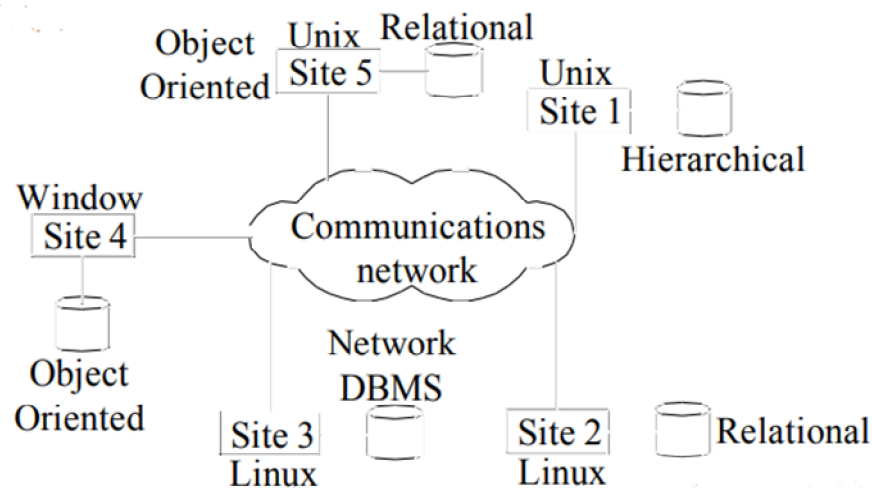
- Easier to maintain consistency across sites.
- Simplified query processing due to uniform schema.
- Low complexity in system integration and management.

##### Disadvantages:

- Limited flexibility; all sites must use the same DBMS software.
- Less suitable for integrating diverse systems.

#### 2. Heterogeneous Distributed Database Systems

In a **heterogeneous distributed database system**, the participating sites may use different DBMS software, schemas, or data models, leading to variations in data representation and processing.



**Fig. 15.3 Heterogeneous Distributed Database Systems**

**Characteristics:**

- Different DBMS software and schemas across sites.
- Complex communication due to data model differences.
- Requires middleware or translators for compatibility.

**Advantages:**

- Allows integration of diverse systems, including legacy databases.
- Flexible in accommodating different database technologies.

**Disadvantages:**

- Higher complexity in system integration and maintenance.
- Query processing is more challenging due to schema and DBMS differences.
- Increased overhead for ensuring data consistency and compatibility.

❖ **Federated Database Management Systems (FDBMS)**

A **federated database management system** integrates multiple autonomous databases, either homogeneous or heterogeneous, into a single unified system while preserving their autonomy.

**Types of Federated Systems:**

- **Loosely Coupled:** Independent systems with minimal integration, often requiring manual coordination.
- **Tightly Coupled:** Higher integration with centralized schema and coordination.

**Advantages:**

- Allows databases to operate independently while supporting integrated queries.
- Facilitates data sharing across diverse systems.
- Can evolve incrementally as new databases are added.

**Disadvantages:**

- Query processing is complex due to heterogeneity and autonomy.
- Performance issues can arise from additional layers of coordination.
- Maintaining global consistency is challenging.

**Issues in Federated Database Management Systems****1. Autonomy:**

- Balancing the autonomy of individual databases with the need for integration.
- Ensuring that local database operations are not disrupted by global transactions.

**2. Schema Integration and Mapping:**

- Resolving schema conflicts (naming, structural, and semantic) across databases.
- Designing global schemas that represent integrated data without losing local schemas' specificity.

**3. Query Processing:**

- Translating global queries into local queries that respect each database's schema and DBMS.
- Optimizing query performance while minimizing communication overhead.

**4. Data Consistency:**

- Synchronizing updates across databases to maintain consistency, especially in heterogeneous systems.
- Addressing replication conflicts when data is updated simultaneously at multiple sites.

**5. Security and Access Control:**

- Implementing unified security policies while respecting the autonomy of local databases.
- Managing user authentication and access rights across systems.

**6. Performance:**

- Handling the additional processing overhead introduced by the middleware or coordination layer.
- Minimizing network delays in distributed query execution.

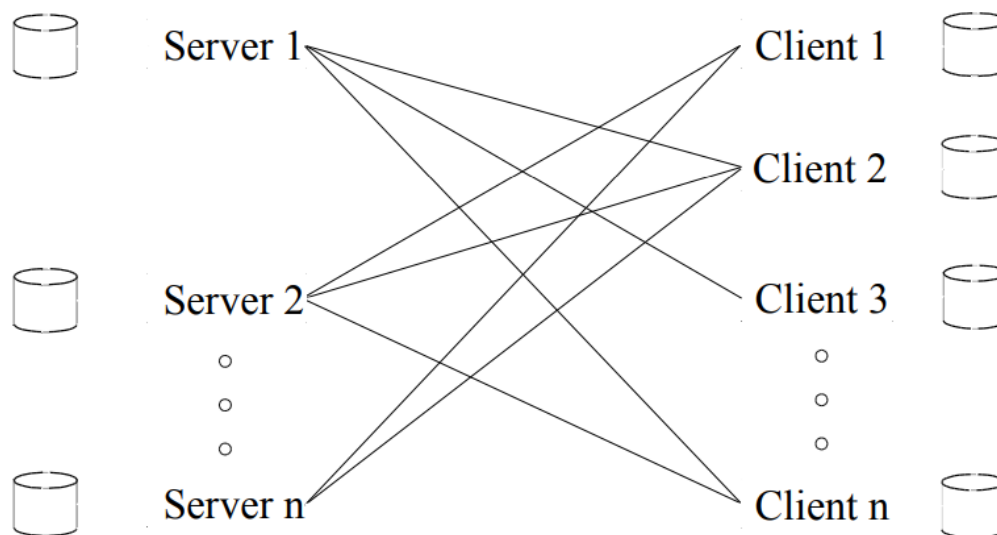
**7. Fault Tolerance and Recovery:**

- Ensuring the federated system remains operational despite failures at individual sites.
- Coordinating recovery mechanisms across autonomous databases.

By addressing these issues, federated systems aim to provide an effective framework for integrating diverse databases while maintaining their autonomy and reliability.

### 15.5. CLIENT-SERVER DATABASE ARCHITECTURE:

It consists of clients running client software, a set of servers which provide all database functionalities and a reliable communication infrastructure.



**Fig. 15.4 Client-Server Database Architecture**

- Clients reach server for desired service, but server does not reach clients.
- The server software is responsible for local data management at a site, much like centralized DBMS software.
- The client software is responsible for most of the distribution function.
- The communication software manages communication among clients and servers.

#### **The processing of a SQL query goes as follows:**

- Client parses a user query and decomposes it into several independent sub-queries. Each subquery is sent to appropriate site for execution.
- Each server processes its query and sends the result to the client.
- The client combines the results of subqueries and produces the final result.

The **3-tier client-server architecture** in the context of a **Database Management System (DBMS)** is a model that divides an application into three logical layers: **the presentation layer**, **the application logic layer**, and **the data layer**. This separation ensures that each layer handles specific responsibilities, improving the scalability, maintainability, and efficiency of complex database-driven applications. In this architecture, the database server is separated from the client interface, and business logic is processed in a middle layer, creating

a more modular system. The 3-tier architecture is widely used in modern DBMS applications, especially for web and enterprise systems.

- 1. Presentation Layer (Client Tier):** The presentation layer is responsible for providing the user interface (UI) and enabling users to interact with the application. This layer acts as the client-side component where users input data and view results. It communicates with the application layer (the middle tier) to process requests and display results. In the context of a DBMS, this layer could be a web browser, a desktop application, or a mobile app. The presentation layer does not directly interact with the database; instead, it sends requests to the application server, which processes them and retrieves the necessary data from the database layer.
- 2. Application Layer (Logic Tier):** The application layer, also known as the middle tier or business logic layer, processes the requests received from the presentation layer. This layer contains the business logic, rules, and algorithms that govern how data is manipulated, validated, and managed within the system. It acts as an intermediary between the presentation and data layers. When the client sends a request, the application server processes it, performs calculations or logic, and interacts with the database to retrieve or manipulate data as required. This separation of logic ensures that changes to the database or UI can be made independently without affecting the other layers.
- 3. Data Layer (Database Tier):** The data layer is the foundational tier in which the database resides. It stores and manages all the persistent data used by the application. The data layer is responsible for handling database queries, data retrieval, data manipulation (insert, update, delete), and ensuring data consistency, integrity, and security. This layer typically uses relational databases (such as MySQL, Oracle, or SQL Server) or NoSQL databases (such as MongoDB) to store application data. The application layer communicates with the data layer to execute SQL queries or API calls to retrieve or modify data based on the logic implemented in the middle layer.

#### **Key Features and Benefits of 3-Tier Client-Server Architecture in DBMS:**

- **Modularity:** By separating the presentation, logic, and data layers, the architecture allows for modular development and easier maintenance. Each layer can be developed, tested, and updated independently.
- **Scalability:** Each layer can be scaled independently. For example, the data layer can be scaled to accommodate large databases, the application layer can be scaled for complex business logic, and the presentation layer can handle multiple client interfaces.
- **Security:** The 3-tier architecture enhances security by isolating the database layer from direct access by the client. Security mechanisms such as authentication, access control, and encryption can be applied at the middle layer to control data access.



- **Flexibility:** The architecture allows for the use of different technologies in each tier. For example, the client can use a web browser while the application server might run on a Java-based platform, and the database server could use a relational or NoSQL database.
- **Load Balancing:** The middle layer can be designed to distribute requests across multiple servers or instances, ensuring that the system can handle high traffic volumes and perform efficiently under load.
- **Maintainability and Upgrades:** With clearly separated layers, updates to one part of the system (e.g., upgrading the database or changing the UI) can be done without impacting the other layers, making the system more maintainable and easier to upgrade.

In DBMS, this 3-tier architecture is particularly valuable for large applications that require robust data management and user interaction, such as e-commerce platforms, enterprise resource planning (ERP) systems, and cloud-based applications. By isolating the data access layer from the user interface, the architecture ensures better performance, security, and scalability for database-driven applications.

#### 15.6. SUMMARY:

This chapter introduces the key concepts behind distributed databases and client-server architectures. It explains distributed database systems, where data is stored across multiple locations to improve performance, reliability, and availability. Key design techniques like data fragmentation (dividing data into smaller parts), replication (creating copies of data for redundancy), and allocation (determining where data should be stored) are discussed as methods to enhance database efficiency.

The chapter also covers different types of distributed database systems, including homogeneous systems (same DBMS across all sites), heterogeneous systems (different DBMSs at different sites), and federated systems (integrating independent databases). Finally, it provides an overview of the 3-tier client-server architecture, which separates the user interface, application logic, and database management into distinct layers, improving scalability and modularity in distributed applications.

#### 15.7. TECHNICAL TERMS:

- Distributed Database
- data fragmentation
- homogeneous systems
- heterogeneous systems
- client-server architecture
- modularity in distributed applications

#### 15.8. SELF ASSESSMENT QUESTIONS:

##### Essay questions:

- 1) Discuss the challenges and advantages of implementing a distributed database system compared to a centralized database?

- 2) Explain the different types of data fragmentation (horizontal, vertical, and mixed) and their impact on distributed database design?
- 3) Compare and contrast homogeneous and heterogeneous distributed database systems, and explain when each type is appropriate to use?
- 4) Analyze the role of data replication in distributed database systems, focusing on its impact on performance, consistency, and fault tolerance?
- 5) Provide an in-depth explanation of the 3-tier client-server architecture, discussing how it supports scalability and modularity in distributed systems?

**Short Notes:**

- 1) What is a distributed database system, and why is it important?
- 2) Explain the concept of data fragmentation in distributed databases.
- 3) What are the main differences between homogeneous and heterogeneous distributed database systems?
- 4) What is the role of data replication in a distributed database?
- 5) Describe the components of a 3-tier client-server architecture?

**15.9. 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". 8<sup>th</sup> Edition. Addison-Wesley.
- 3) Silberschatz, A., Korth, H.F., & Sudarshan, S. (2010). "Database System Concepts". 6<sup>th</sup> Edition. McGraw-Hill.
- 4) Ullman, J.D., & Widom, J. (2008). "A First Course in Database Systems". 3<sup>rd</sup> Edition. Pearson.

**Mrs. Appikatla Pushpa Latha**